PEARSON Education

ADDISON-WESLEY

MASTER CLASS

Andreas Barchfeld

PHP

Einstieg für Anspruchsvolle

ADDISON-WESLEY [in Kooperation mit] PEARSON Studium

4

Programmstrukturen

Bevor wir uns den komplexeren Datentypen zuwenden, sollen an dieser Stelle Programmstrukturen besprochen werden. Unter dem Begriff Programmstruktur summiert man Eigenschaften von Programmiersprachen, mit denen man Einfluss auf den logischen Ablauf eines Programms und/oder den physikalischen Aufbau eines Programms hat. In PHP handelt es sich um Verzweigungen, Schleifen und (externe) Unterprogramme.

Einfache Programme benötigen keine Strukturen. Diese Programme werden von oben nach unten abgearbeitet. Sobald sie aber auf interne oder externe Ereignisse reagieren müssen, benötigen Sie spezielle Programmstrukturen, um darauf zu reagieren. Diese Ereignisse können Rechenergebnisse oder Eingaben des Benutzers sein.

In PHP finden Sie für solche Ereignisse zwei Arten von Strukturen. Es handelt sich um Verzweigungen und Schleifen. In der Einführung in Kapitel 1 haben Sie bereits eine einfache Verzweigung kennen gelernt. Es handelt sich um die `if`-Abfrage. Als Beispiel für Schleifen haben Sie die `for`-Schleife im Übersichtskapitel kennen gelernt.

Allgemein kann man sagen, dass eine Verzweigung aufgrund eines logischen Wertes verschiedene Programmteile ausführt. Bei einer Schleife wird derselbe Programmteil mehrfach wiederholt. Die Steuerung dieser Wiederholung geschieht ebenfalls über eine logische Variable.

Eine logische Variable ist streng genommen ein logischer Ausdruck. Wie Sie bereits wissen, kann ein logischer Ausdruck für sich allein stehen oder das Ergebnis einer Anweisung sein.

Wenn sich bei der Programmierung bestimmte Verarbeitungsschritte wiederholen, so ist das ein Zeichen dafür, diese Arbeitsschritte in ein Unter-

programm auszulagern. Sie müssen dann nur noch dieses Unterprogramm aufrufen. Wollen Sie diese Arbeitsschritte ausführen oder sollten sich irgendwann Änderungen an diesen Arbeitsschritten ergeben, müssen Sie diese nur im Unterprogramm durchführen. An allen Stellen des Programms haben Sie dann automatisch diese Änderungen durchgeführt. In PHP werden Unterprogramme durch Funktionen realisiert.

Da Programme recht umfangreich und komplex werden können, hat man sich sehr früh in der Geschichte der IT mit diesem Problem auseinander gesetzt. Zudem benötigt man eine Methode, die Arbeitsschritte mit den Fachleuten des Auftraggebers durchzugehen, die von Programmierung nicht unbedingt viel verstehen. Hierzu wurden Mitte der 60er-Jahre international die Programmablaufpläne eingeführt, kurz PAP. Diese Ablaufpläne haben auch im Zeitalter der Objektorientierung ihren Sinn nicht verloren. Man kann mit Papier und Bleistift ein Problem mit diesen Ablaufplänen sehr schnell bildlich darstellen. In diesen Ablaufplänen stehen Symbole für bestimmte Strukturen innerhalb eines Programms oder einer Funktion. Diese Symbole sind in Deutschland durch die DIN 66001 geregelt. Dies klingt jetzt komplizierter, als es tatsächlich ist. Im Folgenden werden wir ein paar dieser Symbole kennen lernen. Wenn Sie sich mit diesem Thema einmal näher beschäftigen wollen, brauchen Sie in einer Suchmaschine nur den Begriff „DIN 66001“ zu suchen. Die DIN-Norm selber finden Sie beispielsweise unter der Adresse *http://www.fh-jena.de/~kleine/history/software/DIN66001-1966.pdf*

4.1 Verzweigungen

PHP kennt zwei Arten von Verzweigungen: die einfache Wenn-dann-Beziehung, die typischerweise dann angewandt wird, wenn der weitere Verlauf des Programms nur von einer Entscheidung abhängt. Hängt das weitere Vorgehen des Programms aber von mehreren Zuständen ab, von denen nur einer wahr sein kann, so ist eine Realisierung mit einer `if`-Abfrage in vielen Fällen sehr umständlich. Für solche Mehrfachbedingungen gibt es in PHP die `switch`-Anweisung. Ein Sonderfall der `if`-Abfrage ist das so genannte „tenäre“ if.

4.1.1 Die if-Abfrage

Die einfachste Form einer Programmverzweigung ist die „Wenn-dann“-Struktur. Wie der Name schon andeutet, werden bestimmte Anweisungen nur dann ausgeführt, wenn eine logische Bedingung den Wert `true` ergibt. In Abbildung 4.1 sehen Sie den prinzipiellen Aufbau einer sol-

chen `if`-Verzweigung als Teil eines Programmablaufplans dargestellt. Sollte aufgrund der logischen Bedingung der Ja-Zweig abgearbeitet werden, wird anschließend mit dem Programmteil fortgesetzt, der ansonsten direkt ausgeführt worden wäre.

Ein Beispiel für eine solche Abfrage finden Sie in Listing 4.1. Es wird geprüft, ob die logische Variable `$error` den Wert `true` besitzt. Sollte dies der Fall sein, wird der Ablauf mit der Bearbeitung des Ja-Zweiges fortgesetzt. Im Beispiel wird die Funktion `die()` von PHP ausgeführt. Diese Funktion beendet das Programm mit der Ausgabe der übergebenen Zeichenkette. In diesem Fall werden alle Anweisungen nach der `if`-Abfrage nicht mehr ausgeführt. In Listing 4.1 besteht der Ja-Zweig nur aus einer einzigen Anweisung. Daher kann der PHP-Interpreter genau entscheiden, welche Anweisungen zum Ja-Zweig gehören und welche nicht.

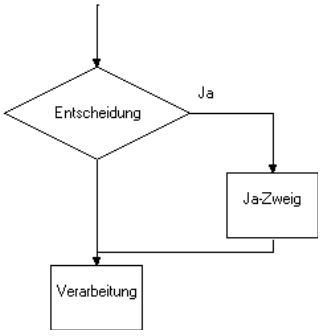


Abbildung 4.1
Struktur der if-Abfrage

Listing 4.1
Eine einfache if-Bedingung mit einer Anweisung

```
<?PHP

$error = false;

if ( $error == true)
   die("Programm wird wegen eines Fehlers beendet.");

echo "Programm nach if-Abfrage.\n";
?>
```

Wenn Sie aber im Ja-Zweig mehrere Anweisungen verarbeiten müssen, kann der PHP-Interpreter ohne Unterstützung nicht entscheiden, wo der Ja-Zweig endet und der „normale“ Programmlauf beginnt. In diesem Fall müssen Sie die Anweisungen mit den geschweiften Klammern begrenzen. Ein Beispiel sehen Sie in Listing 4.2. Wenn das Programm in den Ja-Zweig springt, werden zwei Variablen mit Werten besetzt und erst dann die Fehlermeldung mit der `die()`-Funktion ausgegeben. Im Beispiel ist dies sehr einfach gehalten. Fehlernummer und Fehlertext können natürlich beliebig kompliziert ermittelt werden.

Listing 4.2
Eine einfache if-Bedingung mit mehreren Anweisungen

```
<?PHP

$error = false;

if ( $error == true) {
   $err_num = 25;
   $err_text = "Fehlermeldung im Klartext"
   die("Fehler: $err_num \n $err_text");
   echo "Dies sollte nie erscheinen ... "
}

echo "Programm nach if-Abfrage.\n";
?>
```

Die Anweisungen werden mit `{` und `}` geklammert, so dass PHP genau erkennen kann, welche Anweisungen zum Ja-Zweig gehören. Wenn Sie diese Klammern vergessen, wird PHP diesen „Fehler“ nicht erkennen. Es handelt sich um einen logischen Fehler in Ihren Überlegungen, nicht um einen Fehler im Sinne der PHP-Syntax! Als warnendes Beispiel ist Listing 4.3 abgedruckt. Es handelt sich mit wenigen Ausnahmen um das Programm aus Listing 4.1. Die erste Ausnahme ist der Wert der Variablen `$error`. Mit diesem Wert dürfte das Programm nicht in den Ja-Zweig der Abfrage springen. Um den Programmlauf klarer zu gestalten, werden die beiden Variablen `$err_num` und `$err_text` mit Werten vorbelegt; eine Methode, die Sie zum Standard für Ihre Programme erheben sollten. Sie können durch diese geringe Mehrarbeit viele Stunden Fehlersuche vermeiden.

Listing 4.3
Fehlende Klammerung bei der if-Abfrage

```
<?PHP

$error = false;
$err_num = 1;
$err_text = "Kein Fehler";

if ( $error == true)
   $err_num = 25;
   $err_text = "Fehlermeldung im Klartext";
   die( "Fehler: {$err_num} \n {$err_text}" );

echo "Programm nach if-Abfrage.\n";
?>
```

Der PHP-Interpreter erkennt nur die markierte Zeile als `if`-Zweig, da die Klammern fehlen! Die folgenden beiden Zeilen werden nicht als Anweisungen dieses `if`-Zweiges erkannt und immer ausgeführt: Die Variable `$err_text` bekommt eine neue Zeichenkette zugewiesen, und das Programm wird über die `die()`-Funktion beendet. Da die Variable `$err_num`

noch ihren ursprünglichen Wert behalten hat, kommt es zur Ausgabe von Listing 4.4.

```
Fehler: 1
 Fehlermeldung im Klartext
```

Listing 4.4
Ausgabe des fehlerhaften Programms aus Listing 4.3

In unserem Beispiel sind die Auswirkungen überschaubar, da das Programm abbricht. Wenn Sie allerdings Berechnungen durchführen, kann dies zu schwer auffindbaren Fehlern führen. Sie sollten sich angewöhnen, die Klammern immer zu setzen! Einige PHP-Editoren blenden diese Klammerung mittlerweile automatisch ein. Sie müssen die Klammern also bewusst löschen. Ein Indiz für die Beliebtheit dieses Fehlers.

Sie können `if`-Abfragen beliebig ineinander verschachteln. Eine einfache Verschachtelung können Sie in Listing 4.5 sehen. Abhängig von der Variablen `$val_outer` werden die Anweisungen im Körper der `if`-Abfrage abgearbeitet. Im Beispiel ist dies der Fall, da die Bedingung der äußeren `if`-Abfrage erfüllt ist, `true`. In der `printf()`-Funktion wird von den Formatierungseigenschaften dieser Funktion Gebrauch gemacht. Die Formatanweisung „`%03d`“ weist die Funktion an, eine Integerzahl mit der Länge 3 auszugeben. Zusätzlich sollen evtl. vorhandene führende Leerstellen mit der Zahl 0 gefüllt werden.

```
<?PHP

$val_outer = 5;

$val_inner = "MwSt";

if ( $val_outer >= 5 ) {

   printf( "Warengruppe %03d: ", $val_outer );

   if ( $val_inner == "MwSt" ) {

      printf( " 7%% %6s \n", $val_inner );

   }
}
?>
```

Listing 4.5
Verschachtelte if-Abfrage

Bei der inneren `if`-Abfrage wird auf die Gleichheit einer Zeichenkette geprüft. Wenn die Variable `$val_inner` die Zeichenkette „MwSt“ enthält, ergibt diese Abfrage den Wert `true`, und die Anweisungen im Körper der Abfrage werden ausgeführt. Da es sich hier eindeutig um eine Zeichenkette handelt, benötigen Sie an dieser Stelle keine Abfrage auf die Identität der Variablen (`===`).

Die `printf()`-Funktion enthält innerhalb der Formatanweisung noch ein Beispiel für die Darstellung des Format-Sonderzeichens `%`. Da mit dem `%`-Zeichen eine Formatangabe eingeleitet wird, muss zur Darstellung des Zeichens selber dieses doppelt angegeben werden. Bei Formatangaben wie dieser sollten Sie, wenn möglich, auf die Leerzeichen achten. Nicht, weil PHP die Formatangabe `" 7%%%6s \n"` nicht verstehen würde, PHP kann diese Formatanweisung richtig interpretieren. Wenn Sie allerdings mehrere Variablen auf diese Art ausgeben, wird die Formatangabe für den Leser Ihres Quellcodes sehr schnell sehr unübersichtlich.

Listing 4.6
Ergebnis der Auswertung aus Listing 4.5

```
Warengruppe 005:  7%   MwSt
```

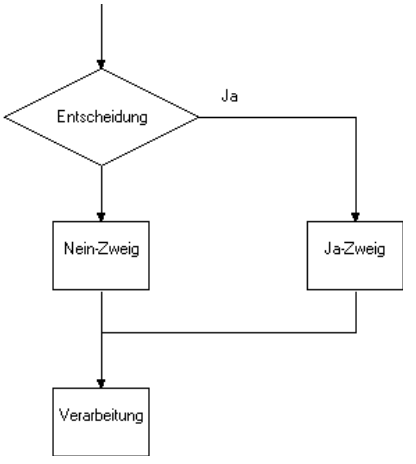
4.1.2 Die if-else-Abfrage

Nun gibt es nicht nur Entscheidungen nach dem Prinzip “wenn – dann”, sondern häufig existiert noch eine Alternative, der „Sonst“-Zweig einer Entscheidung. In PHP können Sie solche Entscheidungen ebenfalls treffen. Der dazugehörende prinzipielle Ablauf ist in Abbildung 4.2 als PAP dargestellt.

Eine reale Anwendung finden Sie in Abbildung 4.3, der Berechnung von Kindergeld nach den zurzeit geltenden Regelungen. In dieser Form ist die Berechnung noch nicht vollständig. In bestimmten Fällen wird das Kindergeld falsch berechnet. Sie werden den Fehler vielleicht schon bemerkt haben. Die vollständige Fassung werden Sie im nächsten Unterkapitel kennen lernen.

