Was gibt's Neues, PHP 5 ?

Marc-Alexander Prowe
marc-a@virtuelle-maschine.de

23. August 2004

Inhaltsverzeichnis

Vorwort

Das vorliegende Dokument war für mich in doppelter Hinsicht eine neue Erfahrung. So konnte ich erstmals die neuen Funktionalitäten der kommenden PHP-Version ausführlich »erleben«, aber auch die Erstellung an sich mit Hilfe von LaTeX war für mich Neuland.

Mein Feuer für die neue Version von PHP war schnell entzündet. Vieles, was man aus anderen Sprachen kennt und was man sich schon immer in PHP gewünscht hat, ist nun implementiert. Und auch zahlreiche neue, eigenständige Ideen wissen zu begeistern.

Die Neuerungen im Schnelldurchgang:

- Die Art und Weise, wie PHP 5 mit Objekten umgeht, wurde komplett neu geschrieben, ist nun performanter und bietet deutlich erweiterte Möglichkeiten.
- Fehler können durch ein einheitliches System, das aus vielen anderen Programmiersprachen bekannt ist, besser abgefangen werden.
- Die XML-Unterstützung wurde komplett rundumerneuert. Alles basiert auf nur noch einer Bibliothek (der libxml des GNOME-Projekts), ist zuverlässiger, schneller und untereinander kompatibel.
- Ein Refelction-API ermöglicht es, während der Laufzeit detailierte Informationen über eingebaute oder selbstdefinierte Klassen und Funktionen zu erfragen.
- Das Handling von Datenstrukturen wird durch eine eingebaute Klassen-Bibliothek (der Standard-PHP-Library) vereinfacht und vereinheitlicht.
- Man kann auch ohne dezidierten Datenbank-Server mit Hilfe der eingebauten SQLite-Funktionen Datenbanken erstellen und mit SQL bearbeiten und abfragen.

- Außerdem sind, wie schon bei den vorherigen Versionssprüngen üblich, neue Funktionen hinzugekommen, die man sich teilweise schon immer gewünscht, aber nachzufragen nie gewagt hatte.

Dieses Dokument ist – und wird es wohl auch bleiben – »work in progress«. Es gibt einige Baustellen, die nach und nach geschlossen werden sollen, doch wie ich die PHP-Entwickler kenne, sind neue Features implementiert, ehe man die alten beschreiben konnte.

Wenn Sie Anmerkungen haben, scheuen Sie sich nicht, mir zu schreiben. Meine E-Mail-Adresse lautet: marc-a@virtuelle-maschine.de

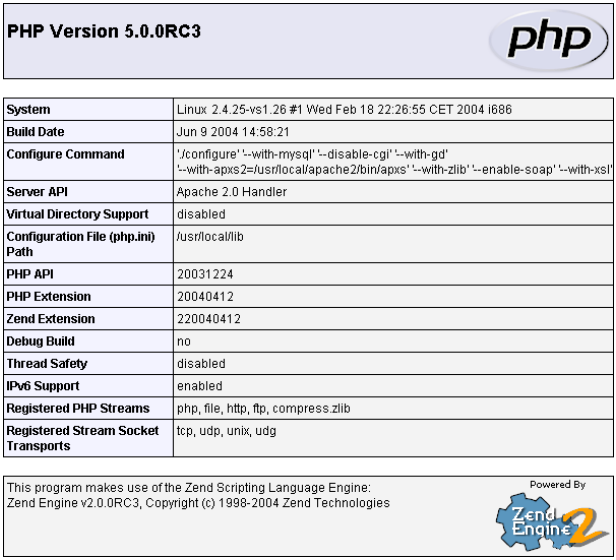
Nun wünsche ich Ihnen viel Spaß und Erfolg beim Eintauchen in die neue Version von PHP.

Marc-Alexander Prowe, im Juli 2004

Nachtrag August 2004:
Groben Schnitzer im Kapitel 2 korrigiert. Dank an Udo von Eynern für den Hinweis!

1. OO Reloaded

Die größten und augenscheinlichsten Veränderungen in PHP 5 betreffen den Umgang mit Objekten. Die Zend-Engine 2, die in PHP 5 zum Einsatz kommt, bietet gerade in Hinblick auf Objekt-Behandlung erweiterte Möglichkeiten und höhere Performance.



