
jetzt lerne ich
PHP & MySQL
Der leichte Einstieg in die dynamische Webseiten-Programmierung
Markt+Technik
GIESBERT DAMASCHKE
Kurz und knapp – locker und verständlich geschrieben
Komplette Webprojekte auch für den Einsteiger
Alle wichtigen Techniken von Arrays bis Objekte
ALWAYS LEARNING
PEARSON

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.dnb.de> abrufbar.

10 9 8 7 6 5 4 3 2 1

14 13 12

ISBN 978-3-8272-4574-8 Print; 978-3-86325-529-9 PDF; 978-3-86325-128-4 ePub

Covergestaltung: Thomas Arlt, tarlt@adesso21.net
Lektorat: Boris Karnikowski, bkarnikowski@pearson.de
Fachlektorat: Bettina Ramm, die-web-architektin.de
Korrektorat: Gaby Meyer, München
Herstellung: Martha Kürzl-Harrison, mkuerzl@pearson.de
Satz: text&form GbR, Fürstenfeldbruck
Druck und Verarbeitung: Drukarnia Dimograf, Bielsko-Biala
Printed in Poland

3 Entscheidungen und Schleifen

SIE LERNEN IN DIESEM KAPITEL:

- WAS LOGISCHE WAHRHEITSWERTE SIND
- WIE SIE IN PHP ENTSCHEIDUNGEN TREFFEN
- WIE SIE PROGRAMMTEILE BELIEBIG OFT AUSFÜHREN KÖNNEN

Unsere Scripte sind inzwischen schon ordentlich gewachsen, allerdings sind sie immer noch ziemlich dumm. Bislang können sie keine Entscheidungen treffen oder bestimmte Aufgaben mehrfach durchführen. Was uns noch fehlt, ist die Fähigkeit, den Programmablauf anhand verschiedener Kriterien zu steuern und zu kontrollieren. Doch das ändern wir jetzt.

Bevor wir uns allerdings mit den verschiedenen Möglichkeiten beschäftigen, in PHP Entscheidungen zu fällen, müssen wir einen kleinen Abstecher in die Logik machen. Denn alle Entscheidungen, die Sie mit PHP treffen, hängen davon ab, ob eine bestimmte Aussage bzw. Bedingung logisch wahr oder logisch falsch ist.

3.1 Wahr und falsch

Die Basis aller Entscheidungen ist die Frage, ob eine Aussage wahr (`TRUE`) oder falsch (`FALSE`) ist. In Abhängigkeit vom Wahrheitswert einer Aussage kann ein Script dann entscheiden, ob es eine bestimmte Anweisung ausführt oder nicht. Einige Beispiele:

- Ist heute denn schon Weihnachten? Wenn ja, dann blende auf der Startseite einen Weihnachtsgruß ein.
- Ist der Besucher der Webseite eingeloggt? Wenn ja, begrüße ihn und zeige sein individuelles Menü an.
- Entspricht der übergebene Wert den Vorgaben? Wenn nicht, fordere den Besucher zur erneuten Eingabe auf.
- Hat der Besucher Produkte in seinem Einkaufskorb? Wenn ja, zeige den Einkaufskorb an.
- Wurden Werte in das Formular eingegeben? Wenn nicht, zeige das Formular an.

Diese und ähnliche Fragen werden in der Regel durch Vergleiche zweier Werte bzw. Aussagen und durch logische Verknüpfungen verschiedener Aussagen formuliert.

3.1.1 Vergleiche

- **Kleiner als:** Die linke spitze Klammer ist der Operator für den Vergleich »kleiner als«; `$a < $b` bedeutet also: Der Wert der Variablen `$a` ist kleiner als der Wert der Variablen `$b`.
- **Größer als:** Die rechte spitze Klammer ist der Operator für »größer als«; `$a > $b` bedeutet also: Der Wert der Variablen `$a` ist größer als der Wert der Variablen `$b`.

- **Gleich:** Möchten Sie ausdrücken, dass zwei Variablen den gleichen Wert besitzen, benutzen Sie dafür zwei (!) Gleichheitszeichen: `$a == $b`. Bei der Überprüfung wird nur der Wert der Variablen verglichen, nicht ihr Typ. Ein String mit dem Wert `10` – etwa: `$a = "10 Eier"` – ist für PHP also mit einer Integerzahl vom Wert `10` (`$b = 10`) gleich.

> **Vorsicht, Falle!**
>
> Die Verwechslung des Zuweisungsoperators – dem einfachen Gleichheitszeichen – mit der Abfrage der Gleichheit – dem doppelten Gleichheitszeichen – ist eine der häufigsten Fehlerquellen bei der Programmierung mit PHP.

- **Identität:** Wenn zwei Variablen nicht nur den gleichen Inhalt, sondern auch noch den gleichen Typ haben, sind sie identisch. Das wird durch ein dreifaches Gleichheitszeichen ausgedrückt: `$a === $b`.

Man kann diese Grundformen auch kombinieren und erhält so folgende Ausdrücke:

- **Kleiner oder gleich:** `$x <= $y` bedeutet: Der Wert der Variablen `$x` ist kleiner oder gleich dem Wert der Variablen `$y`.
- **Größer oder gleich:** `$x >= $y` bedeutet: Der Wert der Variablen `$x` ist größer oder gleich dem Wert der Variablen `$y`.

Schließlich besteht noch die Möglichkeit, die Ungleichheit zweier Werte zu formulieren. Dafür wird das Ausrufezeichen `!` als Negation vor ein Gleichheitszeichen gesetzt:

- **Ungleich:** `$x != $y` bedeutet: Der Wert der Variablen `$x` ist ungleich dem Wert der Variablen `$y`.

In Abhängigkeit vom Wert der verknüpften Variablen ist ein so formulierter Ausdruck entweder wahr oder falsch.

Für $a = 1 und $b = 2 gilt:		
Ausdruck	**Wert**	**Erläuterung**
`$a < $b`	`TRUE`	Der Ausdruck ist wahr, weil 1 kleiner als 2 ist.
`$a > $b`	`FALSE`	Der Ausdruck ist falsch, weil 1 nicht größer als 2 ist.
`$a == $b` `$a === $b`	`FALSE`	Beide Ausdrücke sind falsch, da 1 und 2 nicht gleich und schon gar nicht identisch sind.
`$a != $b`	`TRUE`	Der Ausdruck ist wahr, weil 1 ungleich 2 ist.
`$a <= $b`	`TRUE`	Der Ausdruck ist wahr, weil 1 kleiner oder gleich 2 ist.
`$a >= $b`	`FALSE`	Der Ausdruck ist falsch, weil 1 weder kleiner noch größer als 2 ist.

Tabelle 3.1: Der Wahrheitsgehalt einer Aussage hängt vom Wert der verschiedenen Variablen ab.

In PHP lassen sich diese Ausdrücke auswerten und je nach Ergebnis (TRUE oder FALSE) lässt sich der weitere Verlauf über Wenn/dann-Strukturen des Scripts steuern, etwa: Wenn ein bestimmter Ausdruck TRUE ist, dann tue dies, andernfalls tue jenes.

3.1.2 Wahrheitswerte von Ausdrücken

In PHP besitzen nicht nur explizit formulierte Vergleiche wie `$a < $b` einen Wahrheitswert, sondern jeder Ausdruck ist für PHP entweder TRUE oder FALSE. Glücklicherweise müssen Sie sich da nicht viel merken, denn ein Ausdruck ist (fast) immer TRUE – es sei denn, es tritt einer der folgenden Sonderfälle ein:

- **NULL:** Der Ausdruck ist vom Typ NULL – dann ist er immer FALSE.
- **Der Wert 0:** Ein Ausdruck mit dem Wert 0 ist ebenfalls FALSE. Zuweisungen wie `$a = 0` (Integer), `$a = 0.0` (Float), `$a = "0"` oder Berechnungen, die 0 ergeben, wie etwa `7+3-10` sind also FALSE.
- **Leer:** Eine leere Variable wie `$a = ""` ist FALSE. Obacht! Das sind zwei direkt aufeinanderfolgende Anführungszeichen, ohne Leerzeichen dazwischen. Ein String mit einem Leerzeichen wie `$a = " "` scheint für uns zwar leer zu sein, für PHP ist er das aber nicht. Ein Leerzeichen ist für PHP nur ein Zeichen unter vielen und wird genauso behandelt wie a, b, c und so weiter. Die Zeichenkette ist also nicht leer, ihr Wert ist damit TRUE.