Bei bis zu drei Kindern beträgt das Kindergeld 154 Euro pro Kind, darüber beträgt der Satz 179 Euro.

Abbildung 4.2
Struktur der if-else-Abfrage



Eine Umsetzung dieser Entscheidungslogik in ein PHP-Programm finden Sie in Listing 4.7. Die Variable `$children_no` enthält die Anzahl der Kinder, für die das Kindergeld berechnet werden soll. Der Ausdruck `$children_no > 3` ergibt für den Wert aus Listing 4.7 den Wert `true`. Damit wird das Kindergeld mit einem Satz von 179 Euro berechnet. Wenn die Bedingung den Wert `false` ergibt, wird in den Anweisungsblock nach dem `else` gesprungen. Für diesen Anweisungsblock gelten die gleichen Regeln wie für den Anweisungsblock nach dem `if`. Das heißt, alle zu diesem Zweig gehörenden Anweisungen werden durch geschweifte Klammern begrenzt. Sollte der Zweig nur aus einer Anweisung bestehen, kann man die geschweiften Klammern fortlassen. Sie wären in Listing 4.7 also nicht notwendig.

Listing 4.7
Kindergeldberechnung mit if-else-Abfrage

```
<?PHP

$children_no = 4;

if ( $children_no > 3 ) {

   $child_money = $children_no * 179;
}
else {

   $child_money = $children_no * 154;

}

printf( "Das Kindergeld beträgt %4d Euro.\n", $child_money );

?>
```

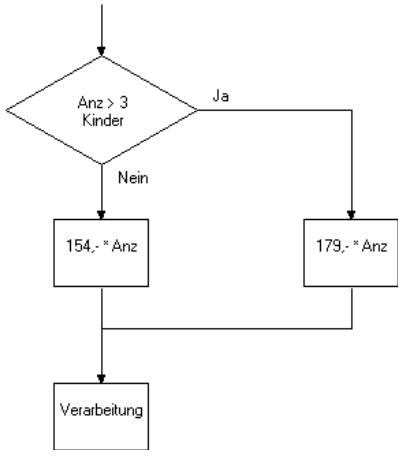


Abbildung 4.3
Berechnung des Kindergeldes

Das Programm hat allerdings wie bereits erwähnt einen Schönheitsfehler. Wenn Sie die Anzahl der Kinder auf 0 ändern und anschließend das Programm starten, sehen Sie, was ich meine. Schlimmer noch, das Programm funktioniert auch bei negativen Werten für die Variable `$children_no`! Wenn Sie die Logik etwas anders strukturieren, können Sie dieses Problem beheben. Eine Möglichkeit finden Sie in Listing 4.8. Innerhalb der `if`-Bedingung können Sie mehrere Bedingungen verknüpfen. In diesem Beispiel wird getestet, ob die Variable `$children_no` größer als 0 und kleiner/gleich 3 ist. In diesen Bereich fallen nur die Zahlen 1, 2 und 3. Die beiden Bedingungen werden durch den logischen `&&`-Operator verbunden, da beide Bedingungen gleichzeitig `true` ergeben müssen.

Listing 4.8
Logisch richtige Berechnung des Kindergeldes

```
<?PHP

$children_no = 4;

if ( ($children_no > 0) && ($children_no <= 3) ) {

   $child_money = $children_no * 154;
}
else {

   $child_money = $children_no * 179;

}

printf( "Das Kindergeld beträgt %4d Euro.\n", $child_money );

?>
```

Das Programm stimmt zwar von der reinen Logik der Berechnung her, aber wenn Sie für die Variable `$children_no` den Wert 1000 einsetzen, wird Ihnen dieses Programm aus Listing 4.7 oder Listing 4.8 anstandslos einen Betrag von 179.000 Euro berechnen. Nun sind aber 1000 Kinder pro Ehepaar eher unwahrscheinlich. Die Lösung dieses Problems liegt in der mehrfachen Abfrage.

4.1.3 Die if-else-if-else-Abfrage

Am Beispiel der Kindergeldberechnung erkennen Sie, dass die Anzahl der Entscheidungen, die ein Programm treffen muss, sehr schnell eskaliert, vor allem dann, wenn Sie Warnungen in das Programm einbauen wollen.

In solchen Fällen können Sie die mehrfache `if`-Abfrage einsetzen. Bei dieser Entscheidung reicht ein einfaches „wenn – dann – sonst“, also `if – else`, nicht mehr aus. Im `else`-Zweig der `if`-Abfrage wird ein weiteres `if` angehängt. Schematisch können Sie eine solche Mehrfachverzweigung als PAP in Abbildung 4.4 sehen. Lautet das Resultat eines Vergleichs `true`, so wird in den entsprechenden Ja-Zweig gesprungen. Anschließend fährt das Programm mit der Verarbeitung am Ende der Mehrfachverzweigung fort. Wenn Sie den Programmablauf aus Abbildung 4.4 in PHP umsetzen wollen, müssen Sie wie in Listing 4.9 gezeigt vorgehen. Dieses Listing ist kein funktionsfähiges PHP-Programm. Es zeigt Ihnen das prinzipielle Vorgehen bei der Programmierung von Mehrfachentscheidungen. Wie Sie erkennen können, wird im `else`-Zweig der `if`-Anweisung eine weitere `if`-Anweisung angefügt. Dabei werden die beiden Worte `else` und `if` zu einem Wort zusammengezogen: `elseif`. Wenn Sie beide Worte getrennt lassen, wird das PHP-Programm ebenfalls laufen. Diese Form ist aber nicht dokumentiert. Es handelt sich auch um eine andere Form. Während der Begriff `elseif` zur oberen `if`-Anweisung gehört, starten Sie mit `else if` eine neue `if`-Abfrage innerhalb der nun äußeren `if`-Abfrage.

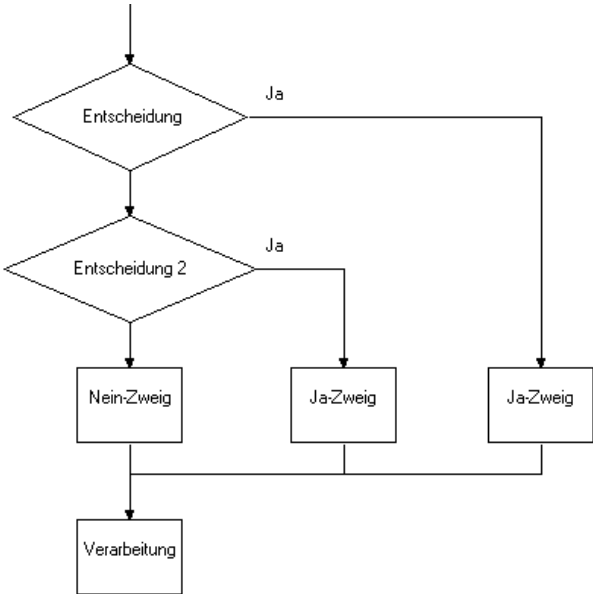
Diese Verzweigung können Sie nicht nur einmal, sondern beliebig oft durchführen. Entsprechend lang können solche `if`-Anweisungen werden. Sollte eine solche Konstruktion über eine Druckseite gehen, sollten Sie darüber nachdenken, ob Sie nicht eine andere Form wählen können. Solche „Bandwürmer“ neigen dazu, recht schnell unübersichtlich zu werden. Eine Möglichkeit ist der Einsatz von Funktionen, zu denen wir noch kommen werden.

Listing 4.9
Pseudo-Listing zu Abbildung 4.4

```
if ( Entscheidung ) {
   rechter Ja-Zweig
}
elseif ( Entscheidung 2 ) {
   mittlerer Ja-Zweig
}
else {
   Nein-Zweig
}
```

Als praktisches Beispiel soll hier wieder die Berechnung des Kindergeldes benutzt werden. Im Listing 4.10 finden Sie ein komplettes Programm für die Befehlszeile (CLI).

Abbildung 4.4
Struktur der if-else-if-else-Abfrage



In der ersten `if`-Anweisung wird geprüft, ob beim Aufruf des Programms die Anzahl der Kinder mit angegeben wurde. Dies ist der Fall, wenn die Anzahl der Übergabeparameter exakt zwei beträgt. Der erste Übergabeparameter ist der Name des Programms, der zweite ist die Anzahl der Kinder. Wenn Sie nicht die richtige Anzahl an Parametern übergeben, wenn also die Anzahl der Übergabeparameter nicht gleich zwei ist, wird das Programm mit einer Fehlermeldung abgebrochen. Der Befehl hierzu lautet `die()`. Diese Funktion kann eine Zeichenkette übernehmen, die vor dem Verlassen des Programms auf dem aktuellen Ausgabegerät dargestellt wird. In diesem Fall ist es die Konsole. Wenn Sie mit einem Browser arbeiten, wird diese Zeichenkette im Browser dargestellt. In `$argv[0]` ist der Name des Programms hinterlegt, so wie er auf der Befehlszeile eingegeben wurde.

In Abbildung 4.5 sehen Sie den Aufruf des Programms ohne die zusätzliche Angabe über die Anzahl der Kinder.

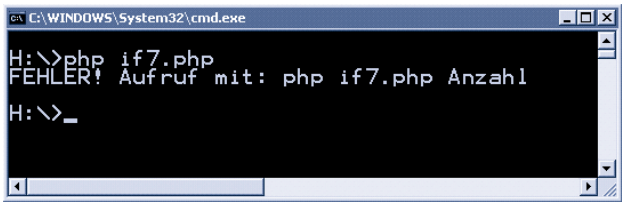
Wenn Sie das Programm mit der Angabe der Kinderzahl aufrufen, wird das Programm an dieser Stelle fortgeführt und die Anzahl der Kinder in der Variablen `$children_no` abgespeichert. Da alle Übergabeparameter als Zeichenkette übergeben werden, wird dabei der Datentyp in Integer geändert. Anschließend werden die zwei noch benötigten Variablen deklariert. Ein entsprechender Aufruf des Programms ist in Abbildung 4.6 zu sehen.

Listing 4.10
Kindergeldberechnung mit if – else – if – else

```
<?php

if ( $argc != 2 ) {
   die("FEHLER! Aufruf mit: php $argv[0] Anzahl\n");
}

$children_no = (int)$argv[1];
$child_money = 0;
$error_msg = "";

if ( $children_no < 0 ) {
   die("Die Anzahl der Kinder ist fehlerhaft ( kleiner 0 )!\n");
}
elseif ( $children_no == 0 ) {
   $child_money = 0;
}
elseif ( $children_no > 0 && $children_no <= 3 ) {
   $child_money = $children_no * 154;
}
elseif ( $children_no > 3 && $children_no <= 6 ) {
   $child_money = $children_no * 179;
}
elseif ( $children_no > 6 && $children_no <= 20 ) {
   $child_money = $children_no * 179;
   $error_msg = "Warnung!\n";
   $error_msg .= "Das Kindergeld für $children_no Kinder wurde ";
   $error_msg .= "berechnet.";
}
else {
   die("Für $children_no wird kein Kindergeld berechnet.");
}

if ( $error_msg ) {
   printf("%s\nDer Kindergeldbetrag beläuft sich auf %4d Euro.\n",
      $error_msg, $child_money);
}
else {
   printf("Der Kindergeldbetrag beläuft sich auf %4d Euro.\n",
      $child_money );
}

?>
```

Dann beginnt der für dieses Kapitel interessante Teil. Die `if`-Anweisung beginnt mit der Abfrage auf einen negativen Wert. Wenn Sie dem Programm als Kinderzahl zum Beispiel `-4` übergeben, ist dies sicherlich ein Fehler.

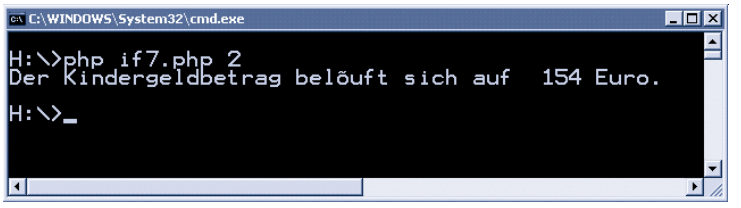
Abbildung 4.5
Einfacher Aufruf zur Kindergeldberechnung



Ist diese Bedingung `false`, wird bei der nächsten Bedingung fortgesetzt. Wenn Sie keine Kinder haben, besteht sicherlich auch kein Anrecht auf Kindergeld, also wird der Betrag hierfür auf null gesetzt.

Wenn Sie ein, zwei oder drei Kinder haben, wird die nächste Bedingung den Wert `true` liefern und die Anweisungen im Block der `elseif`-Bedingung ausführen und das Programm nach der `if`-Anweisung fortsetzen (Abbildung 4.6).

Abbildung 4.6
Programmaufruf für zwei Kinder des Kindergeldprogramms



Bei vier, fünf oder sechs Kindern wird der nächste `elseif`-Zweig ausgeführt. Die Festlegung der oberen Grenze von sechs Kindern für diese Abfrage ist willkürlich. Bei mehr als sechs Kindern wird eine Warnung ausgegeben, wobei die Anzahl der Kinder (`$children_no`) in der Warnmeldung erscheint. Diese Warnung wird in der Variablen `$error_msg` abgespeichert. Bei der Zuweisung der Zeichenkette an diese Variable wird der Verknüpfungsoperator `.` für Zeichenketten in der Kurzschreibweise benutzt (`.=`).

Die nächste Grenze ist in diesem Beispiel ebenfalls willkürlich. Bei mehr als 20 Kindern wird die Berechnung des Kindergeldes abgebrochen. Das Programm geht von einer Fehleingabe aus.