PHP Version 5.0.0RC3

System	Linux 2.4.25-vs1.26 #1 Wed Feb 18 22:26:55 CET 2004 i686
Build Date	Jun 9 2004 14:58:21
Configure Command	'./configure' '--with-mysql' '--disable-cgi' '--with-gd' '--with-apxs2=/usr/local/apache2/bin/apxs' '--with-zlib' '--enable-soap' '--with-xsl'
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/usr/local/lib
PHP API	20031224
PHP Extension	20040412
Zend Extension	220040412
Debug Build	no
Thread Safety	disabled
IPv6 Support	enabled
Registered PHP Streams	php, file, http, ftp, compress.zlib
Registered Stream Socket Transports	tcp, udp, unix, udg

This program makes use of the Zend Scripting Language Engine:
Zend Engine v2.0.0RC3, Copyright (c) 1998-2004 Zend Technologies

Powered By Zend Engine 2

Abbildung 1.1.: In PHP 5 kommt die Zend-Engine 2 zum Einsatz (Ausschnitt aus der phpinfo-Ausgabe)

Viele der Neuerungen werden Ihnen aus anderen objektorientierten Programmiersprachen bekannt vorkommen; gerade Java (welches ja selbst die Verwandschaft mit C++ nicht leugnen kann) stand für das eine oder andere Konzept Pate. Es gibt auch eine Reihe von eigenständigen neuen Ideen, die Einzug in PHP 5 gehalten haben. Und immer wurde versucht, dem Anspruch von PHP gerecht zu werden, eine einfache und trotzdem mächtige Sprache zu sein, in die sich auch ein Programmieranfänger schnell einfinden kann.

Zu den wichtigsten Änderungen in Bezug auf die objektorientierte Programmierung zählen:

- Objektvariablen speichern nun nicht mehr das gesamte Objekt, sondern nur noch eine Referenz auf das Objekt (siehe Abschnitt 1.1).
- Durch Einführung von Sichtbarkeitsmodifizierern für Klassenelemente ist es nun möglich, Informationen zu kapseln (siehe Kapitel 2 auf Seite 7)
- Ein weiteres Paradigma der objektorientierten Programmierung ist die polymorphe Erscheinung von Objekte durch Vererbung. Diese wurde auch schon in PHP 4 unterstützt, aber die neue Version bietet erweiterte Möglichkeiten, Vererbung zu steuern und Typen zu implementieren und sicherzustellen (siehe Kapitel 4 auf Seite 25).
- Klassen können – ein wenig sinnvolle Organisation und Benennung vorausgesetzt – automatisch geladen werden. Endlose `include_once()`-Kaskaden können also in Zukunft entfallen (Siehe Abschnitt 6.1 auf Seite 43)

1.1. Objektvariablen sind Referenzen

In PHP 4, dessen Herz die Zend-Engine 1 war, wurden Objekte wie andere native Datentypen (Integer, Float etc.) behandelt. Hat man sie einer Variablen zugewiesen oder einer Funktion als Parameter übergeben, dann wurden sie jedes Mal im Ganzen kopiert, was deutlich Performance kostete. Dies konnte man zwar vermeiden, wenn man explizit den Referenz-Operator benutzte (in der Form `$Obj =& new Klasse()` bzw. `funktion methode(&$Obj)`), doch war dies umständlich und wurde daher nur selten gemacht. Die Zend-Engine 2 behandelt Objekte nun immer als Referenzen, was zu einem von PHP 4 abweichendem Verhalten führen kann. Hierzu ein Beispiel:

Listing 1.1: Vergleich der Objektbehandlung in PHP 4 und PHP 5

```
<?php
class MeineKlasse
{
    var $a;
```

```
    function MeineKlasse()
    {
        $this->a = 1;
    }

    function erhoehe()
    {
        $this->a += 1;
    }
}

function go($Obj)
{
    $Obj->erhoehe();
}

$MeinObjekt = new MeineKlasse();
go($MeinObjekt);
echo "a hat den Wert: " . $MeinObjekt->a;
?>
```

In PHP 4 wird

```
a hat den Wert: 1
```

ausgegeben, in PHP 5 aber

```
a hat den Wert: 2
```

Diese Änderung der Zend-Engine 2 ist die weitreichendste und auch diejenige, die die meisten Fallen in Hinblick auf die Rückwärts-Kompatibilität birgt.