Diese Eigenschaften erleichtern in der Praxis nicht nur manche Abfragen, um im Programmverlauf Entscheidungen zu treffen, sondern sind leider auch eine häufige Fehlerquelle bei der Arbeit mit PHP. Dann nämlich, wenn statt einer Abfrage auf Gleichheit (==) eine Zuweisung (=) notiert wird. Dazu gleich mehr.

3.1.3 Verknüpfungen

Die einfachen Vergleiche erlauben auch nur einfache Abfragen. Mitunter möchte man aber etwas komplexere Bedingungen formulieren und die Ausführung eines Programmcodes von mehr als nur einem Ausdruck abhängig machen, etwa: »Wenn der Besucher Peter heißt und er mindestens ein Produkt im Einkaufskorb hat, dann …«

In diesem Fall müssen Sie die unterschiedlichen Ausdrücke mit einem logischen `and` (»und«) bzw. einem logischen `or` (»oder«) verknüpfen. Der so gebildete neue Ausdruck erzeugt in Abhängigkeit von den Wahrheitswerten der verknüpften Ausdrücke einen neuen Wahrheitswert.

Eine `and`-Verknüpfung ist dann (und nur dann!) wahr, wenn alle verknüpften Ausdrücke wahr sind. Eine `and`-Verknüpfung lässt sich mit dem doppelten Ampersand `&&` abkürzen:

```
<?PHP
  $aussage = ($user == "Peter") and ($produkte > 0);
  $aussage = ($user == "Peter") && ($produkte > 0);
?>
```

Eine or-Verknüpfung ist dann wahr, wenn mindestens einer der beiden verknüpften Ausdrücke wahr ist. Daraus folgt, dass die or-Verknüpfung auch dann wahr ist, wenn beide verknüpften Aussagen wahr sind. Eine or-Verknüpfung lässt sich mit dem doppelten Pipesymbol oder Verkettungszeichen || abkürzen:

```
<?PHP
  $aussage = ($a < $b) or ($c > $d);
  $aussage = ($a < $b) || ($c > $d);
?>
```

Pipesymbol

Das Pipesymbol ist ein senkrechter Strich, mitunter auch ein senkrechter, in der Mitte durchbrochener Strich. Unter Mac OS X geben Sie das Zeichen mit der Tastenkombination Alt+7 ein, unter Windows drücken Sie AltGr+<.

Eine xor-Verknüpfung ist eine Verschärfung der or-Verknüpfung. Sie ist dann (und nur dann!) wahr, wenn *entweder* der eine *oder* der andere verknüpfte Ausdruck wahr ist. Wenn beide wahr sind, ist eine xor-Verknüpfung falsch, wenn beide verknüpften Aussagen falsch sind, natürlich auch. Für xor gibt es keine Abkürzung:

```
<?PHP
  $aussagen = ($a < $b) xor ($c > $d);
?>
```

Schließlich können Sie jeden Wahrheitswert durch ein vorangestelltes ! negieren. Ein negierter Ausdruck ist dann wahr, wenn der Ausdruck falsch ist, und umgekehrt:

```
<?PHP
  $aussage = !($a < $b);
?>
```

Für $a = 1, $b = 2, $c = 3 und $d = 4 gilt:		
Ausdruck	**Wert**	**Erläuterung**
($a == $b) and ($c < $d)	FALSE	Die Aussage $a == $b ist FALSE, $c < $d ist TRUE, die and-Verknüpfung ist also FALSE.
($a < $b) or ($c > $d)	TRUE	Die Aussage $a < $b ist TRUE, die Aussage $c > $d ist FALSE, die or-Verknüpfung also TRUE.
($a < $b) xor ($c < $d)	FALSE	Die Aussage $a < $b ist TRUE, die Aussage $c < $d ist ebenfalls TRUE, die xor-Verknüpfung also FALSE.
!(($a == $b) and ($c < $d))	TRUE	Die and-Verknüpfung ist FALSE, ihre Negation also TRUE.

Tabelle 3.2: Mit logischen Verknüpfungen lassen sich komplexe Abfragen erzeugen.

3.2 Wenn/dann/andernfalls

Mit der Möglichkeit, Verhältnisse zwischen zwei Variablen bzw. Aussagen zu beschreiben, sind wir nun in der Lage, in PHP Entscheidungen zu treffen.

3.2.1 Wenn/dann

Die klassische »Wenn/dann«-Verknüpfung wird in PHP mit einer `if`-Anweisung realisiert. Der umgangssprachliche Satz »Wenn eine bestimmte Bedingung zutrifft, dann tue dies«, ließe sich in PHP abstrakt so formulieren:

```
if (Bedingung) Anweisung;
```

Jetzt sind wir etwa in der Lage, ein kleines Script zu schreiben, das Entscheidungen treffen kann. Als Test nehmen wir zwei Variablen mit `GET` entgegen und vergleichen sie in drei aufeinanderfolgenden `if`-Abfragen auf ihre mögliche Beziehung. Falls ein Vergleich zutrifft, wird ein entsprechender Text ausgegeben.

```
<?PHP
  $a = $_GET["a"];
  $b = $_GET["b"];
  if ($a < $b) echo "A ist kleiner als B";
  if ($a > $b) echo "A ist größer als B";
  if ($a == $b) echo "A ist gleich B";
?>
```

Listing 3.1: Mit einer `if`-Abfrage lassen sich Aktionen in Abhängigkeit von bestimmten Bedingungen ausführen.

Das Listing 3.1 funktioniert wie erwartet, ist aber weder elegant noch effizient. Denn die drei aufeinanderfolgenden `if`-Abfragen werden immer ausgeführt – auch wenn dies überhaupt nicht notwendig ist.

Stellt das Script etwa gleich in der ersten Zeile fest, dass `$a` kleiner als `$b` ist, kann es sich die Überprüfung der beiden folgenden Zeilen sparen, weil sie garantiert nicht mehr zutreffen können.

Neben der Wenn/dann-Entscheidung benötigen wir also noch die Möglichkeit, ein »andernfalls« in PHP formulieren zu können.

3.2.2 Andernfalls

Für diesen Fall gibt es die `else`- Anweisung. Das englische Wort bedeutet soviel wie »sonst, andernfalls«. In Kombination mit `if` sind damit Konstruktionen der Art »Wenn die Bedingung zutrifft, tue jenes, andernfalls unternehme dieses« möglich.

Trifft die Bedingung in der `if`-Abfrage zu, wird die damit verbundene Anweisung ausgeführt und der `else`-Teil ignoriert. Dieser Teil wird erst berücksichtigt, wenn die `if`-Abfrage den Wert `FALSE` ergeben hat.

In ihrer Grundstruktur sieht eine solche Konstruktion in PHP so aus:

```
if (Bedingung) Anweisung;
else Anweisung;
```

Damit können wir unser kleines Script zum Beispiel folgendermaßen umformulieren (Listing 3.2):

```
<?PHP
  $a = $_GET["a"];
  $b = $_GET["b"];
  if ($a < $b) echo "A ist kleiner als B.";
  else echo "A ist größer oder gleich B.";
?>
```

Listing 3.2: Mit dem Schlüsselwort `else` ***können wir Entscheidungen flexibel formulieren.***

Das ist schon mal nicht schlecht, aber auch nicht richtig gut – wir möchten ja die drei möglichen Fälle (kleiner, größer, gleich) sauber trennen.