Nachdem das Kindergeld berechnet wurde, testet die direkt darauf folgende `if`-Anweisung auf das Vorhandensein einer Fehlermeldung. Die Variable `$error_msg` wurde zu Beginn des Programms mit einer leeren Zeichenkette deklariert. Wenn an dieser Stelle des Programms diese Zeichenkette immer noch leer ist, wird vom PHP-Interpreter die leere Zeichenkette in einen logischen Wert übersetzt: `false`. Enthält die Variable `$error_msg` mindestens ein Zeichen, so lautet das Ergebnis der Umwandlung in einen logischen Datentyp `true`, die Meldung wird ausgegeben.

Diese Art der `if`-Anweisung ist eine Kurzform für:

```
if ( $error_msg !== "" ) {
```

Sie machen hier direkt Gebrauch von der automatischen Typkonvertierung von PHP. Dieser Stil ist an C/C++ angelehnt, wo solche abkürzenden Schreibweisen üblich sind.

4.1.4 Das tenäre if

Für einfache `if`-Anweisungen gibt es eine abkürzende Schreibweise, den tenären Operator, auch tenäre `if`-Anweisung genannt. Dieser Operator hat den Aufbau:

```
Logischer Ausdruck ? true-Ausdruck : false-Ausdruck
```

Wenn der logische Ausdruck links vom Fragezeichen den Wert `true` ergibt, wird der `true`-Ausdruck ausgeführt, anderenfalls der `false`-Ausdruck. Der logische Ausdruck entspricht hierbei dem der üblichen `if`-Anweisung.

```
$child_no < 4 ? $value = 154 : $value = 179;
```

Listing 4.11
Tenäre Version von Listing 4.8

Listing 4.7 ist somit eine Kurzform der `if`-Anweisung aus Listing 4.8.

```
if ( $child_no < 4 ) {
   $value = 154;
}
else {
   $value = 179;
}
```

Listing 4.12
if-Anweisung von Listing 4.7

Der tenäre Operator hat seine Wurzeln ebenfalls in C/C++. Sinnvoll ist der Einsatz dieses Operators nur bei Entscheidungen, die den Umfang von Listing 4.7 nicht wesentlich überschreiten, zumal Sie auf jeden Fall einen „`else`-Zweig“ für diesen Operator benötigen. Sollte dieser nicht vorhanden sein, bekommen Sie einen Fehler des PHP-Interpreters gemeldet.

Wenn Sie sich daran erinnern, dass man die Klammern fortlassen kann, sollte nur eine Anweisung im jeweiligen Zweig der `if`-Anweisung stehen, können Sie Listing 4.8 kürzen. Wenn Sie dann noch bedenken, dass Anweisungen durch ein Semikolon abgeschlossen werden und nicht durch einen Zeilenumbruch, können Sie Listing 4.8 auf die Form von Listing 4.9 bringen.

```
if ( $child_no < 4 ) $value = 154; else $value = 179;
```

Listing 4.13
Gekürzte Fassung von Listing 4.8

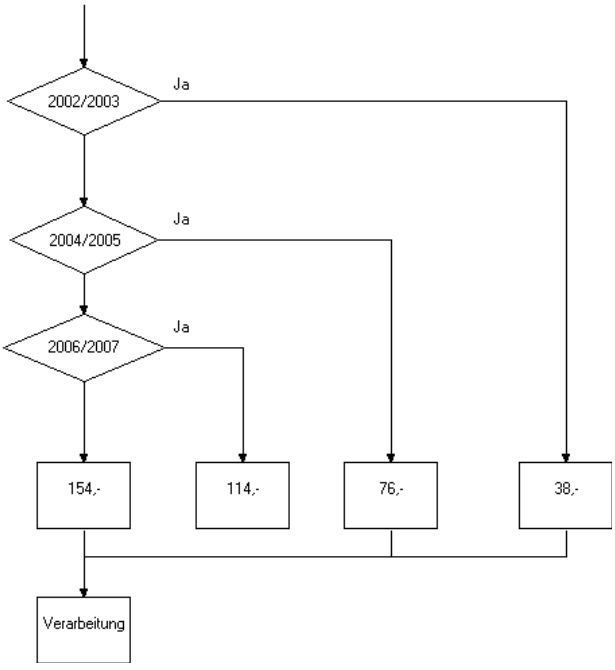
4.1.5 Die switch-Anweisung

Wenn Sie mehrere `elseif`-Zweige innerhalb einer `if`-Anweisung haben, sollten Sie sich überlegen, ob nicht eine andere Struktur von PHP an dieser Stelle sinnvoller ist, die `switch`-Anweisung. Wenn Sie einen Blick auf die Abbildung 4.7 werfen, so lässt sich dieser Abschnitt eines Programms sicherlich mit einer `if`-Anweisung realisieren. Diese `if`-Anweisung könnte mit der Bedingung

```
if ( $jahr == 2002 || $jahr == 2003 ) {
```

beginnen. Aufgrund der Daten, mit denen in diesem Beispiel eine Entscheidung getroffen wird, kann man allerdings auch sehr gut eine `switch`-Anweisung benutzen.

Abbildung 4.7
Schema einer switch-Anweisung



Den prinzipiellen Aufbau einer solchen `switch`-Anweisung sehen Sie in Listing 4.14.

Grundlage für die `switch`-Anweisung ist der Ausdruck in den Klammern nach dem Schlüsselwort `switch`. Dieser Ausdruck muss einen Wert zurückliefern. Dieser Wert kann eine Zahl, eine Zeichenkette oder ein logischer Wert sein. Im letzten Fall ist die `switch`-Anweisung aber in den meisten Fällen nicht die optimale Wahl.

Listing 4.14
Struktur der switch-Anweisung

```
switch ( Ausdruck ) {

   case value_1:
      Anweisungen
      [break;]

   case value_2:
      Anweisungen
      [break;]

   [default:
      Anweisungen
      break;]

}
```

Die einzelnen Fälle (`case`) werden von oben nach unten abgearbeitet. Trifft ein Wert zu, werden die Anweisungen nach der betreffenden `case`-Klausel bis zum nächsten `break` abgearbeitet. Es können also durchaus mehrere Anweisungen unter dem `case` stehen. Diese Anweisungen müssen nicht in geschweifte Klammern gesetzt werden. Die Klammerung erfolgt durch `case` – `break`.

Am Ende der `switch`-Anweisung kann eine Klausel folgen, die automatisch dann abgearbeitet wird, wenn keiner der aufgeführten Werte in den `case`-Klauseln zutrifft. Aus diesem Grund ist diese `default`-Klausel in Listing 4.10 in eckige Klammern gesetzt.

Nach der Ausführung der einzelnen Anweisungen wird das Programm mit den Anweisungen direkt hinter der schließenden Klammer der `switch`-Anweisung fortgesetzt.

Die Umsetzung des Entscheidungsbaums aus Abbildung 4.7 finden Sie in Listing 4.15. Zu Beginn des Programms wird die Variable `$value` mit dem Wert null deklariert. Eine Vorsichtsmaßnahme, die sich in längeren Programmen bewährt. Direkt im Anschluss wird geprüft, ob das Programm mit vernünftigen Werten gestartet wurde. So sagt der Programmablaufplan aus, dass vor dem Jahr 2002 kein Betrag errechnet werden soll. Zudem muss dem Programm natürlich eine Jahreszahl übergeben werden. Wenn also die Anzahl der Übergabeparameter (`$argc`) nicht den Wert zwei ergibt oder die dem Programm übergebene Jahreszahl (`$argv[1]`) kleiner als 2002 ist, so wird das Programm mit einer Fehlermeldung abgebrochen. Diese Schritte finden Sie nicht in Abbildung 4.7. Sie dienen an dieser Stelle „nur“ als Beispiel für die Überprüfung von Übergabewerten.

Listing 4.15
Switch-Programm zu Abbildung 4.7

```
<?PHP
$value = 0;

if ( $argc != 2 || $argv[1] < 2002 ) {
   die("FEHLER! Aufruf mit $argv[0] Jahr ( >2001 )\n");
}

switch ( $argv[1] ) {
   case 2002:
      $value = 38;
      break;

   case 2003:
      $value = 38;
      break;

   case 2004:
      $value = 76;
      break;

   case 2005:
      $value = 76;
      break;

   case 2006:
      $value = 114;
      break;

   case 2007:
      $value = 114;
      break;

   default:
      $value = 154;
}
printf("Betrag: EUR %4d,-\n", $value );
?>
```

Der auszuwertende Ausdruck ist die Jahreszahl, daher steht sie in den Klammern der `switch`-Anweisung. Im Körper der `switch`-Anweisung werden dann alle Fälle abgearbeitet. Da nach dem Jahr 2007 immer ein und derselbe Betrag ermittelt werden soll, eignet sich dieser Schritt für den `default`-Zweig der Anweisung. Wie Sie im Listing 4.15 unschwer erkennen können, werden beispielsweise die Werte für die Jahre 2002/2003 getrennt voneinander berechnet, obwohl die Rechenschritte identisch sind. In solchen Fällen kann man einzelne `case`-Blöcke zusammenführen. Vielleicht ist Ihnen schon aufgefallen, dass die `break`-Anweisung in Listing 4.10 wie das `default`-Schlüsselwort in eckigen

Klammern gesetzt sind. Wenn gewünscht, kann man also eine `break`-Anweisung fortlassen. Wie oben schon erwähnt, werden die Anweisungen bis zum nächsten `break` ausgeführt. Man kann Listing 4.15 somit in die verkürzte Form von Listing 4.16 bringen. Beide Programme sind von ihrer Funktionalität her identisch.

Listing 4.16
Verkürzte Form der switch-Anweisung aus Listing 4.15

```
<?PHP
$value = 0;

if ( $argc != 2 || $argv[1] < 2002 ) {
   die("FEHLER! Aufruf mit $argv[0] Jahr ( >2001 )\n");
}
switch ( $argv[1] ) {
   case 2002:
   case 2003:
      $value = 38;
      break;

   case 2004:
   case 2005:
      $value = 76;
      break;

   case 2006:
   case 2007:
      $value = 114;
      break;

   default:
      $value = 154;
}
printf("Betrag: EUR %4d,-\n", $value );
?>
```

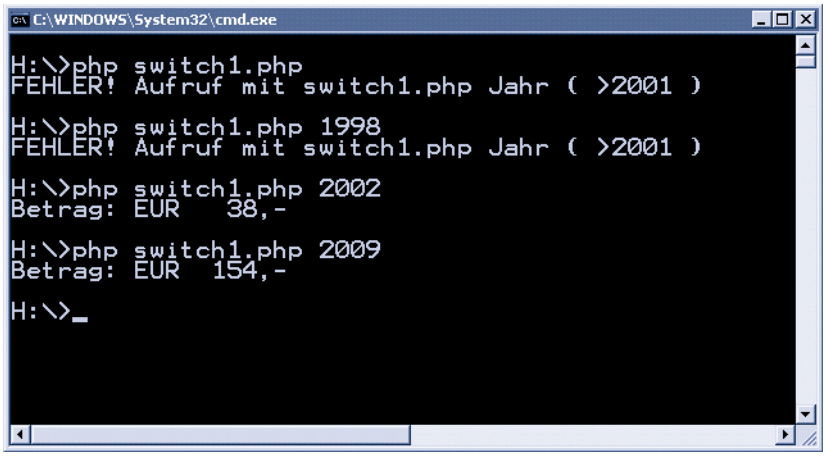


Abbildung 4.8
Ergebnisse aus dem Lauf von Listing 4.16

Wenn Sie eines der beiden Programme von der Kommandoebene aus aufrufen, bekommen Sie Ergebnisse, wie sie zum Beispiel in Abbildung 4.8 dargestellt sind.

Dieses Programm hat keine Beschränkung der Jahreszahl nach oben. Sie können also ohne weiteres den Betrag für das Jahr 3546 berechnen. In realen Programmen sollte man auch solche Grenzen beachten.

4.2 Schleifen

Immer dann, wenn Programmteile wiederholt ausgeführt werden müssen, kommen Schleifen ins Spiel. In den Programmiersprachen gibt es drei verschiedene Arten von Schleifen, die im Folgenden einzeln dargestellt werden. Jede dieser Schleifenarten hat ihre Besonderheiten. Aber man kann in vielen Fällen die eine Art einer Schleife durch eine andere Art ersetzen, ohne die Funktionalität des Programms zu ändern.

4.2.1 Die zählergesteuerte Schleife

Eine Art der Schleifen ist in diesem Buch an manchen Stellen schon benutzt worden, die zählergesteuerte Schleife. Den prinzipiellen Aufbau einer solchen Schleife können Sie in Abbildung 4.9 sehen. Die Schleife wird von einem Schleifenkopf eingeleitet, in dem der Startwert des Zählers, die Schrittweite dieses Zählers, der Maximalwert (auch Endwert genannt) und die Variable für diesen Zähler vereinbart werden. Im Schleifenkörper werden für jeden Wert des Zählers eine oder mehrere Anweisung(en) ausgeführt. Dabei muss die Zählervariable im Schleifenkörper nicht benutzt werden.

In dem Beispiel aus Abbildung 4.9 wird für die Zahlen eins bis zehn das Quadrat dieser Zahl berechnet. Die Umsetzung dieses Beispiels in ein PHP-Programm finden Sie in Listing 4.17.

Eingeleitet wird die zählergesteuerte Schleife mit dem Schlüsselwort `for`. In Klammern folgt die Deklaration der Zählervariablen und die Initialisierung dieser Variablen mit dem Startwert `($i=1`. Durch ein Semikolon getrennt wird der Test auf den Grenzwert eingefügt: `; $i <= 10`. Die Schleife wird so lange durchlaufen, wie dieser logische Ausdruck den Wert `true` ergibt. Zum Schluss wird, ebenfalls durch ein Semikolon getrennt, die Schrittweite bestimmt, mit der die Schleife die Zählervariable hochzählen soll. Da in diesem Beispiel immer um den Wert 1 erhöht wird, lautet diese Anweisung `$i++`.