Fast alle anderen Änderungen der Zend-Engine 2 sind kompatibel mit der Vorversion und in PHP 4 geschriebener Code sollte sich ansonsten ohne Probleme auch in PHP 5 ausführen lassen – zumindest solange man nicht auf die Idee kam, eigene Funktionen mit zwei Unterstrichen beginnen zu lassen oder Bezeichner zu verwenden, die nun als Schlüsselwörter eingeführt wurden[1].

Hat man jedoch alten Code, der sich auf das implizierte Kopieren verlässt (was ja eher unbewusst passiert sein wird), dann ist Vorsicht geboten: Es wird keine

[1] auch für die Zukunft ist es keine gute Idee, Bezeichner zu verwenden, die mit zwei Unterstrichen beginnen oder die aus anderen Programmiersprachen bekannt sind.

Warnung ausgegeben (was auch gar nicht möglich wäre) und der Fehler, der ja nicht syntaktischer sondern logischer Natur ist, lässt sicht nur schwer finden.

1.2. Das Gleiche oder das Selbe? Objekt-Vergleiche

In PHP 4 wurde der Operator === eingeführt, der zusätzlich zum Wert-Vergleich noch auf Typ-Gleichheit prüft. In PHP 5 muss nun beachtet werden, dass dieser Operator bei Objekten-Vergleichen nicht mehr prüft, ob die beiden Operanden vom gleichen Klasentyp sind, sondern ob sie ein und das selbe Objekt referenzieren.

Listing 1.2: Vergleich zweier Objekte

```
<?php
Class Klasse
{
    var $a = 10;
}
$Objekt1 = new Klasse();
$Objekt2 = new Klasse();
$Objekt3 = $Objekt1;
if ( $Objekt1 === $Objekt2 ) {
    echo "\$Objekt1 === \$Objekt2\n";
} else {
    echo "\$Objekt1 !== \$Objekt2\n";
}
if ( $Objekt1 === $Objekt3 ) {
    echo "\$Objekt1 === \$Objekt3\n";
} else {
    echo "\$Objekt1 !== \$Objekt3\n";
}
?>
```

PHP 4 gibt ...

```
$Objekt1 === $Objekt2
$Objekt1 === $Objekt3
```

... aus, während sich PHP 5 folgendermaßen verhält:

```
$Objekt1 !== $Objekt2
$Objekt1 === $Objekt3
```

1.3. Automatisches Dereferenzieren von Objekten, die von Funktionen zurückgegeben werden

In PHP 4 war es nicht möglich, unmittelbar mit einem Objekt weiter zu arbeiten, dass von einer Funktion oder Methode zurückgegeben wurde. Vielmehr musste das Objekt in einer Variablen gespeichert werden und die Methode anschließend über diese Variable aufgerufen werden. Es folgt ein Beispiel einer Fabrik-Methode (ein übliches Entwurfsmuster der objektorientierten Programmierung), welches abhängig von zu übergebenden Parametern unterschiedliche Klassen zurückgibt:

Listing 1.3: PHP 4: Umweg über eine Variable

```
1  <?php
2  class KlasseEins
3  {
4      function ausgabe()
5      {
6          echo "In Klasse Eins\n";
7      }
8  }
9
10 class KlasseZwei
11 {
12     function ausgabe()
13     {
14         echo "In Klasse Zwei\n";
15     }
16 }
17
18 function meineFabrikMethode($art)
19 {
20     switch ($art) {
21         case "eins":
22             return new KlasseEins();
```

```
23          case "zwei":
24              return new KlasseZwei();
25      }
26  }
27
28  $Obj =& meineFabrikMethode("eins");
29  $Obj->ausgabe();
30  $Obj =& meineFabrikMethode("zwei");
31  $Obj->ausgabe();
32  ?>
```

Dies kann in PHP 5 nun verkürzt geschrieben werden (es werden nur die geänderten Zeilen gezeigt):

Listing 1.4: Automatisches Dereferenzieren von Objekten

```
28  meineFabrikMethode("eins")->ausgabe();
29  meineFabrikMethode("zwei")->ausgabe();
30  ?>
```

Im Abschnitt 5.1.2 auf Seite 40 wird ein weiters Entwurfsmuster vorgestellt (das Singleton-Pattern), dessen Benutzung von der automatischen Dereferenzierung in PHP 5 profitiert.