Hier hilft uns `elseif` weiter, mit der sich mehrere Bedingungen nacheinander überprüfen lassen. Sobald eine Bedingung erfüllt ist, werden die restlichen Abfragen übersprungen. Am Schluss steht schließlich eine `else`-Anweisung, die ausgeführt wird, wenn keine der Bedingungen erfüllt wurde:

```
if (Bedingung 1) Anweisung;
elseif (Bedingung 2) Anweisung;
elseif (Bedingung 3) Anweisung;
elseif (Bedingung 4) Anweisung;     // und so weiter …
else Anweisung;
```

Wenden wir dieses Wissen nun auf unser Beispielscript an, erhalten wir folgendes Ergebnis (Listing 3.3):

```
<?PHP
  $a = $_GET["a"];
  $b = $_GET["b"];
  if ($a < $b) echo "A ist kleiner als B.";
  elseif ($a > $b) echo "A ist größer als B.";
  else echo "A ist gleich B.";
?>
```

Listing 3.3: Kombinieren wir `if`, `elseif` ***und*** `else`***, lassen sich auch komplexere Entscheidungen treffen.***

Gehen wir das Listing 3.3 einmal schrittweise durch:

1. Zuerst werden die beiden Variablen `$a` und `$b` als `GET`-Parameter eingelesen.
2. Nun überprüft das Script, ob `$a` kleiner als `$b` ist. Falls dies der Fall ist, gibt es eine passende Meldung aus und ignoriert die folgenden `elseif`- und die `else`-Abfrage.
3. Erst wenn die erste Abfrage als Ergebnis ein `FALSE` liefert, geht das Script zur nächsten Abfrage über und überprüft nun, ob `$a` größer als `$b` ist. Falls diese Abfrage `TRUE` liefert, wird der angegebene Text ausgegeben, die folgende `else`-Anweisung wird ignoriert.
4. Wenn beide Abfragen falsch sind, dann müssen `$a` und `$b` gleich groß sein. Wir können uns also eine weitere Abfrage sparen und den Text in der `else`-Anweisung ausgeben lassen.

3.2.3 Mehrere Anweisungen

In ihrer Grundform werden die `if/elseif/else`-Abfragen eher selten benutzt. Denn häufig ist es so, dass nicht nur eine, sondern gleich mehrere Anweisungen ausgeführt werden sollen, sobald eine Bedingung erfüllt ist.

In diesem Fall werden die verschiedenen Anweisungen in einer Mengenklammer zusammengefasst, die einzelnen Anweisungen werden wie gewohnt mit einem Semikolon voneinander getrennt.

Damit der Code lesbar bleibt, werden die Anweisungen ein wenig eingerückt, die abschließende Mengenklammer steht dann wieder auf der gleichen Höhe wie die `if/elseif/else`-Anweisung selbst.

In abstrakter Form sieht eine solche Konstruktion dann so aus:

```
if (Bedingung) {
  Anweisung 1;
  Anweisung 2;
  Anweisung 3;
} elseif (Bedingung 2) {    // kein ; zwischen } und elseif
  Anweisung 1;
  Anweisung 2;
  Anweisung 3;
} else {                    // kein ; zwischen } und else
  Anweisung 1;
  Anweisung 2;
  Anweisung 3;
}                          // hier könnte ein ; stehen, muss aber nicht
```

Kein Semikolon

Wie Sie sehen, folgt nach den öffnenden und schließenden Mengenklammern kein Semikolon. Sie können nach der öffnenden Klammer { ein Semikolon setzen (dann geht PHP einfach von einer leeren Anweisung innerhalb der `if`-Anweisung aus), was allerdings sehr unüblich ist. Nach der letzten schließenden Mengenklammer } können Sie ebenfalls eines setzen, müssen es aber auch nicht tun. Folgt der schließenden Klammer allerdings ein `elseif` oder `else`, dann ist ein Semikolon ein Fehler. Beide Anweisungen können nur in direkter Verbindung mit einer `if`-Abfrage benutzt werden. Ein Semikolon stünde aber für eine leere Anweisung, `elseif` und `else` würden also nicht mehr direkt zur `if`-Abfrage gehören, weshalb der PHP-Interpreter damit nichts anzufangen weiß und sich beschwert.

Das folgende kleine Beispiel (Listing 3.4) gibt nur dann den Text innerhalb des `if`-Blocks aus, wenn die Variable `$a` kleiner als die Variable `$b` ist. Andernfalls passiert überhaupt nichts:

```
<?PHP
  $a = $_GET["a"];
  $b = $_GET["b"];
```

```
  if ($a < $b) {
    echo "A ist kleiner als B. ";
    $ergebnis = $b-$a;
    echo "B minus A ist $ergebnis";
  }
?>
```

Listing 3.4: Nach einer* `if`*-Anweisung lassen sich mehrere Anweisungen ausführen, indem man sie in Mengenklammern setzt.

3.2.4 Alternative Schreibweise

Statt mit Mengenklammern können Sie einen Anweisungsblock auch mit einem Doppelpunkt und `endif` umschließen, auch hier sind `if`/`elseif`-Kombinationen möglich:

```
if (bedingung):
  Anweisung 1;
  Anweisung 2;
elseif (bedingung2):
  Anweisung3;
  Anweisung4;
endif;
```

Diese Art der Notation wird Ihnen gelegentlich unterkommen, Sie selbst sollten sie eher vermeiden, da sie in der Regel schwerer lesbar ist als die Standardversion mit Mengenklammern.

3.2.5 Beispiel: Variablen auf numerischen Inhalt überprüfen

Das Listing 3.3 führt zwar keine überflüssigen Abfragen mehr durch, aber einen Haken hat die Sache doch noch: Es erkennt nicht, wenn keine Werte für `$a` und `$b` übergeben wurden, und führt anschließend eher unsinnige Vergleiche aus. Das ist in diesem Beispiel zwar noch ein harmloser Fehler ohne große Konsequenzen, kann aber in der Praxis, etwa bei der Auswertung eines Formulars, fatale Folgen haben.

Hier helfen uns die in Kapitel 2 vorgestellten Funktionen zur Überprüfung von Variablen. Möchten wir etwa sicherstellen, dass unserem Script numerische Werte übergeben werden, setzen wir dafür `is_numeric()`. In den Klammern wird der Funktion die Variable übergeben. Handelt es sich dabei um einen numerischen Wert, erhalten wir den Wert `TRUE`, andernfalls ist der Wert `FALSE`. So lässt sich diese Funktion sehr schön in eine `if`-Abfrage integrieren:

```
<?PHP
  if (is_numeric($a)) echo "A hat den Wert $a";
  else echo "Geben Sie für A einen numerischen Wert ein";
?>
```

Unser Beispielscript Listing 3.3 kann nun so umgebaut werden, dass es nur ausgeführt wird, wenn für `$a` und `$b` numerische Werte übergeben wurden. Andernfalls wird ein kurzer Hinweis angezeigt.

Die `if`-Abfragen des bisherigen Scripts rutschen dabei komplett in den Anweisungsteil einer weiteren `if`-Abfrage, die dann (und nur dann) ausgeführt wird, wenn sowohl `$a` als auch `$b` einen numerischen Wert besitzen, die Funktionen `is_numeric($a)` und `is_numeric($b)` also beide Male den Wert `TRUE` liefern. Diese Abfrage lässt sich durch eine `and`- bzw. `&&`-Verknüpfung realisieren (Listing 3.5):

```
<?PHP
  $a = $_GET["a"];
  $b = $_GET["b"];
  if (is_numeric($a) && is_numeric($b)) {
    if ($a < $b) echo "A ist kleiner als B";
    elseif ($a > $b) echo "A ist größer als B";
    else echo "A ist gleich B";
  } else echo "Geben Sie bitte numerische Werte für A und B ein.";
?>
```

Listing 3.5: Die* `if`*-Abfragen lassen sich beliebig verschachteln.

3.2.6 Beispiel: Parameterprüfung mit `isset()` und `empty()`

Im vorherigen Kapitel haben Sie bereits die beiden Variablenfunktionen `isset()` und `empty()` kennengelernt. Mit unserem neu erworbenen Wissen über Kontrollstrukturen und Abfragen können wir uns den Unterschied zwischen diesen beiden Funktionen noch einmal verdeutlichen (und bei der Entwicklung des kleinen Scripts auch gleich lernen, wie wir von einer einfachen Idee zum fertigen Script gelangen).