Der Schleifenkörper wird durch geschweifte Klammern begrenzt. In diesem Körper wird das Quadrat der Zählervariablen berechnet, indem die Zählervariable mit sich selbst multipliziert wird. Wir hätten an dieser Stelle auch die Funktion `pow()` anwenden können. Die Multiplikation mit sich selber ist an dieser Stelle aber die schnellere Methode.

Das Ergebnis dieses kleinen Programms finden Sie in Listing 4.18. Durch die Formatierung in der `printf()`-Funktion stehen die Zahlen rechtsbündig ausgerichtet direkt untereinander.

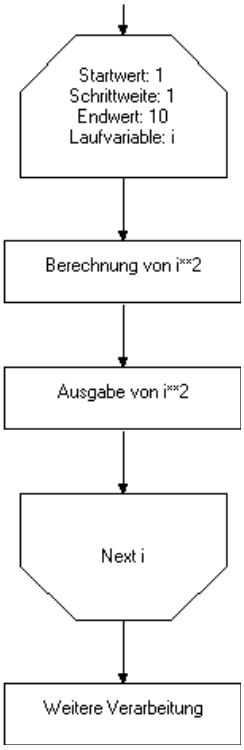


Abbildung 4.9
PAP einer zählergesteuerten Schleife

```
<?PHP

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

   $square = $i * $i;
   printf("Das Quadrat von %2d ist %4d.\n", $i, $square) ;

}

?>
```

Listing 4.17
Umsetzung des PAP aus Abbildung 4.9

Listing 4.18
Ergebnis der Quadraturberechnung aus Listing 4.13

```
Das Quadrat von  1 ist    1.
Das Quadrat von  2 ist    4.
Das Quadrat von  3 ist    9.
Das Quadrat von  4 ist   16.
Das Quadrat von  5 ist   25.
Das Quadrat von  6 ist   36.
Das Quadrat von  7 ist   49.
Das Quadrat von  8 ist   64.
Das Quadrat von  9 ist   81.
Das Quadrat von 10 ist  100.
```

Die zählergesteuerte Schleife lässt sich auf vielfache Weise beeinflussen und steuern. Wenn beispielsweise der Startwert der Schleife das Ergebnis einer Berechnung ist, können Sie im Kopf der Schleife den betreffenden Eintrag fortlassen. In Listing 4.19 finden Sie hierzu ein Beispiel.

Listing 4.19
Zählergesteuerte Schleife mit externem Startwert

```
<?PHP

$i = 1;

for ( ; $i<=10; $i++ ) {

   $square = $i * $i;
   printf("Das Quadrat von %2d ist %4d.\n", $i, $square) ;

}

?>
```

Der Startwert `$i` wird in der Klammer nicht aufgeführt. Wichtig ist bei dieser Vorgehensweise, dass das folgende Semikolon nicht fortgelassen werden darf. Dies würde zu einem Programmfehler führen.

Eine weitere Form der zählergesteuerten Schleife addiert die Schrittweite innerhalb des Schleifenkörpers zur Zählervariablen hinzu. Im Listing 4.16 wird hierzu die Addition `$i++` aus dem Schleifenkopf entfernt und in den Schleifenkörper integriert. Die Inkrementierung von `$i` muss am Ende der Schleife erfolgen. Wenn Sie dies am Beginn des Schleifenkörpers durchführen, bekommen Sie die falschen Werte berechnet. Im Beispiel von Listing 4.20 würde die Schleife bei 2 beginnen und bis 11 laufen.

Sie finden in Listing 4.20 aber noch eine weitere Veränderung der Zählervariablen `$i`. Wenn die Schleifenvariable den Wert 5 erreicht hat, wird diese Variable auf den Wert 8 gesetzt. Es werden also einige Schleifendurchläufe ausgelassen.

Listing 4.20
Zählergesteuerte Schleife mit Änderung der Zählervariablen im Körper

```
<?PHP

for ( $i=1; $i<=10; ) {

   $square = $i * $i;
   printf("Das Quadrat von %2d ist %4d.\n", $i, $square) ;

   $i++;

   if ( $i == 5 )
      $i = 8;

}

?>
```

Wenn Sie dieses Programm ausführen, erhalten Sie die Ausgabe von Listing 4.21. In dieser Schleife haben Sie die `if`-Abfrage direkt hinter dem Post-Inkrement von `$i`. Sie können diese beiden Anweisungen zu einer Anweisung zusammenführen. Dabei müssen Sie allerdings das Post-Inkrement in ein Prä-Inkrement tauschen, da die Addition von `1` ja vor dem Vergleich durchgeführt werden muss. Sie erhalten damit die `if`-Abfrage:

```
if ( ++$i == 5 )
```

Das Ergebnis entspricht wieder der Ausgabe von Listing 4.20.

Listing 4.21
Ausgabe von Listing 4.20

```
Das Quadrat von  1 ist    1.
Das Quadrat von  2 ist    4.
Das Quadrat von  3 ist    9.
Das Quadrat von  4 ist   16.
Das Quadrat von  8 ist   64.
Das Quadrat von  9 ist   81.
Das Quadrat von 10 ist  100.
```

Wenn man den Schleifenstart und die Schrittfolgenberechnung in den Körper der Schleife ziehen kann, liegt die Vermutung nicht weit, dass man dies auch mit der Abbruchbedingung machen kann. Diese Variante finden Sie in Listing 4.22.

Da die Schleife bis einschließlich zum Wert 10 laufen soll, muss diese Schleife abgebrochen werden, wenn dieser Wert überschritten wird. Die entsprechende Anweisung lautet in PHP `break`. Sie haben diese Anweisung schon im Zusammenhang mit der `switch`-Anweisung kennen gelernt. Hier erfüllt sie eine ähnliche Aufgabe.

Mit dieser Anweisung wird die Schleife abgebrochen, in der die `break`-Anweisung unmittelbar steht. Dies bedeutet bei Schleifen, die innerhalb einer äußeren Schleife eingebettet sind, dass nur die innere Schleife abgebrochen wird.

Listing 4.22
Zählergesteuerte Schleife mit Abbruchbedingung im Schleifenkörper

```
<?PHP

for ( $i=1; ; $i++ ) {

   if ( $i>10 )
      break;

   $square = $i * $i;
   printf("Das Quadrat von %2d ist %4d.\n", $i, $square) ;

}
?>
```

Diese Schreibweise ist die Abkürzung für die Anweisung `break 1`. Das lässt die Vermutung aufkommen, dass es auch Anweisungen der Form `break 2`, `break 3` etc. gibt. Während `break 1` oder `break` nur eine Ebene abbrechen, wird durch die Anweisung `break 2` auch die nächste darüber liegende Ebene abgebrochen. Sie können also mehrere ineinander verschachtelte Schleifen mit einer Anweisung abbrechen. Dies gilt aber nicht nur für Schleifen.

Schauen Sie sich hierzu das Listing 4.19 genau an. Es handelt sich um ein Programm, das vom Prinzip her den bisher vorgestellten Programmen bezüglich der `for`-Schleife ähnelt. Innerhalb der Schleife ist eine `switch`-Anweisung eingebettet, welche die Zählervariable `$i` zusätzlich um den Wert 1 erhöht, sollte der Wert 5 erreicht sein.

Listing 4.23
Break über mehrere Ebenen

```
<?PHP

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

   $square = $i * $i;
   printf("Das Quadrat von %2d ist %4d.\n", $i, $square) ;

   switch ( $i ) {
      case 5:
         $i++;
         break;
      case 7:
         break 2;
   }
}
?>
```

Damit wird die Zählervariable `$i` auf den Wert 6 erhöht. Da nach der `switch`-Anweisung keine weitere Anweisung folgt, wird der nächste Schleifendurchlauf gestartet. Dabei wird der Wert von `$i` wiederum um 1 erhöht, das normale Verhalten für diese Schleife. Damit ist der Wert der Zählervariablen bei 7 angelangt. Für diesen Fall gibt es ebenfalls eine Anweisung innerhalb der `switch`-Anweisung: `break 2`. Es werden also zwei Ebenen abgebrochen: die „`switch`-Ebene" und die direkt darüber liegende „`for`-Ebene". Damit springt der PHP-Interpreter hinter das Ende der `for`-Schleife. Da keine weiteren Anweisungen folgen, wird das Programm beendet. Die Ausgabe dieses Programms finden Sie in Listing 4.20.

```
Das Quadrat von  1 ist    1.
Das Quadrat von  2 ist    4.
Das Quadrat von  3 ist    9.
Das Quadrat von  4 ist   16.
Das Quadrat von  5 ist   25.
Das Quadrat von  7 ist   49.
```

Listing 4.24
Ergebnis der Berechnung aus Listing 4.19

Zur vollständigen Beschreibung der zählergesteuerten Schleife fehlt jetzt nur noch die Betrachtung einer Schleife ohne jegliche Steuerung! Dies ist ebenfalls möglich. In einem solchen Fall finden Sie im Schleifenkopf keine Angaben über die Steuerung der Schleife, es werden nur die beiden Semikola zur Trennung der Steueranweisungen angegeben.

In Listing 4.21 finden Sie diese Form der Schleife mit einer weiteren neuen Anweisung: `continue`. Bei dieser Art der Schleifensteuerung verlagern Sie das Verhalten der Schleife komplett in den übrigen Programmcode. Auf diese Art und Weise können Sie problemlos eine Endlosschleife programmieren, da die Anweisung `for (; ;)` für sich alleine genommen auf immer und ewig (so wie PHP dies versteht, also normalerweise 30 Sekunden) weiterläuft.

```
<?PHP

$i = 0;

for ( ; ; ) {

   $i++;

   if ( $i>10 )
      break;

   $square = $i * $i;
   printf("Das Quadrat von %2d ist %4d.\n", $i, $square) ;
```

Listing 4.25
Zählergesteuerte Schleife ohne Kopf und mit der Anweisung continue

Listing 4.25 (Forts.) Zählergesteuerte Schleife ohne Kopf und mit der Anweisung continue

```
    if ( $i == 5 ) {
    continue;
    }

    printf( "Anweisungen" );
    printf( " nach" );
    printf( " continue.\n");

}

?>
```

Die Anweisungen zur Steuerung der Schleife dürften mittlerweile hinlänglich klar sein. Schauen wir also auf die neue Anweisung `continue`. Diese Anweisung soll ausgeführt werden, wenn die Zählervariable den Wert 5 zugewiesen bekommen hat.

Hier sind die Anweisungen nach dem `continue` wichtig. Innerhalb der Schleife werden nach dem `continue` weitere Anweisungen ausgeführt. Im Beispiel sind dies nur drei Ausgabebefehle. Es können aber beliebig viele andere Anweisungen dort stehen.

Die Anweisung `continue` beendet die Schleife an der Stelle ihres Auftretens. Dabei wird aber nicht wie bei der `break`-Anweisung die komplette Schleife beendet, sondern nur der Teil der Schleife, der dem `continue` folgt, wird ausgelassen. Die Schleife wird mit dem nächsten Durchlauf weiter ausgeführt. Schauen Sie sich das Ergebnis aus Listing 4.21 in Listing 4.22 einmal an. Dort finden Sie einen Teil der Ausgabe um den Wert 5 herum.

Hat die Zählervariable den Wert 1 bis 4, werden die Anweisungen nach dem `continue` ausgeführt. Dies gilt auch für die Werte 6 bis 10. Bei dem Wert 5 fehlen diese Anweisungen.

Listing 4.26 Ausschnitt der Ausgabe von Listing 4.21

```
Das Quadrat von  4 ist   16.
Anweisungen nach continue.
Das Quadrat von  5 ist   25.
Das Quadrat von  6 ist   36.
Anweisungen nach continue.
Das Quadrat von  7 ist   49.
Anweisungen nach continue.
```

Sie können mit dieser Anweisung gezielt Teile einer Schleife von der Ausführung ausschließen, abhängig von den Bedingungen, die Sie hierfür festlegen.

Schauen wir uns zum Schluss dieses Kapitels über zählergesteuerte Schleifen ein etwas umfangreicheres Beispiel an. In diesem Beispiel soll festgestellt werden, ob es etwa geheimnisvolle Beziehungen zwischen einigen willkürlich gewählten Zahlen und Naturkonstanten gibt. Als eine mögliche Naturkonstante wird in diesem Beispiel die Kreiszahl PI (3,1415926...) genommen.

In der Natur erscheinen Beziehungen sehr häufig als Potenzgesetze. Solche Gesetze haben die Form y = a * xz. Dabei hängt die Größe y über ein Potenzgesetz von der Größe x ab. Es gibt aber auch Gesetze, die von mehreren Größen gleichzeitig abhängen. Nun gibt es ein ungeschriebenes Gesetz, wonach reale Abhängigkeiten umso unwahrscheinlicher werden, je mehr Größen man für diese Abhängigkeit einführt. Dieses Gesetz lassen wir in diesem Beispiel bewusst außen vor. Wir betrachten eine Funktion der Form:

Y = ai * bj * ck * dl * em

Den Größen a bis e ordnen wir beliebige Werte zu. Im Beispiel aus Listing 4.23 sind dies das Geburtsjahr des Autors, das Erscheinen von PHP, die Anzahl der Buchstaben von PHP, das Erscheinungsjahr dieses Buches und die Anzahl der Bücher des Autors. Also alles Größen, die einen großen Einfluss auf die Natur des Universums haben.

Um zu testen, ob es eine Beziehung zwischen diesen Größen und der Kreiszahl PI gibt, werden nun in fünf geschachtelten, zählergesteuerten Schleifen die Potenzen i bis m von –5 bis +5 berechnet und die Differenz zur Kreiszahl ermittelt. Diese Differenz entspricht einer Abweichung von einem Prozent. Es werden also nur dann die aktuellen Schleifenwerte ausgegeben, wenn die Abweichung von der Kreiszahl PI unter einem Prozent liegt. Anschließend wird dann die aktuelle Abweichung noch in eine Prozentangabe umgerechnet. Die Werte von i, j, k, l und m werden dann zusammen mit dieser Prozentangabe ausgegeben. Da nur der positive Wert der Abweichung interessiert, werden eventuell auftretende negative Werte vorher mit der Funktion `abs()` in positive Werte umgewandelt.