Achtung: Es ist nicht möglich, mit einem Objekt unmittelbar nach seiner Konstruktion weiterzuarbeiten. Folgendes funktioniert daher (noch?) nicht:

Listing 1.5: Unmittelbares Dereferenzieren nach der Konstruktion

```
<?php
class KlasseEins
{
    function ausgabe()
    {
        echo "In Klasse Eins\n";
    }
}
//! folgendes funktioniert (noch) nicht:
(new KlasseEins())->ausgabe();
?>
```

2. Die im Dunkeln sieht man nicht

Eine häufig anzutreffende Forderung an objektorientierte Programmierung ist die Kapselung von Information (Information-Hiding). Dies war in PHP 4 kaum zu realisieren, da auf alle Klassenelemente von außen zugegriffen und Werte damit unkontrolliert verändert werden konnten.

PHP 5 führt daher sowohl für Eigenschaften als auch Methoden die aus anderen objektorientierten Programmiersprachen bekannten Schlüsselwörter `public`, `private` und `protected` ein.

2.1. Sichtbarkeit von Eigenschaften

Wird eine Eigenschaft einer Klasse als `public` gekennzeichnet, so ist sie nach außen sichtbar und kann direkt gelesen und verändert werden. Sie verhält sich damit genau wie in PHP 4, und aus Kompatibilitätsgründen ist dies auch das Standardverhalten, wenn bei der Deklaration kein Sichtbarkeitsmodifizierer angegeben wird. Eine als `private` deklarierte Eigenschaft kann dagegen nur in der Klasse, in der sie deklariert wurde, gelesen und verändert werden, während eine Deklaration als `protected` zudem die Sichtbarkeit auch in von dieser Klasse abgeleiteten Klassen sicherstellt.

Listing 2.1: Sichtbarkeit von Eigenschaften

```
1 <?php
2 class KlasseEins
3 {
4     public    $a = "Ich bin a!\n";
5     protected $b = "Ich bin b!\n";
```

```
6       private   $c = "Ich bin c!\n";
7
8       function ausgabe()
9       {
10          echo "In KlasseEins:\n";
11          echo $this->a;
12          echo $this->b;
13          echo $this->c;
14      }
15  }
16  class KlasseZwei extends KlasseEins
17  {
18      function ausgabe()
19      {
20          echo "In KlasseZwei:\n";
21          echo $this->a;
22          echo $this->b;
23          echo $this->c;
24          parent::ausgabe();
25      }
26  }
27  $ObjektEins    = new KlasseEins();
28  $ObjektZwei    = new KlasseZwei();
29  $ObjektEins->ausgabe();
30  $ObjektZwei->ausgabe();
31  echo "Ausserhalb der Klassen:\n";
32  echo $ObjektEins->a;
33  echo $ObjektEins->b;
34  ?>
```

Dieses Skript gibt folgendes aus:

```
In KlasseEins:
Ich bin a!
Ich bin b!
Ich bin c!
In KlasseZwei:
Ich bin a!
Ich bin b!

Notice: Undefined property:  KlasseZwei::$c in
/htdocs/php5neu/sichtbarkeiteigenschaften.php5 on line 23

In KlasseEins:
Ich bin a!
Ich bin b!
```

```
Ich bin c!
Ausserhalb der Klassen:
Ich bin a!

Fatal error: Cannot access protected property KlasseEins::$b
in /htdocs/php5neu/sichtbarkeiteigenschaften.php5 on line 33
```

Bitte beachten Sie: Damit PHP überhaupt auf die nicht definierte Eigenschaft hinweist, muss in der php.ini durch den Eintrag `error_reporting=E_ALL` die Ausgabe von Warnungen und Bemerkungen eingeschaltet sein. Dies ist für einen Entwicklungsrechner unbedingt zu empfehlen, während auf einem Produktionsserver PHP generell keine Fehlermeldungen ausgeben sollte (in der php.ini `display_errors=Off` einstellen).