Die Idee: Einem Script kann ein URL-Parameter übergeben werden, der in einer Variablen `$a` gespeichert wird. Das Script soll nun mit `isset()` und `empty()` prüfen, ob überhaupt ein Parameter übergeben wurde, und wenn ja, ob die Variable `$a` leer ist. Falls die Variable einen Inhalt hat, soll dieser angezeigt werden.

```
<?PHP
  $a = $_GET["a"];
  if (isset($a)) echo "\$a existiert.";
  else echo "\$a existiert nicht.";
  echo "<br />";
  if (empty($a)) echo "\$a ist leer.";
  else echo "\$a ist nicht leer und hat den Wert $a.";
?>
```

Listing 3.6: Das kleine Script überprüft die Existenz eines* `GET`*-Parameters und seinen Inhalt.

Listing 3.6 liest zuerst einen eventuell übergebenen Parameter `a` ein. Anschließend wird mit `isset($a)` geprüft, ob die Variable tatsächlich existiert, also ob überhaupt ein Parameter vorliegt. Falls dies der Fall ist, wird eine entsprechende Meldung ausgegeben. Andernfalls (`else`) wird die Nicht-Existenz von `$a` festgestellt. Anschließend fügen wir einen Zeilenumbruch ein und überprüfen mit `empty()`, ob die Variable leer ist. Auch hier wird eine entsprechende Meldung ausgegeben.

Speichern Sie das kleine Script als `empty.php` und rufen Sie es anschließend ohne und mit verschiedenen Parametern auf, wobei Sie ruhig fehlerhafte bzw. unvollständige Aufrufe ausprobieren sollten – schließlich können Sie nie wissen, was ein Anwender in die Adresszeile seines Browsers eintippt. Etwa so:

- **Ohne Parameter:** Rufen Sie das Script mit `empty.php` auf.
- **Unvollständig:** Übergeben Sie einen Parameter, weisen Sie ihm aber keinen Wert zu, z.B.: `empty.php?a` oder `empty.php?a=`
- **Null:** Weisen Sie dem Parameter den Wert `0` zu: `empty.php?a=0`.

- **Beliebiger Wert:** Übergeben Sie einen beliebigen Wert, ganz gleich ob Zahl oder Zeichenkette, etwa `empty.php?a=10` oder `empty.php?a=test`.

Sie werden feststellen, dass das Script in Listing 3.6 auch dann die Existenz der Variablen `$a` meldet, wenn diese ohne Wert übergeben wurde. Nur wenn Sie das Script ohne Parameter aufrufen, wird die Meldung `$a existiert nicht` ausgegeben. Übergeben Sie eine `0`, wird gemeldet, dass die Variable `$a` existiert und dass die Variable leer ist. Wir erinnern uns: `empty()` ist auch dann `TRUE`, wenn eine Variable den Wert `0` enthält.

Das Script funktioniert, hat aber noch einen Schönheitsfehler: Es überprüft selbst dann, ob eine Variable leer ist, wenn bereits feststeht, dass diese gar nicht existiert. Das führt zu der redundanten Ausgabe von `$a existiert nicht. $a ist leer.`

Um das zu vermeiden, wenden wir einen kleinen Trick an und fragen zuerst ab, ob die Variable *nicht* existiert. Erst wenn diese Abfrage `FALSE` ergibt – die Variable also existiert –, führen wir eine Prüfung ihres Inhalts durch. Dazu setzen wir den Negationsoperator `!` ein:

```
<?PHP
  $a = $_GET["a"];
  if (!isset($a)) echo "\$a existiert nicht.";
  else {
    echo "\$a existiert.";
    echo "<br />";
    if (empty($a)) echo "\$a ist leer.";
    else echo "\$a ist nicht leer und hat den Wert $a.";
  }
?>
```

Listing 3.7: Nur wenn die Variable `$a` tatsächlich existiert, wird überprüft, ob sie leer ist.

Speichern Sie Listing 3.7 erneut als `empty.php` und rufen Sie es wieder mit verschiedenen Parametern auf (Abbildung 3.1).

Abbildung 3.1: Obwohl die Variable `$a` existiert und ihr mit `GET` ein Wert zugewiesen wurde, gilt sie für PHP als leer, da sie den Wert `0` besitzt.

3.2.7 Beispiel: Gleich und identisch

Mit einem kleinen Script (Listing 3.8) lässt sich auch der Unterschied zwischen Gleichheit und Identität zweier Werte besser verstehen.

```
<?PHP
  $a = 10;
  $b = 10.0;
  if ($a === $b) echo "A und B sind identisch.";
  elseif ($a == $b) echo "A und B sind gleich, aber nicht identisch.";
  else echo "A und B sind ungleich.";
?>
```

Listing 3.8: Gleich und identisch sind für PHP verschiedene Zustände.

Im ersten Schritt wird die Identität der beiden Variablen getestet, gegebenenfalls eine passende Meldung angezeigt und das Script beendet. Falls die Variablen nicht identisch sind, werden sie auf Gleichheit geprüft. Falls die ersten beiden Abfragen `FALSE` waren, müssen beide Variablen unterschiedliche Werte haben.

Probieren Sie das Listing 3.8 auch mit einer Zeichenkette wie `"10 Eier"` oder einer Fließkommazahl wie `10.1` aus.

3.3 Entscheidungen mit Switch

Solange Sie nur zwei Werte vergleichen, werden Sie mit `if`, `elseif` und `else` keine Probleme bekommen. Sobald es aber darum geht, mehr als nur zwei oder drei Möglichkeiten zu beachten, tendieren `if`-Abfragen zur Unübersichtlichkeit.

3.3.1 Alles auf einen Streich

Hier steht mit `switch` eine Alternative bereit, mit der sich gewissermaßen beliebig viele `if`-Abfragen zusammenfassen lassen.

Die `switch`-Anweisung wertet einen Ausdruck aus und vergleicht anschließend beliebig viele Varianten mit dem Ergebnis. Sollte eine Variante (`case`) zutreffen, wird der dazugehörige Programmcode ausgeführt und die Anweisung schließlich über das Kommando `break` verlassen. Ein frei definierbarer `default`-Block enthält überdies Anweisungen, die ausgeführt werden, wenn keine der Vorgaben zutrifft. Die Struktur der `switch`-Anweisung hat folgenden Aufbau (achten Sie darauf, dass die `case`- und `default`-Zeilen mit einem Doppelpunkt `:` und nicht mit einem Semikolon `;` abgeschlossen werden):

```
switch (Ausdruck) {
  case Ergebnis1:             // Doppelpunkt, kein Semikolon!
    Anweisung 1;
    Anweisung 2;
    break;
  case Ergebnis2:
    Anweisung 1;
    Anweisung 2;
    break;
```

```
// beliebig viele weitere case-Einträge
  default:                      // Doppelpunkt, kein Semikolon
    Anweisung_default;
}
```

Wenn nur wenige, kurze Anweisungen abgearbeitet werden, können Sie diese auch in einer Zeile notieren. Sie müssen nur darauf achten, dass die Anweisungen mit einem Semikolon getrennt werden. Also zum Beispiel so:

```
switch (Ausdruck) {
  case Ergebnis1: Anweisung; break;
  case Ergebnis2: Anweisung; break;
// beliebig viele weitere case-Einträge
  default: Anweisung_default;
}
```

3.3.2 Beispiel: Aktuellen Wochentag anzeigen

Als Beispiel soll uns PHP den aktuellen Wochentag in Form von `Heute ist Montag` anzeigen. Dazu benutzen wir die Datumsfunktion `date()`. Mit der werden wir uns später noch ausführlicher beschäftigen, für unsere Zwecke reicht die Übergabe des Parameters `"l"`, mit dem uns `date()` den Namen des aktuellen Tages nennt (der Parameter ist ein kleines L, nicht die Zahl 1):

```
<?= "Heute ist ",date("l"); ?>
```

Der Haken an der Sache: Der Tag wird mit der englischen Bezeichnung angezeigt, die Codezeile erzeugt also einen Text wie `Heute ist Monday`. Das ist natürlich unschön, aber hier hilft uns `switch` weiter (Listing 3.9):

```
<?PHP
  $tag = date("l");
  switch ($tag) {
    case "Monday": $tag = "Montag"; break;
    case "Tuesday": $tag = "Dienstag"; break;
    case "Wednesday": $tag = "Mittwoch"; break;
    case "Thursday": $tag = "Donnerstag"; break;
    case "Friday": $tag = "Freitag"; break;
    case "Saturday": $tag = "Samstag"; break;
    default: $tag = "Sonntag";
  }
  echo "Heute ist $tag";
?>
```

Listing 3.9: Mit `switch` ***lassen sich beliebig viele Fälle untersuchen.***

Zuerst wird der aktuelle Tagesname in der Variablen `$tag` gespeichert. Anschließend überprüfen wir mit `switch`, welchen Inhalt `$tag` besitzt. Sobald ein Fall (`case`) zutrifft, wird der Inhalt der Variablen `$tag` durch den deutschen Begriff ersetzt und die `switch`-Anweisung mit `break` verlassen. Falls die ersten sechs Abfragen von `Monday` bis `Saturday` kein zutreffendes Ergebnis geliefert haben, muss es sich beim Inhalt von `$tag` um `Sunday` halten, die Variable wird also ohne weitere Prüfung auf `Sonntag` gesetzt. Im Anschluss an die `switch`-Anweisung wird schließlich der aktuelle Tag genannt.