Listing 4.27
Berechnung von mystischen Zusammenhängen

```
<html>

<head>
  <title>Mystische Beziehungen</title>
</head>

<body>

<?php
```

Listing 4.27 (Forts.)
Berechnung von mystischen Zusammenhängen

```
$a = 1958; // Geburtsjahr des Autors
$b = 1995; // Geburtsjahr von PHP
$c = 3;    // PHP besteht aus drei Buchstaben
$d = 2005; // Erscheinungsjahr des Buches
$e = 2;    // 2. Buch des Autors

$natur_konstante = M_PI;

for ( $i = -5; $i <= 5; $i++ )
{
   for ( $j = -5; $j <= 5; $j++)
   {
      for ( $k = -5; $k <= 5; $k++ )
      {
         for ( $l = -5; $l <= 5; $l++)
         {
            for ( $m = -5; $m <= 5; $m++)
            {
               $res = pow($a,$i) * pow($b,$j) * pow($c,$k) *
                  pow($d,$l) * pow($e,$m);
               $diff = $res - $natur_konstante;

               if ( abs($diff) <= 0.01)
               {
                  print "Heureka: " . abs($diff) * 100.0 .
                     "% zu PI bei:<br>";
                  print "\$i = $i<br>";
                  print "\$j = $j<br>";
                  print "\$k = $k<br>";
                  print "\$l = $l<br>";
                  print "\$m = $m<br>";
               }
            }
         }
      }
   }
}

?>

</body>

</html>
```

In Abbildung 4.10 sehen Sie einen Ausschnitt dieser Berechnungen als Ausgabe auf einem Browser. Das PHP-Programm ist in eine HTML-Seite eingebettet, damit die Darstellung auf dem Browser erfolgen kann. Wie Sie sehen, sind die `\n` für den Zeilenumbruch auf der Kommandozeile einem `<br>` für den Zeilenumbruch in HTML gewichen.

Das zweite Ergebnis in dieser Abbildung liefert eine Abweichung von rund 0,1 Prozent von der Kreiszahl PI für die Formel:

Y = a-2 * c * d2

Was nichts anderes bedeutet wie

p [a] (Buchstabenanzahl von PHP) * (Erscheinungsjahr des Buches)2 / (Geburtsjahr des Autors)2

Sie sehen, Sie haben ein wahrlich besonderes Buch vor sich. Diese Art der Berechnung können Sie natürlich für jede andere Naturkonstante und Ihre Daten wiederholen und zu ähnlichen Ergebnissen kommen. Wie wäre es mit der Quadratmeterzahl Ihrer Wohnung?

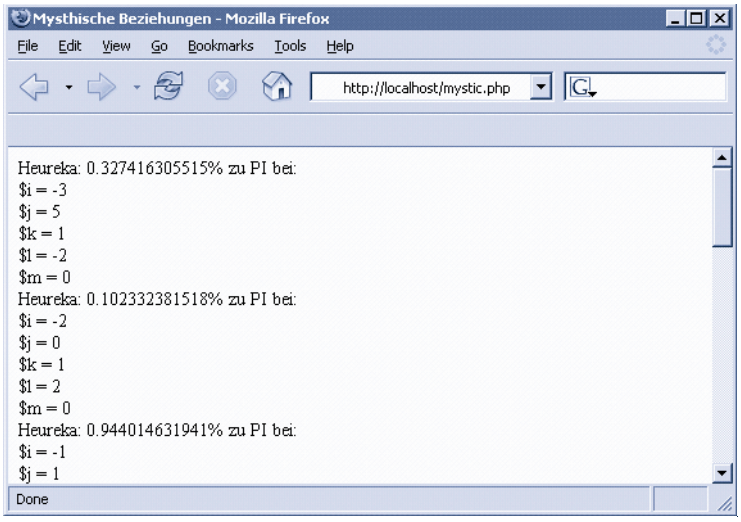


Abbildung 4.10
Ergebnisse der Berechnung aus Listing 4.23

4.2.2 Die Schleife mit Startbedingung

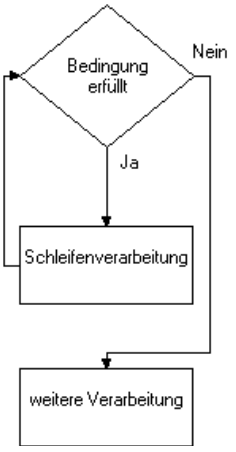
Neben der zählergesteuerten Schleife gibt es noch zwei Schleifenarten, die nicht über eine Zählervariable gesteuert werden, sondern nur über eine Bedingung. Diese Bedingung kann am Beginn oder am Ende der Schleife stehen. Im ersten Fall spricht man von einer kopfgesteuerten Schleife. Einen Programmablaufplan dieser Schleifenart sehen Sie in Abbildung 4.11. Die Umsetzung dieser Struktur in die Programmiersprache PHP können Sie dem Schema aus Listing 4.24 entnehmen. Sollte der Schleifenkörper nur aus einer Anweisung bestehen, können Sie die begrenzenden Klammern fortlassen.

Listing 4.28
Prinzipieller Aufbau einer while-Schleife

```
<?PHP

while ( Bedingung ) {
   Anweisungen;
}
?>
```

Als konkretes Beispielprogramm berechnen wir die Fibonacci-Zahlen. Es handelt sich um eine Zahlenfolge, die in der IT für die Berechnung bestimmter kryptografischer Schlüssel benutzt werden.

Listing 4.29
PAP einer kopfgesteuerten Schleife



Die Fibonacci-Zahlen sind definiert als

F(1) = 0,

F(2) = 1,

F(n) = F(n - 1) + F(n - 2)

Sie berechnen also die Fibonacci-Zahl von 3 F(3), indem Sie die Fibonacci-Zahl von 2 F(2) und die Fibonacci-Zahl von 1 F(1) addieren. Die Fibonacci-Zahlen von 0 und 1 sind festgelegt.

Das entsprechende Programm finden Sie in Listing 4.25. Um die Zahlen nur bis zu einer festgelegten Grenze zu berechnen, wird die Variable `$border` deklariert. Die ersten beiden Fibonacci-Zahlen werden mit `$f0` und `$f1` bezeichnet, die Folgezahl mit `$f2`.

Listing 4.30
Berechnung der Fibonacci-Zahlen mit einer while-Schleife

```
<?PHP
$border = 20;

$f0 = 0;
$f1 = 1;
$f2 = $f0 + $f1;
```

```
printf( "Fibonacci-Zahlen bis %3d: %3d,%3d", $border, $f0, $f1 );

while ( $f2 <= $border ) {
   printf( ",%3d", $f2 );

   $f0 = $f1;
   $f1 = $f2;
   $f2 = $f0 + $f1;

}

?>
```

Listing 4.30 (Forts.)
Berechnung der Fibonacci-Zahlen mit einer while-Schleife

Nachdem die ersten beiden Fibonacci-Zahlen ausgegeben wurden, wird in die Schleife gesprungen. Dies kann nicht in der Schleife geschehen, da diese Werte fest vorgegeben sind. Die Schleife wird so lange ausgeführt, wie die Variable `$f2` kleiner oder gleich der Grenzvariable `$border` ist.

Innerhalb der Schleife werden nach der Ausgabe von `$f2` die Variablen `$f0` und `$f1` wiederverwendet. Der ursprüngliche Wert von `$f0` wird verworfen und mit dem Wert aus `$f1` besetzt. Analog wird mit der Variablen `$f1` verfahren, und die Variable `$f2` wird mit diesen Werten neu berechnet.

Man hätte an dieser Stelle auch neue Variablen einsetzen können, diese beiden Variablen werden aber in der Folge des Programms mit ihrer ursprünglichen Bedeutung nicht mehr gebraucht. Diese Art der Umdefinition von Variablen sollten Sie allerdings in einem überschaubaren Rahmen halten. Damit ist der Umfang von Programmzeilen gemeint. Wenn Sie solche Umdefinitionen nach Dutzenden von Zeilen durchführen, kann dies sehr schnell zu nicht mehr interpretierbaren Programmen führen: „Welche Bedeutung hat die Variable denn jetzt an dieser Stelle??“

Die Ausgabe dieses Programms sehen Sie in Listing 4.26.

```
Fibonacci-Zahlen bis  20:   0,  1,  1,  2,  3,  5,  8, 13
```

Listing 4.31
Fibonacci-Zahlen mit dem Programm aus Listing 4.25

Diese Art der kopfgesteuerten Schleife können Sie auch mit einer zählergesteuerten Schleife nachbilden, indem Sie die Start- und Folgebedingung im Kopf der `for`-Anweisung nicht angeben. Sie bekommen die Formulierung:

```
for ( ; $f2 <= $border; ) {
```

Wie bei der zählergesteuerten Schleife können Sie innerhalb des Schleifenkörpers die Anweisungen `break` und `continue` einsetzen. In Listing 4.27 wird die `break`-Anweisung dazu genutzt, das Programm bei Überschreiten des Grenzwertes zu beenden. Hier wird das `break` nicht zum vorzeitigen Beenden der Schleife genutzt, sondern dient dem regulären Beenden der Schleifendurchläufe.

Listing 4.32 Endlosschleife mit Abbruchbedingung

```
<?PHP

$border = 20;

$f0 = 0;
$f1 = 1;
$f2 = $f0;

printf( "Fibonacci-Zahlen bis %3d: %3d,%3d", $border, $f0, $f1 );

while ( true ) {
   $f2 = $f0 + $f1;

   if ( $f2 > $border )
      break;

   printf( ",%3d", $f2 );

   $f0 = $f1;
   $f1 = $f2;

}

?>
```

Der Schleifenkopf in Listing 4.27 leitet eine Endlosschleife ein, da die Bedingung für die Fortsetzung der Schleife immer `true` ist. Sie können eine solche Schleifenkonstruktion verwenden, wenn das Abbruchkriterium für die Schleife innerhalb der Schleife berechnet wird. Da Sie in diesem Fall beim Start der Schleife noch nicht wissen, ob ein Abbruchkriterium erreicht wird, müssen Sie in einem solchen Fall einen Zwangsausstieg einbauen, ansonsten wird Ihr gesamtes Programm standardmäßig nach 30 Sekunden durch den PHP-Interpreter abgebrochen.

4.2.3 Die Schleife mit Schlussbedingung

Bei der Schleife mit Schlussbedingung, auch fußgesteuerte Schleife genannt, wird die Bedingung für das weitere Durchlaufen der Schleife am Ende des Schleifenkörpers getestet. Den entsprechenden Programmablaufplan sehen Sie in Abbildung 4.12. Die Umsetzung nach PHP

geschieht nach dem Schema aus Listing 4.28. Eingeleitet wird die fußgesteuerte Schleife mit dem Schlüsselwort `do`. Der folgende Schleifenkörper wird durch geschweifte Klammern begrenzt, denen die Bedingung mit dem Schlüsselwort `while` folgt.

Zwischen der kopf- und der fußgesteuerten Schleife gibt es nur einen prinzipiellen Unterschied. Während es bei der kopfgesteuerten Schleife aufgrund des Eingangstests dazu kommen kann, dass die Schleife nicht abgearbeitet wird, wird die fußgesteuerte Schleife mindestens einmal durchlaufen. Dies liegt einfach daran, dass erst der Schleifenkörper abgearbeitet und danach die Bedingung für einen erneuten Durchlauf getestet wird.

Listing 4.33
Prinzipieller Aufbau einer do-Schleife

```
<?PHP

do {
   Anweisungen;
} while ( Bedingung );

?>
```

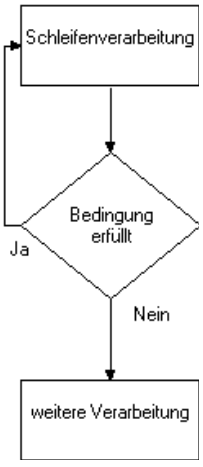


Abbildung 4.11
PAP einer fußgesteuerten Schleife

Ein konkretes Beispiel für diese Art einer Schleife finden Sie in Listing 4.29. In diesem Programm werden per Zufall einige Zeichen aus dem ASCII-Zeichensatz berechnet und ausgegeben. Man könnte diese Zeichen recht gut als Vorschlag für Kennwörter nehmen.

Listing 4.34
Berechnung von Zufallsbuchstaben mit do-while

```
<?PHP

$counter = 0;

$seed = microtime() * 1000000;
mt_srand( $seed );
```

Listing 4.34 (Forts.) Berechnung von Zufallsbuchstaben mit do-while

```
printf( "Zufallszeichen: < ");

do {
   $character = mt_rand( 48, 126 );

   printf( "%s", chr($character ) );
   $counter++;
} while ( $counter < 10 );

printf( " >\n" );

printf( "Wert von \$counter = %d\n", $counter );

?>
```

Zu Beginn des Programms wird die Variable `$counter` mit Null deklariert. Diese Variable wird im Schleifenkörper bei jedem Durchlauf um 1 hochgezählt.

Die beiden `printf()`-Anweisungen dienen der Erklärung der Funktion `microtime()`, da diese Funktion im Zusammenhang mit dem Zufallszahlengenerator sehr häufig eingesetzt wird. Diese Funktion berechnet die Zeit seit dem 01.01.1970 abhängig von dem Parameter, den Sie dieser Funktion übergeben. Übergeben Sie dieser Funktion keinen Parameter oder der Parameter wird als `false` interpretiert, so gibt Ihnen diese Funktion eine Zeichenkette zurück, in der zwei Zahlen stehen. An erster Stelle finden Sie die Anzahl der Mikrosekunden seit dem letzten Sekundenwechsel. Diese Zahl ist auf den Wertebereich von 0 bis 1 skaliert. Wenn Sie die Anzahl der Mikrosekunden benötigen, müssen Sie diesen Wert mit 1.000.000 multiplizieren. Der zweite Teil der Zeichenkette ist nun die Anzahl der Sekunden nach dem 01.01.1970.