Mit der als `public` deklarierten Eigenschaft `$a` der Klasse KlasseEins gibt es überhaupt keine Probleme – auf sie kann von überall zugegriffen werden. Ist eine Eigenschaft als `protected` deklariert (wie `$b`), dann kann auf sie in allen abgeleiteten Klassen zugegriffen werden, außerhalb von Klassen oder auch in Klassen, die nicht von KlasseEins abgeleitet sind, ist die Eigenschaft jedoch nicht sichtbar. Auf die als `private` deklarierte Eigenschaft `$c` kann dagegen ausschließlich in der Klasse KlasseEins zugegriffen werden.

Spannend ist in dieser Hinsicht das unterschiedliche Fehlerverhalten: Ein Zugriffsversuch in einer abgeleiteten Klasse auf eine private Eigenschaft wird mit einem Hinweis quittiert; das Skript läuft jedoch weiter. Bei einem Zugriff von außerhalb der Klasse zeigt PHP 5 jedoch einen fatalen Fehler an und beendet die weitere Ausführung. Es bleibt abzuwarten, ob sich dieses nicht ganz konsistente Verhalten auch noch in späteren Versionen der Zend-Engine 2 wiederfinden wird.

2.2. Sichtbarkeit von Methoden

Der Zugriff auf Methoden wird ebenfalls mit den Schlüsselwörten `public`, `protected` und `private` gesteuert, das Verhalten entspricht genau dem der Eigenschaften.

Listing 2.2: Sichtbarkeit von Klassen-Methoden

```
<?php
class KlasseEins
{
    public    function a() {
        echo "Ich bin a!\n";
    }
    protected function b() {
        echo "Ich bin b!\n";
    }
    private   function c() {
        echo "Ich bin c!\n";
    }
    function ausgabe()
    {
        echo "In KlasseEins:\n";
        $this->a();
        $this->b();
        $this->c();
    }
}
class KlasseZwei extends KlasseEins
{
    function ausgabe()
    {
        echo "In KlasseZwei:\n";
        $this->a();
        $this->b();
        $this->c();
        parent::ausgabe();
    }
}
$ObjektEins = new KlasseEins();
$ObjektZwei = new KlasseZwei();
$ObjektEins->ausgabe();
$ObjektZwei->ausgabe();
?>
```

Hier führt schon der Zugriffsversuch der abgeleiteten Klasse auf eine private Methode zu einem Fehler und Ausführungsabbruch:

```
In KlasseEins:
Ich bin a!
Ich bin b!
Ich bin c!
In KlasseZwei:
```

```
Ich bin a!
Ich bin b!

Fatal error: Call to private method KlasseEins::c() from
context 'KlasseZwei' in
/htdocs/php5neu/sichtbarkeitmethoden.php5 on line 28
```

Unabhängig vom Zugriffsmodifizierbar ist es aber weiterhin möglich, Methoden in abgeleiteten Klassen zu überschreiben. Das folgende Skript wird ohne Murren ausgeführt:

Listing 2.3: Private Methoden in abgeleiten Klassen überschreiben

```
1  <?php
2  class KlasseEins
3  {
4      private function getString()
5      {
6          return "In Klasse Eins\n";
7      }
8      public function ausgabe()
9      {
10         echo $this->getString() . "\n";
11     }
12 }
13 class KlasseZwei extends KlasseEins
14 {
15     private function getString()
16     {
17         return "In Klasse Zwei\n";
18     }
19     public function ausgabe()
20     {
21         echo $this->getString() . "\n";
22     }
23 }
24 $Obj = new KlasseZwei();
25 $Obj->ausgabe();
26 ?>
```

```
In Klasse Zwei
```

2.3. Abfangen von Zugriffen auf nicht vorhandene Klassen-Elemente

In diesem Abschnitt werden drei neue Methoden vorgestellt, die, sofern sie in einer Klasse definiert sind, immer dann aufgerufen werden, wenn Zugriffe auf nicht vorhandene Klassenelemente stattfinden. Da ihnen der Name der fehlenden Eigenschaft bzw. Methode und gegebenenfalls der zugewiesene Wert bzw. die Parameterliste übergeben werden, hat man die Möglichkeit, eine entsprechende Eigenschaft bzw. Methode zu simulieren. Ganz neu ist diese Funktionalität übrigens nicht: Es handelt sich dabei um die Overload-Extension, welche bislang im PECL zu finden war und die jetzt zu einem festen Bestandteil der PHP 5-Grundfunktionalität geworden ist.