3.3.3 Alternative Schreibweise

Auch bei `switch` müssen Sie nicht zwingend Mengenklammern einsetzen, sondern können die Anweisung auch mit Doppelpunkt und `endswitch` notieren:

```
switch (Ausdruck):
  case Ergebnis1:
    // Anweisungen
    break;
  case Ergebnis2:
    // Anweisungen
    break;
// beliebig viele weitere case-Einträge
  default:
    Anweisung_default;
endswitch;
```

3.3.4 Beispiel: Verschiedene Einsprungspunkte

Wie anfangs erwähnt, werden die Anweisungen bei einem zutreffenden `case` so lange ausgeführt, bis ein `break` erreicht oder die gesamte `switch`-Anweisung beendet ist. Dabei werden eventuell folgende `case`-Abfragen ignoriert. Würden wir im Wochentags-Beispiel auf `break` verzichten, würde die Variable `$tag` immer auf `Sonntag` gesetzt werden, was sicherlich nicht erwünscht ist.

Diese Eigenschaft von `switch` lässt sich auch sinnvoll nutzen, indem man über `case` verschiedene Einsprungspunkte in eine Reihe von Anweisungen setzen kann. Machen wir uns das an einem einfachen Beispiel klar.

Einem Script soll ein Buchstabe von `A` bis `E` übergeben werden können, worauf das Script in Abhängigkeit vom Buchstaben alphabetisch weiterzählt und bei einem `E` das Wort `Ende` gefolgt von einem Punkt ausgibt. Übergibt man etwa ein `A`, soll es also den Text `A, B, C, D, Ende.` anzeigen, bei einem `B` lautet die Reihe `B, C, D, Ende.`, ein `C` führt zu `C, D, Ende.` und so weiter.

Mit einer `if`-Abfrage müssten wir immer wieder den gleichen Text schreiben, etwa so (Listing 3.10):

```
<?PHP
  $buchstabe = $_GET["a"];
  if ($buchstabe == "a") echo "A, B, C, D, Ende.";
  elseif ($buchstabe == "b") echo "B, C, D, Ende.";
  elseif ($buchstabe == "c") echo "C, D, Ende.";
  elseif ($buchstabe == "d") echo "D, Ende.";
  else echo "Ende.";
?>
```

Listing 3.10: Eine umständliche Lösung mit `if`***,*** `elseif` ***und*** `else`***.***

Mit `switch` lässt sich eine solche Aufgabenstellung mit weniger Tipparbeit eleganter lösen, indem man die `echo`-Anweisungen für die einzelnen Buchstaben abfragt und auf ein `break` bewusst verzichtet (Listing 3.11):

```
<?PHP
  $buchstabe = $_GET["a"];
  switch ($buchstabe) {
    case "a": echo "A, ";
    case "b": echo "B, ";
    case "c": echo "C, ";
    case "d": echo "D, ";
    default: echo "Ende.";
  }
?>
```

Listing 3.11: Schreiben wir die* `switch`*-Anweisung ohne* `break`*, können wir die verschiedenen Bedingungen als Einsprungspunkt benutzen.

Ruft man das Script Listing 3.11 etwa mit dem Parameter `?a=b` auf, dann ist die erste Abfrage `case "a"` `FALSE` und wird ignoriert, `case "b"` ist allerdings `TRUE` und es wird `B,` ausgegeben. Da kein `break` folgt, geht PHP zu den nächsten Anweisungen über, ohne die Wahrheitswerte der folgenden `case`-Anweisungen zu berücksichtigen, gibt also im Anschluss `C, D, Ende.` aus.

3.4 Fehlerquelle: Zuweisung statt Gleichheit

Es wurde bereits darauf hingewiesen, dass ein vergessenes Gleichheitszeichen in der `if`-Abfrage nach der Gleichheit zweier Werte eine der häufigsten Fehlerquellen in einem PHP-Script ist.

3.4.1 Der Fehler im Detail

Schauen wir uns nun diesen typischen Fehler einmal im Detail an (Listing 3.12):

```
<?PHP
  $a = 5;
  if ($a = 4) echo "A ist gleich 4.";          // FEHLER!
  elseif ($a > 4) echo "A ist größer als 4.";
  else echo "A ist kleiner als 4.";
?>
```

Listing 3.12: Einer der häufigsten Fehler in einer* `if`*-Anweisung ist der Einsatz eines einfachen Gleichheitszeichen bei der Abfrage nach dem Wert einer Variablen.

Welche Ausgaben erzeugt dieses Script? Probieren Sie's aus – es wird immer behauptet, dass die Variable `$a` den Wert 4 besitzt. Und zwar völlig unabhängig davon, welchen Wert `$a` vor der `if`-Anweisung tatsächlich besitzt.

Der Fehler liegt, Sie ahnen es, darin, dass in der `if`-Abfrage eine Zuweisung stattfindet – kein Vergleich. Der Ausdruck `$a = 4` besitzt – wie in Abschnitt 3.1.2 erläutert – aber immer den Wahrheitswert `TRUE`, womit die `if`-Anweisung erfüllt ist und der entsprechende Text ausgegeben wird.

Das geht natürlich auch in die andere Richtung schief, wenn die versehentliche Zuweisung automatisch `FALSE` ist:

```
<?PHP
  $a = 0;
  if ($a = 0) echo "A ist gleich 0.";          // FEHLER!
  elseif ($a > 0) echo "A ist größer als 0.";
  else echo "A ist kleiner als 0.";
?>
```

Listing 3.13: Auch hier wird eine Zuweisung statt eines Vergleichs benutzt, zudem ist eine Variable mit dem Wert `0` ***immer*** `FALSE`**.**

In Listing 3.13 wird der Text `A ist kleiner als 0.` ausgegeben, schließlich ist die Anweisung `$a = 0` immer `FALSE`, PHP geht also zur `elseif`-Abfrage über. Da `$a` aber nicht größer als `0` ist, wird schließlich die `else`-Anweisung ausgeführt und die falsche Aussage erscheint als Ergebnis der fehlerhaft formulierten `if`-Abfrage.

Beide Beispiele geben korrekte Aussagen aus, wenn keine Zuweisung, sondern tatsächlich eine Abfrage nach der Gleichheit stattfindet, also `if ($a == 4)` bzw. `if ($a == 0)`.

3.4.2 Wie man den Fehler (manchmal) vermeiden kann

Dieser Fehler ist deshalb so tückisch und schwer zu finden, weil die `if`-Anweisung formal völlig korrekt ist und PHP daher keine Fehlermeldung ausgibt. In unserem kleinen Beispiel ist der Fehler noch sofort augenfällig, sobald Sie aber etwas komplexere Scripte schreiben, kann die Fehlersuche eine recht frustrierende Angelegenheit werden.

Eine Methode, diesen Tippfehler zu vermeiden, besteht darin, die beiden Werte im Vergleich zu vertauschen, also statt `$a == 5` lieber `5 == $a` zu schreiben. Das liest sich im Quelltext etwas seltsam, ist formal aber völlig korrekt. Der Vorteil: Ein Tippfehler wie `5 = $a` ist ein Syntaxfehler in PHP und wird entsprechend moniert. Allerdings funktioniert das nur beim Vergleich mit festen Werten, sobald Sie die Werte zweier Variablen vergleichen, hilft Ihnen dieser Trick auch nicht weiter. Ein Grund mehr, bei diesen Abfragen besonders sorgfältig zu arbeiten und den Code lieber einmal zu viel als einmal zu wenig zu überprüfen.