Übergeben Sie der Funktion `microtime()` einen Parameter, der als `true` interpretiert werden kann, bekommen Sie die Anzahl der Sekunden seit dem 01.01.1970 als Fließkommazahl zurückgeliefert.

Die folgenden zwei Zeilen dienen der Initialisierung des Zufallszahlengenerators. Diese Initialisierung benötigen Sie ab der PHP-Version 4.2.0 eigentlich nicht mehr. Da Sie aber bei fast allen Programmen, die mit Zufallszahlen arbeiten, darauf stoßen werden, sind sie hier mit aufgeführt.

Es wird zuerst der Initialisierungswert für den Zufallszahlengenerator berechnet. Dieser wird in der Programmierung, also nicht nur in PHP, als „seed“, „Samen“ bezeichnet. Bei gleichem Wert von `$seed` bekommen Sie die exakt gleiche Abfolge von Zufallszahlen. Man spricht daher

auch von Pseudo-Zufallszahlen. Um dem Zufall also möglichst nahe zu kommen, muss die Variable `$seed` ebenfalls zufällig sein; ein Münchhausen-Effekt. Sie kommen dem Zufall aber schon recht nahe, wenn Sie den Wert der Mikrosekunden als Initialisierungswert nehmen. Die Wahrscheinlichkeit, denselben Wert für `$seed` zu bekommen, liegt bei 1 : 1000000.

Die Funktion `mt_srand()` initialisiert nun den Zufallszahlengenerator.

In der Schleife wird mit der Funktion `mt_rand()` ein Zufallswert zwischen 48 und 126 berechnet. Dieser Zahlenbereich entspricht den druckbaren Zeichen im ASCII-Zeichencode. Diese Zahlen werden dann in der `printf()`-Anweisung mit der Funktion `chr()` in eine Zeichenkette umgewandelt. Beispielsweise ergibt `chr(65)` das Zeichen „`A`“.

Die Ausgabe des Programms aus Listing 4.29 sehen Sie beispielhaft in Listing 4.30. An den Zufallszeichen zwischen „<“ und „>“ sehen Sie den potenziellen Nutzen als Kennwort.

In der letzten Zeile erkennen Sie, dass der Wert der Variablen `$counter` nicht automatisch zurückgesetzt wird, wie manchmal vermutet wird. Diese Variable behält den Wert, den sie zum Zeitpunkt der Beendigung der Schleife hatte.

Listing 4.35
Ausgabe von Listing 4.29

```
Aufruf microtime(): 0.55245200 1095920166

Aufruf microtime(true): 1095920166.5528

Zufallszeichen: < c~S<uYyhrM >
Wert von $counter = 10
```

Die Funktionen `mt_srand()` und `mt_rand()` sind Erweiterungen von PHP, die standardmäßig in PHP eingebunden sind. Die eigentlichen PHP-Funktionen für die Ermittlung von Zufallszahlen sind `srand()` und `rand()`. Diese Funktionen werden weithin als nicht so zufällig und recht langsam klassifiziert. Daher findet man im Allgemeinen nur die hier benutzten Funktionen.

4.3 Funktionen

Im Verlaufe der vergangenen Kapitel haben Sie schon verschiedene Funktionen kennen gelernt, die Ihnen von PHP angeboten werden. Sie werden festgestellt haben, dass Funktionen immer eine bestimmte Aufgabe ausführen, dass aber mit hoher Zuverlässigkeit. Vielleicht ist Ihnen bei dem einen oder anderen Beispiellisting der Gedanke gekommen,

dass man diese Funktionalität als Funktion ganz gut gebrauchen könnte. Übrigens kommt daher auch der Name.

Eigene Funktionen zu erstellen, ist kein großes Geheimnis. Schauen wir uns die Möglichkeiten an, in PHP eine benutzerdefinierte Funktion zu erstellen und zu nutzen.

4.3.1 Funktionsdefinition

Es soll hier um zwei Arten der Definition gehen: Was ist eigentlich eine Funktion und wie definiere ich eine Funktion in PHP?

Um eine benutzerdefinierte Funktion zu schreiben, müssen Sie erst einmal erkennen, welche Programmteile sich als potenzielle Funktion eignen. Diese Feststellung können Sie recht einfach treffen: Immer dann, wenn Sie selbst geschriebenen Code in identischer Form wiederholt einsetzen, sollten Sie diesen Codeabschnitt in eine Funktion kapseln. Mit „wiederholt“ kann hier durchaus „mehr als einmal“ gemeint sein, also schon beim zweiten Mal! Je häufiger Sie einen Codeabschnitt benötigen, je besser ist dieser Abschnitt für eine Funktion geeignet. Der Vorteil liegt nicht nur im Ersparnis des Tippens, wofür hat man „Cut and Paste“? Wenn Sie aber an diesem Codeteil etwas ändern müssen, dürfen Sie das an x Stellen in Ihrem Programm durchführen. Dabei vergessen Sie wahrscheinlich eine Stelle oder vertippen sich. Wenn Sie die Änderung an einer zentralen Stelle durchführen, haben Sie dies direkt für Ihr gesamtes Programm durchgeführt!

Eine benutzerdefinierte Funktion wird immer nach dem Schema aus Listing 4.31 aufgebaut. Die Funktion wird mit dem Schlüsselwort `function` eingeleitet. Diesem Schlüsselwort folgt der Name der Funktion. Dieser Name kann von Ihnen frei vergeben werden. Er muss dabei den Regeln für Variablennamen entsprechen. In Klammern folgt dann eine Reihe von Argumenten (den Übergabeparametern), die Sie dieser Funktion übergeben wollen. Diese Liste kann auch leer sein, die beiden Klammern müssen aber auf jeden Fall angegeben werden. Der Funktionskörper wird durch geschweifte Klammern begrenzt. In diesem Funktionskörper befinden sich die Anweisungen, die für das Verhalten der Funktion zuständig sind.

Listing 4.36 Schema einer benutzerdefinierten Funktion

```
function FUNKTIONS_NAME ( ARGUMENT_1, ARGUMENT_2, ... ) {
   Anweisung_1;
   Anweisung_2;
   ...
}
```

4.3.2 Funktionen ohne Argumente

Eine sehr einfache Funktion hat keine Übergabeparameter und eine begrenzte Funktionalität, wie Sie sie in Listing 4.32 finden. Die Funktion trägt den Namen `hello` und hat keine Übergabeparameter. Diese Funktion macht nichts anderes, als die Zeichenkette „Hallo Welt“ mit einem Zeilenumbruch auszugeben. Nach der Definition der Funktion beginnt das eigentliche Programm, auch Hauptprogramm genannt.

Listing 4.37
Einfache Funktion zur Ausgabe einer festen Zeichenkette

```
<?PHP

function hello() {
   printf( "Hallo Welt.\n" );
}

printf( "Start des Hauptprogramms.\n" );
hello();
?>
```

Nach der Ausgabe der Zeichenkette „Start des Hauptprogramms.“ wird die neue Funktion aufgerufen. Dies geschieht exakt so, wie Sie es von den PHP-eigenen Funktionen gewohnt sind. Die Ausgabe dieses Programms finden Sie in Listing 4.337.

Listing 4.38
Ausgabe des Programms aus Listing 4.32

```
Start des Hauptprogramms.
Hallo Welt.
```

Eine Funktion kann aber nicht nur direkt etwas ausgeben, sie kann auch berechnete Werte an das Hauptprogramm zurückliefern. PHP hat leider keine Funktion, die das heutige Datum in deutscher Schreibweise direkt ausgibt. Also eine Möglichkeit, daraus eine benutzerdefinierte Funktion zu erstellen.

Aus nahe liegenden Gründen lautet der Name dieser Funktion `today()`. Diese Funktion benötigt keine Übergabeparameter, da man das heutige Datum aus der Uhr des Computers beziehen kann. Dies erledigt die PHP-Funktion `date()`, die standardmäßig allerdings eine vollkommen andere Ausgabe produziert. Dieser Funktion `date()` können Sie eine Zeichenkette übergeben, die der Formatierung der Datumsangabe dient. Der Aufbau dieser Zeichenkette kann sehr umfangreich sein. Sie sollten dafür einen Blick in das Handbuch werfen. Die Funktion gibt das entsprechende Datum als Zeichenkette zurück.

Für unsere Zwecke ist die Zeichenkette für die Formatierung recht einfach. Sie liefert eine Datumsangabe im Format TT.MM.JJJJ, also beispielsweise 01.01.2005. Diese Zeichenkette wird in der Variablen `$date2day` abgelegt.

Listing 4.39 Funktion zur Ermittlung des heutigen Datums in deutscher Schreibweise

```
<?PHP

function today() {

   $date22day = date("d.m.Y");

   return $date22day;

}

printf( "Start des Hauptprogramms.\n" );
printf( "Heute ist der %s.\n", today() );

?>
```

In diesem Variablennamen steckt ein beliebter Trick, Namen zu verkürzen. Im amerikanischen sind die Sprechweisen der Zahl 2 und des Wortes „to“ identisch. Gleiches gilt auch für andere Zahlen. Der Variablenname könnte in ausgeschriebener Form auch `$date_to_today` lauten.

Diese Variable wird mit der Anweisung `return` an das aufrufende Programm zurückgegeben. Wenn Sie das Programm aus Listing 4.34 laufen lassen, bekommen Sie eine Ausgabe ähnlich der in Listing 4.35.

Listing 4.40 Ausgabe des Programms aus Listing 4.34

```
Start des Hauptprogramms.
Heute ist der 21.07.2004.
```

4.3.3 Funktionen mit Argumenten

Die bisher erstellten benutzerdefinierten Funktionen waren recht unabhängig vom Umfeld, in dem sie aufgerufen wurden. Dies ist nicht immer so. Nehmen Sie als Beispiel die Erzeugung eines Kennwortes. In einem Fall soll dieses Kennwort vier Zeichen lang sein, in einem anderen Fall aber acht Zeichen. Die Erzeugung des Kennwortes ist bis auf die Länge identisch. In einer solchen Situation können Sie eine Funktion schreiben, die Ihnen ein neues Kennwort erstellt und der Sie die Länge des Kennwortes übergeben. Eine mögliche Lösung finden Sie in Listing 4.36. Sie basiert auf dem Programm aus Listing 4.29.

Listing 4.41 Funktion get_password()

```
<?PHP

function get_password( $length ) {
   $counter = 0;
   $password = "";

   do {
      $character = mt_rand( 48, 126 );
```

```
      $password .= chr($character );
      $counter++;

   } while ( $counter < $length );

   return $password;
}

printf( "Erzeugtes Kennwort= %s\n", get_password( 8 ) );

?>
```

Listing 4.41 (Forts.)
Funktion get_password()

Die Funktion `get_password()` bekommt die gewünschte Länge des Kennwortes als Argument übergeben. Dieses Argument dient im Funktionskörper als Abbruchbedingung für die fußgesteuerte Schleife. Das eigentliche Programm besteht nur aus der `printf()`-Anweisung am Ende von Listing 4.36. Innerhalb der `printf()`-Funktion wird die benutzerdefinierte Funktion mit der Zahl 8 als Argument aufgerufen. In Listing 4.37 sehen Sie eine mögliche Ausgabe dieses Programms.

```
Erzeugtes Kennwort= ~lfvJI4i
```

Listing 4.42
Ausgabe des Programms aus Listing 4.36

In der Funktionsdefinition spricht man bei der Variablen `$length` auch von einem formalen Parameter. Ein formaler Parameter wird beim Programmablauf durch den aktuellen Parameter ersetzt. In Listing 4.36 ist dieser aktuelle Parameter die Zahl 8 beim Aufruf der Funktion. Beim Aufruf einer Funktion können drei Ereignisse eintreten:

1. Die Anzahl der formalen und der aktuellen Parameter stimmt überein.
2. Die Anzahl der formalen Parameter ist kleiner als die Anzahl der aktuellen Parameter.
3. Die Anzahl der formalen Parameter ist größer als die Anzahl der aktuellen Parameter.

Im ersten Fall ist alles okay, und die Funktion verhält sich, wie Sie es bisher kennen gelernt haben.

Im zweiten Fall werden die zusätzlichen aktuellen Parameter, die in der Folge von links nach rechts hinter dem letzten formalen Parameter stehen, einfach ignoriert. Wenn Sie beispielsweise die Funktion aus Listing 4.36 mit

```
get_password( 4, "hallo" )
```

Listing 4.43
Funktionsaufruf von get_password() mit zu vielen Argumenten

aufrufen, wird die Zeichenkette „hallo" ignoriert, und die Funktion liefert Ihnen ein Kennwort mit einer Länge von vier Zeichen.

Im dritten Fall bekommen Sie vom PHP-Interpreter eine Warnung, die Funktion wird aber trotzdem ausgeführt. Alle formalen Parameter werden dann so behandelt, als wären sie nicht deklariert worden. Das heißt, Zeichenketten werden als leere Zeichenketten, Zahlen als 0 betrachtet!

Wenn Sie also die Funktion aus Listing 4.36 ohne einen Übergabeparameter aufrufen:

Listing 4.44 Funktionsaufruf von get_password() mit zu wenigen Argumenten

```
get_password()
```

erhalten Sie eine Warnung, wie Sie in Abbildung 4.13 gezeigt ist. Sie sehen auch, dass ein Kennwort aus nur einem Buchstaben zurückgeliefert wird. Sollte die Länge nicht 0 betragen, also eine leere Zeichenkette? Ja, aber in der Funktion wird eine fußgesteuerte Schleife benutzt. Bevor also festgestellt wird, dass die Länge null betragen soll, wird ein Buchstabe generiert und zurückgeliefert.