Was auf den ersten Blick sehr mächtig und nützlich aussieht, stellt sich bei genauer Betrachtung tatsächlich als sehr mächtig, aber nur in speziellen Fällen als nützlich heraus. Sie sollten lernen, diese speziellen Fälle zu erkennen, denn dann stellt diese neue Funktionalität eine wirkliche Bereicherung dar.

2.3.1. Zugriff auf Eigenschaften mit `__get()` und `__set()`

Im Grunde ermöglichen diese beiden Methoden virtuelle[1] Eigenschaften. `__get()` liefert den Wert, `__set()` setzt ihn. Beiden wird der Name der aufgerufenen Eigenschaft als erstes Argument übergeben, `__set()` wird zusätzlich der zugewiesene Wert als zweites Argument übermittelt.

Es folgt ein einfaches Beispiel. Die dort definierte Klasse stellt die Eigenschaften `de`, `fr` und `gb` zur Verfügung, wobei die Groß- und Klein-Schreibung dieser Eigenschaften keine Rolle spielt:

Listing 2.4: Beispiel: __get() und __set()

```
1 <?php
```

[1] Definition nach Kaiser:
virtuell: man sieht es, aber man hat es nicht;
transparent: man hat es, aber man sieht es nicht;
physikalisch: man hat es und sieht es.

```php
class Hauptstadt
{
    protected $hiddenArray = array(
        "de" => "Bonn",
        "fr" => "Paris",
        "gb" => "London"
    );
    function __get($bezeichner)
    {
        if ( isset($this->hiddenArray[strtolower($bezeichner)]) ) {
            return $this->hiddenArray[strtolower($bezeichner)];
        }
    }

    function __set($bezeichner, $wert)
    {
        if ( isset($this->hiddenArray[strtolower($bezeichner)]) ) {
            $this->hiddenArray[strtolower($bezeichner)] = $wert;
            return;
        }
    }
}

$Objekt = new Hauptstadt();
echo $Objekt->DE . "\n";
echo $Objekt->fR . "\n";
echo $Objekt->gb . "\n";
$Objekt->de = "Berlin";
echo $Objekt->DE . "\n";
?>
```

```
Bonn
Paris
London
Berlin
```

So weit, so klar (hoffe ich zumindest ...). Aber bevor Sie jetzt auf die Idee kommen, Ihre Eigenschaften nur noch mit diesen beiden neuen Freunden abzubilden, schauen Sie sich bitte das folgende ausführlichere Beispiel an:

Listing 2.5: Weiteres Beipiel: __get() und __set()

```php
<?php
class KlasseEins
{
```

```
4       private   $a = "Ich bin a!\n";
5       protected $hiddenArray = array(
6           "x" => "Ich bin x!\n"
7       );
8
9       function ausgabe()
10      {
11          echo "In KlasseEins:\n";
12          echo $this->x;
13          echo $this->z;
14          echo $this->a;
15          $this->x = "Ich, x, wurde geändert!\n";
16          echo $this->x;
17      }
18
19      function __get($bezeichner)
20      {
21          if ( $bezeichner == "a" ) {
22              return "Aufruf von \$a in __get()\n";
23          }
24          if ( isset($this->hiddenArray[$bezeichner]) ) {
25              return $this->hiddenArray[$bezeichner];
26          }
27      }
28
29      function __set($bezeichner, $wert)
30      {
31          if ( isset($this->hiddenArray[$bezeichner]) ) {
32              $this->hiddenArray[$bezeichner] = $wert;
33              return;
34          }
35      }
36  }
37  class KlasseZwei extends KlasseEins
38  {
39      function ausgabe()
40      {
41          echo "In KlasseZwei:\n";
42          echo $this->x;
43          echo $this->z;
44          echo $this->a;
45      }
46  }
47
48  $ObjektEins    = new KlasseEins();
49  $ObjektZwei    = new KlasseZwei();
50  $ObjektEins->ausgabe();
```

```
51
52 echo "\n";
53 $ObjektZwei->ausgabe();
54 echo "\n";
55 echo "Ausserhalb der Klassen:\n";
56 echo $ObjektEins->x;
57 echo $ObjektEins->z;
58 //! Auf private oder geschützte Eigenschaften kann
59 //! via __get() nicht zugegriffen werden, also
60 //! führt Folgendes zu einem Fehler:
61 echo $ObjektEins->a;
62 ?>
```

```
In KlasseEins:
Ich bin x!
Ich bin a!
Ich, x, wurde geändert!

In KlasseZwei:
Ich bin x!
Aufruf von $a in __get()

Ausserhalb der Klassen:
Ich, x, wurde geändert!

Fatal error: Cannot access private property KlasseEins::$a in
/htdocs/php5neu/__get__set.php5 on line 61
```