3.5 Der ternäre Operator

Eine mitunter ganz praktische, wenn auch selten benutzte Alternative für eine `if/else`-Anweisung ist der ternäre Operator. Der hat folgende abstrakte Form:

```
bedingung ? Wenn TRUE : Wenn FALSE;
```

Wenn `bedingung` `TRUE` ist, wird die erste Anweisung ausgeführt, wenn sie `FALSE` ist, die zweite. Das folgende Beispiel packt eine `if/else`-Anweisung in eine Zeile:

```
<?PHP
  echo $a > $b ? "A ist größer B" : "A ist kleiner oder gleich B";
?>
```

Falls die Variable `$a` größer als `$b` ist, wird der erste Text ausgegeben, andernfalls der zweite.

Diese Notation ist zum Beispiel dann ganz praktisch, wenn Sie einen korrekt deklinierten Satz ausgeben möchten.

Nehmen wir an, ein Besucher Ihrer Webseite kann verschiedene Produkte auswählen. Das Script soll, je nach Anzahl der gewählten Produkte, die Bestellung mit `Sie haben ein Produkt bestellt` bzw. `Sie haben mehrere Produkte bestellt` bestätigen.

Mit `if/else` sähe eine solche Anweisung etwa so aus:

```
<?PHP
  if ($zahl == 1) $text = "Produkt";
  else $text = "Produkte";
  echo "Sie haben $zahl $text bestellt.";
?>
```

Mit dem ternären Operator lässt sich dieser Codeschnipsel deutlich verkürzen:

```
<?PHP
  echo "Sie haben $zahl ",($zahl == 1)?"Produkt":"Produkte"," bestellt.";
?>
```

Falls der Besucher nur ein Produkt bestellt hat, wird in die `echo`-Anweisung das Wort `Produkt`, andernfalls der Plural `Produkte` eingefügt.

3.6 Programmschleifen

Bislang werden die Anweisungen in den Beispielscripten genau ein Mal durchlaufen. Doch ein Computer ist endlos geduldig und entfaltet seine wahre Stärke erst dann, wenn er einen bestimmten Anweisungsblock so lange wiederholt, wie Sie das für richtig halten. Diese Schleifen sind das Kernstück eines jeden Programms und machen es flexibler und leistungsfähiger.

3.6.1 Solange, bis: Schleifen mit `while`

Die einfachste Form einer Schleife in PHP ist die `while`-Konstruktion. »While« ist das englische Wort für »während, solange«, und genau so funktioniert die Schleife auch. Solange eine bestimmte Bedingung erfüllt ist, solange werden die Anweisungen innerhalb der Schleife durchlaufen. In der allgemeinen Form sieht das etwa so aus:

```
while (Bedingung) {
  // Anweisungen
}
```

Alternativ dazu lässt sich `while` auch mit Doppelpunkt und `endwhile` formulieren:

```
while (Bedingung):
  // Anweisungen
endwhile;
```

Solange `Bedingung` den Wert `TRUE` besitzt, werden die Anweisungen in den geschweiften Klammern ausgeführt. Mindestens eine dieser Anweisungen muss also den Wert des Ausdrucks so verändern, dass `Bedingung` den Wert `FALSE` erhält, ansonsten wird die Schleife nie verlassen und das Programm hängt fest.

Vorsicht vor endlosen Schleifen!

Fehlende oder fehlerhaft formulierte Bedingungen produzieren in der Regel endlose Schleifen, da die Bedingung immer TRUE ist. Sie sind eine der häufigsten Ursachen für einen Programmabsturz.

Das klingt jetzt vielleicht etwas abstrakt und trocken, also probieren wir das rasch an einem einfachen Beispiel aus. Lassen wir PHP ein wenig zählen.

```
<?PHP
  $anfang = $_GET["a"];
  $ende = $_GET["b"];
  echo "<p>Die Zahlen von $anfang bis $ende:</p>";
  echo "<p>";
  while ($anfang <= $ende) {
    echo $anfang," ";
    $anfang++;
  }
  echo "</p>";
?>
```

Listing 3.14: Eine einfache* while-*Schleife.

Speichern Sie das Script Listing 3.14 etwa als `counter.php` und rufen Sie es mit zwei Parametern auf, etwa so:

```
http://localhost/counter.php?a=0&b=10
```

Das Script weist zuerst die übergebenen Werte den Variablen `$anfang` und `$ende` zu und gibt anschließend eine Meldung als Überschrift aus. Danach folgt eine `while`-Anweisung, die so lange durchlaufen wird, solange die Bedingung `$anfang <= $ende` den Wert TRUE besitzt. (Das umschließende <p>-Tag ist für das Script selbst nicht notwendig, für korrekten HTML-Code allerdings schon.)

Innerhalb der Schleife wird der Wert von `$anfang`, gefolgt von einem Leerzeichen, ausgegeben. Anschließend wird `$anfang` um 1 erhöht. Nun überprüft PHP erneut, ob die Bedingung TRUE ergibt. Falls dies der Fall ist, wird die Schleife erneut durchlaufen. Das wird so lange wiederholt, bis `$anfang` größer als `$ende` und die Bedingung also FALSE ist.

Strenggenommen müssten wir noch einige Sicherheitsabfragen für den Fall einfügen, dass keine Werte übergeben werden oder dass der Endwert kleiner als der Anfangswert ist. Aber wirklich nötig sind sie in diesem Beispiel nicht, denn die Schleife wird dann (und nur dann) durchlaufen, solange `$anfang` kleiner oder gleich `$ende` ist. In den anderen Fällen passiert gar nichts.

3.6.2 Annehmende Schleife: do … while

Eine Schleife mit `while` ist ein Beispiel für eine sogenannte »abweisende Schleife«, bei der noch vor dem ersten Durchlauf die Bedingung überprüft wird. Abweisend heißt sie, weil sie überhaupt nicht durchlaufen wird, wenn die Bedingung von Anfang an FALSE ist.

Das Gegenstück ist eine »annehmende Schleife«, bei der die Bedingung erst nach den Anweisungen überprüft wird. Eine solche Schleife wird also mindestens einmal, eine abweisende Schleife unter Umständen gar nicht durchlaufen.

Eine annehmende Schleife in PHP wird mit `do … while` konstruiert:

```
do {
  // Beliebige Anweisungen
} while (Ausdruck);
```

Beachten Sie, dass (anders, als bei der `while`-Konstruktion) die letzte Zeile mit einem Semikolon abgeschlossen werden muss. Unser kleines Zählbeispiel sieht mit der `do`-Schleife dann so aus (Listing 3.15):

```
<?PHP
  $anfang = $_GET["a"];
  $ende = $_GET["b"];
  echo "<p>Die Zahlen von $anfang bis $ende:</p>";
  echo "<p>";
  do{
    echo $anfang," ";
    $anfang++;
  } while ($anfang <= $ende);
  echo "</p>";
?>
```

Listing 3.15: Eine annehmende Schleife wird mindestens einmal durchlaufen.

3.6.3 Schleifen mit Zähler (von/bis): `for`

Neben der `while`- kennt PHP noch die `for`-Schleife, die ähnlich wie `while` arbeitet und ihre Stärken für alle irgendwie abzählbaren Bedingungen besitzt. Im Unterschied zu einer `while`-Schleife sorgen wir nicht innerhalb der Schleife dafür, dass die Bedingung der Schleife einmal `FALSE` erreicht, sondern wir formulieren die notwendige Steueranweisung bereits zu Beginn der Schleife. Die allgemeine Syntax einer `for`-Schleife sieht so aus:

```
for (Initialsierung; Bedingung; Veränderung) {
  // Beliebige Anweisungen
}
```

Wie gewohnt, können Sie also mehrere Anweisungen in Mengenklammern zusammenfassen. Besteht die Schleife allerdings nur aus einer Anweisung, lässt sich diese auch direkt nach der `for`-Anweisung angeben:

```
for (Initialsierung; Bedingung; Veränderung) Anweisung;
```

Kernstück der `for`-Schleife sind die drei durch ein Semikolon getrennten Anweisungen in Klammern.