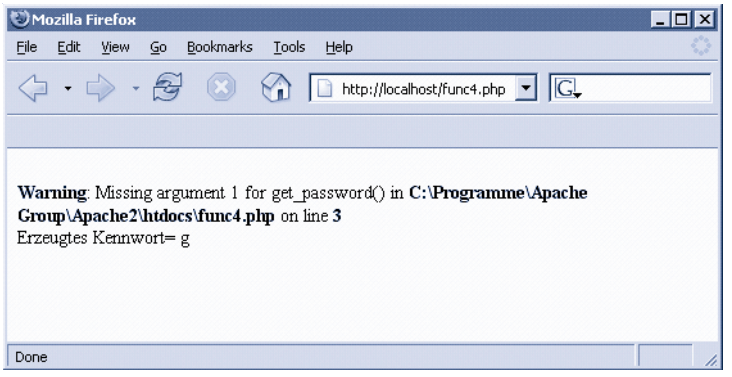
Um solche Fehler bei zu wenigen Übergabeparametern zu vermeiden, kann man in PHP im Funktionskopf für jedes Argument einen Standardwert angeben. Außerdem ersparen Sie sich Tipparbeit, wenn in den meisten Fällen ein bestimmter Wert gewünscht ist.

Um die Länge des Kennwortes standardmäßig auf acht Zeichen zu setzen, müssen Sie den Kopf der Funktion in

```
function get_password( $length = 8 ) {
```

ändern. Wenn Sie diese Funktion dann wie in Listing 4.39 aufrufen, wird für das formale Argument `$length` der Wert 8 eingesetzt und mit diesem Wert die Funktion ausgeführt.

Abbildung 4.12 Funktionsaufruf mit zu wenigen Argumenten



Das Verhalten des PHP-Interpreters bei Fehlern oder Warnungen können Sie übrigens mit der Funktion `error_reporting()` beeinflussen.

Sollten Sie mehrere Argumente an eine Funktion übergeben wollen, so müssen Sie die formalen Parameter durch ein Komma voneinander trennen.

Dies können Sie anhand der Funktion `check()` aus Listing 4.40 sehen. Dieser Funktion werden zwei Argumente übergeben, die im Funktionskörper auf Identität getestet werden. Diese Funktion wird im Anschluss an die Deklaration dreimal mit unterschiedlichen aktuellen Parametern aufgerufen.

Listing 4.45
Die benutzerdefinierte Funktion check() zum Test auf Identität

```
<?PHP

function check( $val1, $val2 ) {

   if ( $val1 === $val2 )
      return true;
   else
      return false;

}

$result = check( "Hallo", "hallo");
printf( "Hallo === hallo : %u\n", $result );

$result = check( 15, "15" );
printf( "\"15\" === 15     : %u\n", $result );

$result = check( 15, 15 );
printf( "15 === 15       : %u\n", $result );

?>
```

Wie Sie sicher noch wissen, wird die 0 als `false` und die 1 als `true` interpretiert. Was Sie an diesem Beispiel auch noch einmal sehen, ist, dass die Zeichenkette „15“ nicht mit der Zahl 15 identisch ist. Sie werden nur beim Gleichheitsoperator `==` als „gleich“ erkannt.

Listing 4.46
Ausgabe des Programms aus Listing 4.40

```
Hallo === hallo : 0
"15" === 15     : 0
15 === 15       : 1
```

4.3.4 Gültigkeitsbereich von Variablen

Bisher haben wir immer nur verschiedene Variablennamen benutzt. Was passiert aber, wenn innerhalb einer Funktion eine Variable benutzt wird, die es außerhalb auch schon gibt? Dies kann bei längeren Programmen ja durchaus passieren. Zumal wenn wir die Funktion aus dem Quelltext ausgelagert haben (als externes Programm).

Allgemein gilt, die Funktion kennt die Variablen außerhalb der Funktion nicht, und das Programm kennt die Variablen innerhalb der Funktion nicht.

In Listing 4.42 können Sie dieses Verhalten nachvollziehen. Die Variable `$counter` existiert sowohl im Hauptprogramm als auch innerhalb der Funktion. Weiterhin gibt es eine Variable `$string` im Hauptprogramm, die innerhalb der Funktion ausgegeben wird. Zudem wird versucht, die Variable `$password` aus der Funktion im Hauptprogramm auszugeben.

Listing 4.47 Gültigkeitsbereich von Variablennamen

```
<?PHP

function get_password( $length = 8 ) {
   printf("Eintritt in die Funktion.\n");

   $counter = 0;
   $password = "";

   do {
      $character = mt_rand( 48, 126 );

      $password .= chr($character );
      $counter++;

   } while ( $counter < $length );

   printf("Aktueller Wert von \$counter: %d\n", $counter );
   printf("Aktueller Wert von \$string : %s\n", $string );

   printf( "Austritt aus der Funktion.\n" );
   return $password;
}

printf( "Start des Hauptprogramms.\n");
$counter = 100;
$string = "Test";

printf( "Erzeugtes Kennwort : %s\n", get_password() );
printf( "Wert von \$counter : %d\n", $counter );
printf( "Wert von \$password: %s\n", $password );

?>
```

Die Variable `$counter` wird zu Beginn des Programms auf den Wert 100 gesetzt. Mit diesem Wert wird die Funktion aufgerufen, die eine Variable mit dem gleichen Namen erst einmal auf 0 setzt und im Laufe des Programms hochzählt. Wie Sie an der Ausgabe von Listing 4.43 erkennen,

hat diese Variable am Ende der Funktion den Wert 8. Sobald Sie sich aber wieder im Hauptprogramm befinden, erscheint der alte Wert 100.

Die Variable `$string` aus dem Hauptprogramm hat innerhalb der Funktion eine leere Zeichenkette als Inhalt. Sie wird innerhalb der Funktion als nichtdeklarierte Variable interpretiert und bekommt daher diesen Wert. Analoges gilt für die Variable `$password` im Hauptprogramm.

Listing 4.48
Ausgabe des Programms aus Listing 4.42

```
Start des Hauptprogramms.
Eintritt in die Funktion.
Aktueller Wert von $counter: 8
Aktueller Wert von $string :
Austritt aus der Funktion.
Erzeugtes Kennwort : IhPQ??uW
Wert von $counter : 100
Wert von $password:
```

Will man eine Variable aus dem Hauptprogramm innerhalb einer Funktion benutzen, kann man diese Variable als „global“ definieren. Dies geschieht innerhalb der Funktion mit dem Schlüsselwort `global`.

In Listing 4.44 wird die Variable `$counter` im Hauptprogramm deklariert. Innerhalb der Funktion `global_test()` wird diese Variable aus dem Hauptprogramm mit der Anweisung `global $counter` übernommen.

Listing 4.49
Globale Variable innerhalb einer Funktion

```
<?PHP

function global_test() {
   global $counter;

   $counter *= 2;

}

$counter = 100;

global_test();

printf( "Wert von \$counter: %d\n", $counter );

?>
```

Der Einfachheit halber wird diese Variable verdoppelt. Sie bemerken sicherlich, dass die Variable nicht mit einem `return` zurückgegeben wird.

Im Hauptprogramm wird diese Funktion aufgerufen und der Wert von `$counter` mit der folgenden `printf()`-Anweisung ausgegeben. Wie vermutet hat die Variable an dieser Stelle den Wert `200`.

Dies können Sie natürlich mit beliebig vielen Variablen machen. Aber Achtung! Solche Konstruktionen bergen die Gefahr in sich, dass Sie Werte an Stellen im Programm verändern, an denen Sie dies überhaupt nicht gebrauchen können. Sie sollten sich den Einsatz von `global` sehr gründlich überlegen.

4.3.5 Statische Variable

Die Variablen, die wir bisher in den Funktionen benutzt haben, werden bei jedem Aufruf der Funktion neu gesetzt. Bei jedem Aufruf der Funktion `get_password()` aus Listing 4.36 wird die Variable `$counter` mit `0` und die Variable `$password` mit einer leeren Zeichenkette initialisiert.

Es gibt Situationen, wo Sie eine Funktion gut gebrauchen können, die zwischen den einzelnen Aufrufen den Wert einer oder mehreren Variablen behält. Solche Variablen werden (nicht nur) in PHP als statische Variablen bezeichnet.

Diese Variablen deklarieren Sie mit dem Schlüsselwort `static`. Dabei können Sie auch den Startwert angeben, mit dem die Variable bei der Deklaration initialisiert werden soll.

In Listing 4.45 sehen Sie die Anwendung einer solchen statischen Variablen in der Funktion `increment_counter()`. Die Variable `$counter` wird als `static` deklariert und mit dem Wert 10 vorbelegt. In der Funktion wird diese Variable jeweils um den Wert 1 hochgezählt und dem aufrufenden Programm zurückgegeben.

Listing 4.50 Einsatz einer statischen Variablen als Zähler

```
<?PHP

function increment_counter() {

   static $counter = 10;

   ++$counter;

   return $counter;

}

printf( "Wert von \$counter: %d\n", increment_counter() );
printf( "Wert von \$counter: %d\n", increment_counter() );
printf( "Wert von \$counter: %d\n", increment_counter() );
printf( "Wert von \$counter: %d\n", increment_counter() );

?>
```

Im Hauptprogramm wird diese Funktion viermal aufgerufen und der aktuelle Wert ausgegeben. Dies sehen Sie in Listing 4.46. Wenn Sie das Schlüsselwort `static` fortlassen sollten, bekommen Sie bei jedem Aufruf den Wert 11 zurückgeliefert.

Sie können das Inkrementieren des Zählers als Abkürzung auch in die `return`-Anweisung übernehmen. Dabei haben Sie zwei Möglichkeiten. Als Prä-Inkrement

```
return ++$counter;
```

oder als Post-Inkrement

```
return $counter++;
```

Die Ausgabe wird allerdings dann um den Wert 1 differieren. Während die Zählung beim Prä-Inkrement-Operator bei 11 beginnt, startet die Zählung beim Post-Inkrement-Operator bei 10. Der Grund hierfür dürfte Ihnen sicherlich klar sein, oder?

```
Wert von $counter: 11
Wert von $counter: 12
Wert von $counter: 13
Wert von $counter: 14
```

Listing 4.51
Ausgabe des Programms aus Listing 4.45

4.3.6 Referenzen

Bisher haben Sie Funktionen geschrieben, die zwar mehrere Argumente übernehmen konnten, aber immer nur maximal einen Wert zurückliefern konnten. Nun gibt es sicherlich Funktionen, bei denen Sie nicht nur ein Ergebnis zurückgeliefert bekommen möchten, sondern vielleicht auch, ob die Funktion überhaupt ordnungsgemäß durchlaufen wurde. Oder die Funktion soll mehrere Ergebnisse zurückliefern.

Dann schlägt die Stunde der Referenzen. Sie können Referenzen an jeder beliebigen Stelle Ihres Programms einsetzen. Aber im Zusammenhang mit Funktionen werden sie häufiger eingesetzt. Daher sollen sie auch hier behandelt werden.

Eine Referenz kann man sich wie einen Zeiger vorstellen. Sollten Sie aus der C/C++-Welt kommen: Es handelt sich *nicht* um einen klassischen Zeiger im C/C++-Sinn. Eine Referenz bezieht sich einzig und allein auf den Wert. Wenn Sie sich an die Grundlagen der Variablen von PHP erinnern, besteht eine Variable in PHP aus wesentlich mehr als nur aus ihrem Wert. Es ist aber trotzdem recht hilfreich, sich Referenzen als Zeiger vorzustellen.

In Listing 4.47 wird eine Variable `$value` mit dem Wert 5 deklariert. In der nächsten Zeile wird die Variable `$ref_value` als Referenz auf diese Variable deklariert. Wenn Sie genau auf die Zuweisung achten, fällt Ihnen sicherlich auf, dass vor dem Variablennamen das `&`-Zeichen erscheint. Mit diesem Zeichen sagen Sie dem PHP-Interpreter, dass Sie einen Verweis auf den Wert dieser Variablen haben möchten.

Listing 4.52
Referenz auf eine Variable

```
<?PHP

$value = 5;

$ref_value = &$value;

$ref_value += 3;

printf( "\$value hat jetzt den Wert: %d\n", $value );

?>
```

Wenn Sie dieses Programm ausführen, wird nach dieser Zuweisung der Wert von `$ref_value` und damit eigentlich der Wert von `$value` um drei erhöht. Die Ausgabe dieses kleinen Programms lautet also folgerichtig:

Listing 4.53
Ausgabe des Programms aus Listing 4.47

```
$value hat jetzt den Wert: 8
```

In Listing 4.49 sehen Sie die Umsetzung von Referenzwerten innerhalb eines Funktionsaufrufes. Die Funktion `division()` liefert als direkten Funktionswert `true` zurück, wenn die Funktion die Division durchführen konnte. Sollte die Division nicht möglich sein, wird `false` zurückgegeben. Bei einer Division ist dies typischerweise bei einer Division durch null der Fall. Eine solche Division ist nicht definiert. Sollte eine solche Division vom PHP-Interpreter festgestellt werden, wird das gesamte Programm abgebrochen.

Sollten Sie die Vermutung haben, dass eine solche Division in Ihrem Programm möglich ist, sollten Sie diesen Fall auffangen. Eine mögliche klassische Methode stellt diese Funktion `division()` dar. Ab Version 5 von PHP gibt es aber noch ein anderes Verfahren, mit dem in solchen Situationen vorgegangen werden kann, die Exceptions.

Bleiben wir erst einmal bei der klassischen Vorgehensweise der Funktion `division()`. Der Funktion werden drei Parameter übergeben, der Zähler (`$numerator`), der Nenner (`$denominator`) und die Referenz auf das Ergebnis der Division `&$result`.

Bevor Sie diese Funktion aufrufen können, müssen Sie eine Variable deklarieren, deren Referenz Sie der Funktion übergeben können.