Wie schon bei den vorherigen Kontrollstrukturen gibt es auch bei `for` eine alternative Schreibweise, die gelegentlich benutzt wird:

```
for (Initialsierung; Bedingung; Veränderung):
  // Beliebige Anweisungen
endfor;
```

- **Initialisierung:** Diese Anweisung wird genau einmal beim Start der Schleife ausgeführt. Hier wird der Startpunkt der Schleife definiert, etwa `$a = 0`.
- **Bedingung:** Solange die hier genannte Bedingung `TRUE` ist, wird die Schleife durchlaufen, etwa `$a <= 10`.
- **Veränderung:** Schließlich sorgt die kontinuierliche Veränderung einer Variablen dafür, dass die Bedingung zu einem definierten Zeitpunkt `FALSE` und die Schleife verlassen wird, etwa `$a++`.

Um die Zahlen von 1 bis 10 (gefolgt von einem Leerzeichen) auszugeben, benötigt man also zum Beispiel diese Schleife:

```
<?PHP
  for ($a = 1;$a <= 10; $a++) echo $a," ";
?>
```

Das kleine `while`-Beispiel ließe sich also in einer `for`-Schleife folgendermaßen notieren (Listing 3.16):

```
<?PHP
  $anfang = $_GET["a"];
  $ende = $_GET["b"];
  echo "<p>Die Zahlen von $anfang bis $ende:</p>";
  echo "<p>";
  for ($wert = $anfang; $wert <= $ende; $wert++) echo $wert," ";
  echo "</p>";
?>
```

Listing 3.16: Unsere Zählschleife, diesmal mit `for`.

3.6.4 Beispiel: Eine ANSI-Tabelle

Ein typischer Einsatz für die `for`-Schleife ist das Abzählen von Werten. Will man etwa die ANSI-Codes von 32 bis 255 ausgeben, so benötigt man dafür gerade mal eine Zeile PHP-Code.

HTML- und numerische Entities

In HTML können Zeichen durch sogenannte Entities dargestellt werden. Eine Entity wird mit einem Ampersand & eingeleitet und mit einem Semikolon ; beendet. Das auszugebende Zeichen wird entweder mit seinem ISO-Namen angegeben (z.B. `auml` für »a-Umlaut«) oder mit seinem numerischen ANSI-Code, etwa `#228`. Ein ä lässt sich in HTML also als `&auml;` oder als `&#228;` darstellen.

Man lässt in einer Schleife eine Variable von 32 bis 255 hochzählen und setzt diese Variable in einer `echo`-Anweisung als Wert für die numerische Entity ein:

```
<html>
  <head>
    <title>ANSI-Codes 32 bis 255</title>
    <style type="text/css">
      p {font-family : Courier;}
    </style>
  </head>
  <body>
    <p>Zeichensatz der ANSI-Codes von 32 bis 255:</p>
    <p>
      <?PHP for ($x = 32; $x <= 255; $x++) echo "&#$x; "; ?>
    </p>
  </body>
</html>
```

Listing 3.17: Für die Ausgabe einer ANSI-Tabelle genügt im Kern eine einzige Zeile in PHP, der Rest ist HTML zur Formatierung der Anzeige.

Speichern Sie Listing 3.17 als `ansi.php` und rufen Sie es im Browser auf. Sie erhalten eine Tabelle wie in Abbildung 3.2.

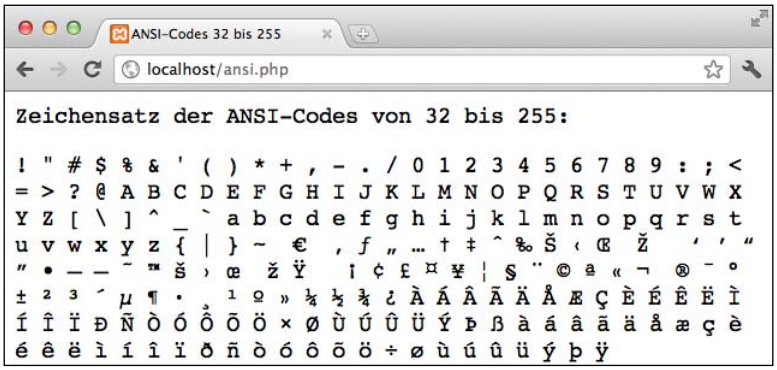


Abbildung 3.2: Eine einzige Zeile in PHP genügt, um sich den Zeichensatz der ANSI-Codes ausgeben zu lassen. (Einige der Codes sind leer oder mit Steuerzeichen belegt und erzeugen keine Ausgabe.)

3.7 Schleifen schachteln

Schleifen lassen sich beliebig verschachteln, also innerhalb einer Schleife lässt sich eine weitere Schleife ausführen. So sind komplexe Strukturen möglich, die allerdings nicht immer auf Anhieb verständlich sind und dazu tendieren, etwas unübersichtlich zu werden. Gehen wir das Verfahren also einfach an einem Beispiel durch.

3.7.1 Das kleine Einmaleins

Aufgabe ist es, das kleine Einmaleins als Tabelle mit PHP auszugeben. Dazu werden zwei Schleifen benötigt: eine für die Reihen, die von 1 bis 10 zählt, und eine entsprechende Schleife für die Spalten.

Das Grundgerüst sieht dann etwa so aus (Listing 3.18):

```
<?PHP
  for ($reihe = 1; $reihe <= 10; $reihe++) {
    for ($spalte = 1; $spalte <= 10; $spalte++) {
      echo $reihe * $spalte," ";
    }
    echo "<br />";
  }
?>
```

Listing 3.18: Das Grundgerüst für die Ausgabe des kleinen Einmaleins.

Die erste Schleife benutzt die Variable `$reihe` als Zähler von 1 bis 10. Die erste Anweisung innerhalb der Schleife ist eine weitere Schleife, in der die Variable `$spalte` ebenfalls von 1 bis 10 gezählt wird. Was passiert nun, wenn wir das Script aufrufen? Spielen wir den Ablauf einmal im Kopf durch.

1. Zu Beginn wird der Variablen `$reihe` der Wert 1 zugewiesen. Da die Bedingung `$reihe <= 10` damit (noch) `TRUE` ist, werden die Anweisungen in der Schleife ausgeführt.
2. Nun wird die zweite Schleife ausgeführt, in der die Variable `$spalte` mit 1 initialisiert wird. Die Bedingung der Schleife ist `TRUE`, sie wird also ausgeführt.
3. Innerhalb der zweiten Schleife wird nun das Ergebnis der Multiplikation von `$reihe` und `$spalte`, gefolgt von einem Leerzeichen, ausgegeben. Da beide Variablen aktuell den Wert 1 besitzen, erhalten wir also eine 1, gefolgt von einem Leerzeichen.
4. Nun wird die Variable `$spalte` um 1 erhöht, hat also nun den Wert 2. Die Bedingung der Schleife ist immer noch `TRUE`, sie wird also erneut durchlaufen. Da `$reihe` nach wie vor den Wert 1 besitzt, erhalten wir jetzt die Ausgabe 2 (ebenfalls gefolgt von einem Leerzeichen), die direkt an die vorherige Ausgabe angefügt wird.
5. Das Ganze wiederholt sich so lange, bis `$spalte` den Wert 11 erreicht. Als Ergebnis produziert die zweite Schleife also folgende Ausgabe: `1 2 3 4 5 6 7 8 9 10`
6. Nun wird die zweite Schleife verlassen und die nächste Anweisung der ersten Schleife ausgeführt, also ein `<br />` der Zahlenreihe angefügt.
7. Damit ist auch die erste Schleife abgeschlossen, `$reihe` wird um 1 erhöht und hat nun den Wert 2.
8. Da die Bedingung der ersten Schleife damit immer noch `TRUE` ist, wird sie erneut durchlaufen und die zweite Schleife produziert die nächste Zeile: `2 4 6 8 10 12 14 16 18 20`
9. Das Script gibt als nächste Anweisung der ersten Schleife erneut ein `<br />` aus.
10. Das Ganze wiederholt sich so lange, bis `$reihe` den Wert 11 erreicht und auch die erste Schleife verlassen wird.