Ist die Variable $denominator ungleich null, wird das Ergebnis der Division der Variablen $result zugeordnet. An dieser Stelle benötigen Sie das Referenzzeichen & nicht. Sie können diese Referenzvariable innerhalb der Funktion wie eine normale Variable behandeln. In dem Moment, in dem das Ergebnis der Division der Variablen zugeordnet wird, ändert sich der Wert der Variablen $division_result aus dem Hauptprogramm.

Listing 4.54
Referenzvariable als Funktionsargument bei der Funktion division()

```
<?PHP

function division( $numerator, $denominator, &$result ) {

   if ( $denominator == 0 )
      return false;

   $result = $numerator / $denominator;

   return true;

}

$division_result = 0;

if ( division( 5, 2, &$division_result ) )
   printf("Divisionsergebnis = %f\n", $division_result );
else
   printf( "Division durch null!\n" );

if ( division( 5, 0, &$division_result ) )
   printf("Divisionsergebnis = %f\n", $division_result );
else
   printf( "Division durch null!\n" );


?>
```

Da diese Funktion einen logischen Wert zurückliefert, können Sie den Aufruf der Funktion direkt in die Bedingung der if-Abfrage einsetzen. Die Ergebnisse der beiden Funktionsaufrufe sind in Listing 4.50 zu sehen.

Listing 4.55
Ausgabe des Programms aus Listing 4.49

```
Divisionsergebnis = 2.500000
Division durch null!
```

Ähnlich wie jede andere Variable können Sie auch Referenzvariablen löschen. Hierfür wird die Funktion unset() benutzt. Bedenken Sie dabei, dass zwischen Wert und Name einer Variablen unterschieden werden kann. In der Variablen wird ein Referenzzähler gepflegt, der mitzählt,

wie viele Variablen eine Referenz auf den Wert halten. Wenn zwei Variablen auf diesen Wert verweisen (Original und die erste Referenz), kann das Original ruhig gelöscht werden. Dabei wird der Referenzzähler um eins verringert. Die Referenz kann dann immer noch auf den Wert zugreifen. Dieses Verhalten entspricht zum Beispiel überhaupt nicht dem Verhalten von C/C++-Zeigern.

Im Listing 4.51 wird die Originalvariable gelöscht, während die Referenz auf diese Variable bestehen bleibt.

Listing 4.56 Löschen des Originals einer Referenz

```
<?PHP

$value_org = "Original";

$value_ref = &$value_org;

printf( "Referenzvariable mit Wert \"%s\" vorher\n", $value_ref );

unset( $value_org );

printf( "Referenzvariable mit Wert \"%s\" nachher\n", $value_ref );

?>
```

Die Ausgabe dieses Programms sehen Sie in Listing 4.52. Der Wert der Referenzvariablen ändert sich erwartungsgemäß nach dem Löschen der Ursprungsvariablen nicht.

Listing 4.57 Ausgabe des Programms aus Listing 4.51

```
Referenzvariable mit Wert "Original" vorher
Referenzvariable mit Wert "Original" nachher
```

4.3.7 Funktionen als Variable

PHP hat eine Eigenschaft, die das Arbeiten mit unterschiedlichen Funktionen über denselben Aufruf ermöglicht. Wenn Sie hinter einen Variablennamen ein Klammerpaar setzen, so versucht der Interpreter eine Funktion zu finden, die sich aus dem Wert der Variablen und dem Klammerpaar zusammensetzt.

```
Schauen Sie sich hierzu das Programm in <?PHP

function first() {
   printf("Funktion 1.\n");
}

function second() {
   printf("Funktion 2.\n");
}
```

```
function third( $argument ) {
   return $argument * $argument;
}

$value = "first";
$value();

$value = "second";
$value();

$value = "third";
$result = $value( 4 );
printf( "Ergebnis der Funktion $value: %d.\n", $result );

?>
```

Listing 4.53 an. Es werden zwei Funktionen mit dem Namen `first()` und `second()` deklariert. Diese Funktionen sind für dieses Beispiel sehr einfach gehalten. Das Verhalten des Interpreters ist aber auch für umfangreichere Funktionen mit mehreren Argumenten gültig. Dies sehen Sie an der Funktion `third()`.

Listing 4.58
Funktionsaufruf über einen Variablenwert

```
<?PHP

function first() {
   printf("Funktion 1.\n");
}

function second() {
   printf("Funktion 2.\n");
}

function third( $argument ) {
   return $argument * $argument;
}

$value = "first";
$value();

$value = "second";
$value();

$value = "third";
$result = $value( 4 );
printf( "Ergebnis der Funktion $value: %d.\n", $result );

?>
```

Nach der Deklaration der Funktionen wird die Variable `$value` mit der Zeichenkette „first“ deklariert. Dies entspricht dem Namen der ersten Funktion. Im Folgenden wird die Variable mit einem angehängten Klammerpaar aufgerufen. Analog werden die beiden anderen Funktionen aufgerufen. Wie Sie der Ausgabe in Listing 4.54 entnehmen können, funktioniert der Aufruf unterschiedlichster Funktionen über einem Variablennamen tatsächlich.

Listing 4.59 Ausgabe des Programms aus Listing 4.53

```
Funktion 1.
Funktion 2.
Ergebnis der Funktion third: 16.
```

Ein schönes Feature, das die Programmierung bei einigen Problemen erheblich vereinfacht.

4.3.8 Variable Anzahl von Funktionsargumenten

Bei einem Konsolenprogramm haben Sie bereits gesehen, wie man eine unterschiedliche Anzahl an Argumenten an das Programm übergeben kann: `$argc` und `$argv[]`. Eine ähnliche Funktionalität können Sie auch Ihren Funktionen verleihen. Eine Anwendung dieser Funktionalität haben Sie im Verlauf dieses Kapitels schon gesehen: die Funktion `printf()`.

Um eine Funktion mit diesen Fähigkeiten zu programmieren, müssen Sie nicht in tiefere Ebenen der Programmierung wandeln. In Listing 4.55 sehen Sie am Beispiel der Funktion `function_var_args()`die Programmierung und den Aufruf einer solchen benutzerdefinierten Funktion.

Im Funktionskopf werden keine Argumente angegeben. Die einzelnen Argumente holen Sie sich im Körper der Funktion ab. Um festzustellen, ob der Funktion überhaupt Argumente übergeben wurden, dient die Funktion `func_num_args()` aus dem Vorrat der eingebauten Funktionen von PHP.

Sollten der Funktion Argumente übergeben worden sein, können Sie mit der eingebauten Funktion `func_get_arg()` jedes einzelne Argument abholen und in der Funktion weiterverarbeiten. Dabei beginnt die Zählung mit null. Das erste Argument bekommen Sie also mit dem Aufruf von `func_get_arg(0)`.

Listing 4.60
Funktion mit variabler Anzahl an Argumenten

```
<?PHP

function function_var_args() {
   $count = func_num_args();
   printf("Anzahl der Funktionsargumente = %d.\n", $count );

   for ($i=0; $i < $count; $i++ ) {

      $argument = func_get_arg( $i );
      printf("$i.tes Argument: %s.\n", $argument );

   }
}

function_var_args();
function_var_args( 4, 5, "Hallo" );

?>
```

In Listing 4.55 werden die einzelnen Argumente über eine `for`-Schleife einfach wieder ausgegeben. Am Ende des Listings wird die Funktion einmal ohne Argumente und ein zweites Mal mit drei Argumenten aufgerufen.

Listing 4.61
Ausgabe des Programms aus Listing 4.55

```
Anzahl der Funktionsargumente = 0.
Anzahl der Funktionsargumente = 3.
0.tes Argument: 4.
1.tes Argument: 5.
2.tes Argument: Hallo.
```

Da Sie bei einer solchen Funktion nicht von vornherein wissen, in welcher Reihenfolge welche Argumente an die Funktion übergeben werden, benötigen Sie eine ausgeklügelte Fehlerbehandlung innerhalb der Funktion. Diese können Sie sich über Testfunktionen wie `is_double()`, `is_string()` etc. aufbauen. Wenn Sie Zeichenketten analysieren müssen, helfen Ihnen die Zeichenkettenfunktionen weiter.

4.4 Einbinden externer Programmdateien

Im Laufe der Zeit wird sich eine mehr oder weniger stattliche Anzahl an Funktionen ansammeln, die Sie nicht nur in einem Programm benutzen können. Diese Funktionen nun immer mit Cut and Paste in den neuen Quelltext einzufügen, ist zum einen sehr mühselig, zum anderen auch wieder fehleranfällig, wenn Sie an der Funktion etwas verbessern oder einen Fehler berichtigen. Wo haben Sie die Funktion überall verwendet?

Die Entwickler von PHP haben an solche Situationen gedacht und die Möglichkeit in PHP eingebaut, externe Quelltextdateien in ein PHP-Programm einzubinden.

Voraussetzung hierfür ist erst einmal, dass der Eintrag `include_path` in der Datei *php.ini* richtig gesetzt ist. In der Datei *php.ini-dist* finden Sie einige Beispiele für Linux und für Windows. Diese Datei finden Sie im Verzeichnis Ihrer PHP-Installation.

Als Beispiel lagern wir die Funktion zur Konstruktion eines neuen Kennwortes in eine externe Datei aus. Den Inhalt der Datei *getpasswd.inc.php* finden Sie in Listing 4.58.

Bei der Wahl des Dateinamens sollten Sie darauf achten, dass die Datei-Extension vom System als zu interpretierende Datei erkannt wird. Gleichzeitig sollte am Namen erkennbar sein, dass es sich nicht um ein vollständiges Programm handelt, sondern um eine Datei, die in ein PHP-Programm eingebunden werden kann.

Dazu können Sie die Datei-Extension „*.inc*“ benutzen und den Apache-Server so konfigurieren, dass Dateien mit dieser Endung als PHP-Skripte erkannt werden. Eine weitere Regelung benutzt die Endung „*.inc.php*“. Achten Sie auf den Punkt vor dem „*inc*“. Damit ist klar, dass es sich um eine PHP-Datei handelt, die eingebunden werden kann und die vom Apache-Server als PHP-Skript erkannt wird.

Diese Maßnahme dient nicht dazu, dem Apache-Server oder dem PHP-Interpreter diese Datei verständlich zu machen. Diese würden auch eine Datei mit der Endung *.txt* einbinden. Sollte allerdings jemand Unbefugtes an Ihren PHP-Dateien manipulieren, so werden demjenigen zumindest nicht sofort alle eingebundenen Dateien im Klartext angezeigt! Aus Sicherheitsgründen sollte das `include()`-Verzeichnis auch außerhalb des root-Verzeichnisses Ihres Webservers stehen. Der Server muss aber die notwendigen Rechte haben, um auf dieses Verzeichnis zugreifen zu können. Sie sollten also dem User des Servers Leserechte auf dieses Verzeichnis geben.

Listing 4.62
Einbinden eines externen Quelltextes mittels include()

```
<?PHP

include( "getpasswd.inc.php" );

$passwd = get_password( 10 );

printf( "Neues Passwort ist: %s\n", $passwd );

?>
```

Nachdem Sie also einige grundlegende Vorsichtsmaßnahmen getroffen und die Funktion in das *include*-Verzeichnis kopiert haben, ist das Arbeiten mit dieser Datei sehr einfach. Werfen Sie einen Blick auf Listing 4.57. Der Dateiname der *include*-Datei wird der `include()`-Anweisung als Zeichenkette übergeben. Anschließend haben Sie vollen Zugriff auf die in dieser Datei definierten Funktionen. In unserem Beispiel handelt es sich nur um eine Funktion. Es können aber beliebig viele Funktionen in einer solchen Datei stehen. Sie sollten sich für jedes Thema eine solche Datei anlegen.

Listing 4.63
Inhalt der Datei getpasswd.inc.php

```
<?PHP

function get_password( $length = 8 ) {

   $counter = 0;
   $password = "";

   do {
      $character = mt_rand( 48, 126 );

      $password .= chr($character );
      $counter++;

   } while ( $counter < $length );

   return $password;
}

?>
```

Diese Dateien werden bei jedem Aufruf des Programms neu eingebunden. Da dies mit Festplattenzugriff und Dateifunktionen zu tun hat, dauert dieser Vorgang umso länger, je größer diese Datei ist. Passen Sie den Inhalt einer solchen *include*-Datei Ihrer Umgebung an.

Sie können mit solchen *include*-Dateien nicht nur Funktionen einbinden, sondern alle PHP-Konstrukte, die Ihnen einfallen, zum Beispiel grundlegende Variablen mit Ihren Adressangaben oder Variablen mit Benutzer- und Kennwortangaben für eine Datenbank. Im letzteren Fall müssen Sie besonders vorsichtig mit diesen *include*-Dateien umgehen.

Wenn Sie mehrere *include*-Dateien einsetzen, kann es passieren, dass eine oder mehrere Dateien ebenfalls durch *include*-Dateien eingebunden werden. Dies führt zu keinem Fehler, es kostet nur unnötig Zeit, da diese Datei(en) an jeder geforderten Stelle eingebunden werden.

Sie können dies verhindern, indem Sie statt der Anweisung `include()` die Anweisung `include_once()` benutzen. Wenn der PHP-Interpreter bei dieser Form der Einbindung feststellt, dass eine Datei schon eingebunden wurde, wird dieser Vorgang nicht nochmals durchgeführt. Die Anweisung lautet in einem solchen Fall für unser Beispiel

```
include_once( "getpasswd.inc.php" );
```

Sollte die `include()`- oder die `include_once()`-Anweisung die geforderte Datei nicht finden, wird eine Warnung ausgegeben, das Programm läuft aber weiter. Dies kann zu nicht gewünschten Effekten und Fehlern führen. Um dies zu vermeiden, gibt es die Anweisungen `require()` und `require_once()`. Diese beiden Anweisungen unterscheiden sich nur in einem Punkt von den `include`-Anweisungen: Bei einer fehlenden Datei wird ein fataler Fehler gemeldet und das Programm abgebrochen.