Damit ist der PHP-Code bereits fertig und wir können einen kleinen Testlauf starten (Abbildung 3.3).

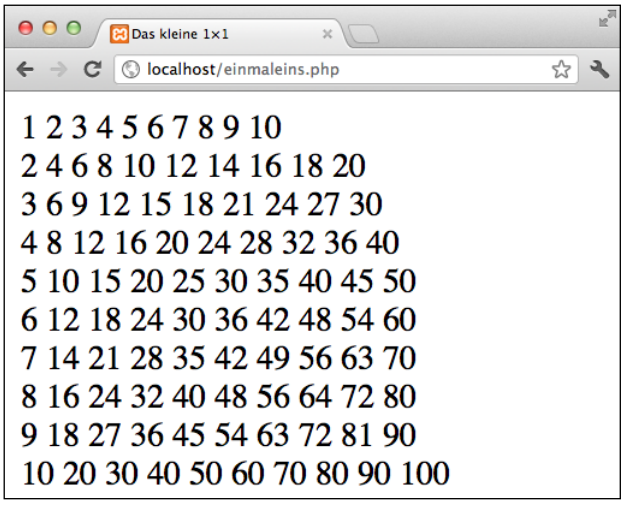


Abbildung 3.3: Die Berechnung des kleinen Einmaleins ist korrekt – aber die Darstellung lässt doch etwas zu wünschen übrig.

3.7.2 Formatierung der Ausgabe

Das Listing 3.18 arbeitet wie gewünscht, doch die Ausgabe ist noch ein wenig schmucklos. Zum einen sollte eine Tabelle auch als Tabelle dargestellt werden, zum anderen wäre es schön, wenn die Einträge in der ersten Spalte und in der ersten Reihe gefettet würden.

Die Darstellung als Tabelle ist einfach. Statt eines `<br />` zur Trennung der einzelnen Zeilen benutzen wir Tabellenzeilen mit `<tr>`, in denen die Werte als Tabellenzellen in `<td>` eingetragen werden.

Um die Einträge in der ersten Spalte/ Reihe hervorzuheben, fügen wir eine `if`-Abfrage ein, mit der die Platzierung des notwendigen `<strong>`-Tags gesteuert wird. Eine Zahl steht in der ersten Reihe bzw. Spalte, wenn `$reihe` bzw. `$spalte` den Wert 1 enthält. Oder in PHP formuliert:

```
$reihe == 1 || $spalte == 1
```

Wenn dieser Ausdruck `TRUE` ist, dann soll die Zahl gefettet werden.

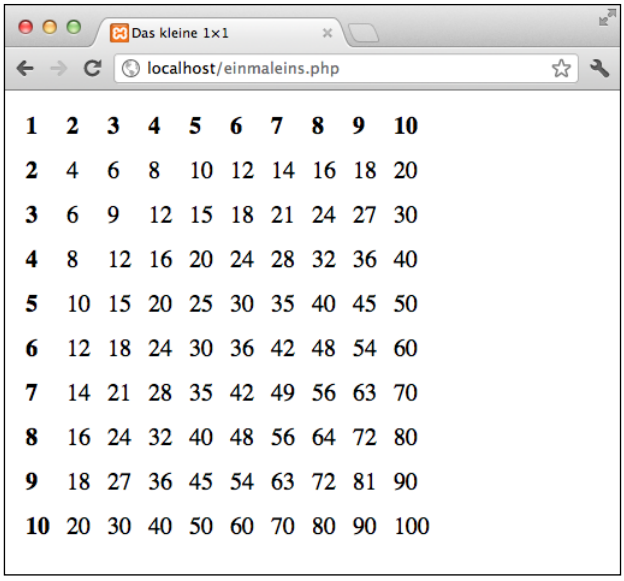
Setzen wir unsere Überlegungen zu einem vollständigen Script zusammen und fügen die notwendigen HTML-Anweisungen hinzu, dann erhalten wir etwa das folgende Ergebnis (Listing 3.19):

```
<html>
  <head>
    <title>Das kleine 1&times;1</title>
  </head>
```

```
<body>
  <table cellpadding="5">
  <?PHP
    for ($reihe = 1; $reihe <= 10; $reihe++) {
      echo "<tr>";
      for ($spalte = 1; $spalte <= 10; $spalte++) {
        echo "<td>";
        if ($reihe == 1 || $spalte == 1) echo "<strong>";
        echo $reihe * $spalte;
        if ($reihe == 1 || $spalte == 1) echo "</strong>";
        echo "</td>";
      }
      echo "</tr>";
    }
  ?>
  </table>
</body>
</html>
```

Listing 3.19: Das Script gibt das kleine Einmaleins sauber als Tabelle mit hervorgehobenen Spalten-/Reihenköpfen aus.

Das Ergebnis dieses Scripts sehen Sie in Abbildung 3.4.



1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Abbildung 3.4: Der Inhalt der Tabelle wird mit PHP erzeugt, bei der Gestaltung werden die üblichen HTML-Kommandos benutzt, die wieder via PHP gesetzt werden.

3.8 Zusammenfassung

Ein PHP-Script kann anhand der logischen Wahrheitswerte TRUE und FALSE Entscheidungen treffen. Ein logischer Wahrheitswert kann durch ein vorangestelltes Ausrufezeichen ! negiert werden. Bei einer Entscheidung werden Vergleiche zwischen Aussagen und Werten durchgeführt und Entscheidungen nach dem Muster Wenn/dann/andernfalls gefällt. Sich wiederholende Arbeitsschritte werden in Schleifen formuliert, die entweder abzählbar durchlaufen werden (`for`) oder so lange, bis eine Abbruchbedingung erfüllt ist (`while`, `do`).

3.9 Fragen und Antworten

1. Was ist der Unterschied zwischen einer annehmenden und einer ablehnenden Schleife?

   Bei einer annehmenden Schleife wird die Abbruchbedingung erst nach dem Schleifendurchlauf überprüft, sie wird also mindestens einmal ausgeführt. Eine ablehnende Schleife überprüft vor dem Start, ob die Bedingung erfüllt ist, und führt die Schleife unter Umständen überhaupt nicht aus.

2. Welche Ausdrücke sind in PHP FALSE?

   Generell sind alle Ausdrücke TRUE – mit drei Ausnahmen: Ein Ausdruck ist vom Typ NULL, hat den Wert 0 oder ist leer.

3. Wann ist eine AND-, wann eine OR-Verknüpfung TRUE?

   Eine AND-Verknüpfung ist dann (und nur dann) TRUE, wenn alle verknüpften Ausdrücke TRUE sind. Bei einer OR-Verknüpfung genügt es, wenn mindestens einer der Ausdrücke TRUE ist.

4. Warum ist eine `if`-Abfrage nach dem Muster `if ($ = 10)` ein Fehler? Und wie behebt man ihn?

   Durch das einfache Gleichheitszeichen wird kein Vergleich, sondern eine Zuweisung formuliert. Die Variable `$a` erhält also den Wert `10`. Dieser Ausdruck ist immer TRUE, die `if`-Anweisung wird also immer ausgeführt. Einen Vergleich formulieren Sie mit einem doppelten Gleichheitszeichen: `if ($a == 10)`.

5. Wie können Sie in einer `if/else`-Abfrage mehrere Anweisungen ausführen?

   Die Anweisungen werden von einer Mengenklammer umschlossen.

3.10 Übungen

Die Antworten zu den folgenden Fragen finden Sie im Anhang.

1. Schreiben Sie eine Schleife, die Ihnen die Vielfachen von 7 nach dem Muster `1 * 7 = 7` bis zu `10 * 7 = 70` ausgibt.
2. Formulieren Sie die Schleife so um, dass sie mit beliebigen Zahlen funktioniert.
3. Ändern Sie die Schleife so, dass die Ausgabe `7 14 21 … 70` lautet. Arbeiten Sie ohne Rechenoperation innerhalb der Schleife.
4. Schreiben Sie ein kleines Programm, das via GET eine ganze Zahl annimmt und überprüft, ob es sich dabei um eine gerade oder ungerade Zahl handelt.

Copyright