Der Zugriff auf die (als Objekt-Variable nicht vorhandene) Eigenschaft `$x` wird vollständig von den Methoden `__get()` und `__set()` übernommen. Dabei ist es egal, von wo zugegriffen wird; `$x` erscheint wie eine als `public` deklarierte Eigenschaft.

Seltsam ist das Verhalten, wenn eine als `private` deklarierte Eigenschaft in der Klasse existiert (in diesem Beispiel `$a`): In der eigenen Klasse kann direkt auf sie zugegriffen werden (was ja nachvollziebar ist), in der abgeleiteten Klasse wird der Aufruf von den neuen Methoden abgefangen und von außerhalb führt der Zugriff zum Programmabbruch. Es bleibt abzuwarten, ob dieses inkonsistente Verhalten in späteren Versionen (getestet wurde bis einschließlich Version 5.0 RC3) korrigiert wird.

Generell scheint es also eine gute Idee, die Eigenschaften, die mit den beiden neuen Methoden bereitgestellt werden sollen, zu »verstecken« – zum Beispiel

in einem eigenen Array.

Der Zugriff auf die nicht vorhandene Eigenschaft `$z` wird durch die neuen Methoden abgefangen, aber nicht behandelt. Das ist schlecht! Sie sollten es sich daher *immer* zur Gewohnheit machen, eine Warnmeldung auszugeben, wenn `__get()` oder `__set()` für einen Bezeichner aufgerufen werden, den Sie nicht vorgesehen haben (im folgenden Listing die Zeilen 27 und 36):

Listing 2.6: Zugriff auf nicht vorgesehene Eigenschaft abfangen (Ausschnitt)

```
19      function __get($bezeichner)
20      {
21          if ( $bezeichner == "a" ) {
22              return "Aufruf von \$a in __get()\n";
23          }
24          if ( isset($this->hiddenArray[$bezeichner]) ) {
25              return $this->hiddenArray[$bezeichner];
26          }
27          echo "Achtung: Aufruf der nicht definierten Eigenschaft '"
                . __CLASS__ . "::" . $bezeichner . "' in der Methode '"
                 . __METHOD__ . "'\n";
28      }
29
30      function __set($bezeichner, $wert)
31      {
32          if ( isset($this->hiddenArray[$bezeichner]) ) {
33              $this->hiddenArray[$bezeichner] = $wert;
34              return;
35          }
36          echo "Achtung: Aufruf der nicht definierten Eigenschaft '"
                . __CLASS__ . "::" . $bezeichner . "' in der Methode '"
                 . __METHOD__ . "'\n";
37      }
```

Sie merken schon: Ich bin kein großer Freund dieser beiden neuen Methoden. Zumindest, wenn man sie als reinen *Ersatz* für »gemeine« Eigenschaften einsetzt. Das hat mehrere Gründe:

- In diesen Methoden können sich schnell sehr viele `if-else`- oder `switch`-Anweisungen ansammeln, was zu schwer lesbarem Code führt.
- Eine vernünftige Dokumentation ist nahezu unmöglich. Man führt Eigenschaften ein, hat aber im Code selbst kaum die Möglichkeit, diese zu dokumentieren. (Ich spreche hier von Inline-Dokumentation, bei der

der Quellcode so gekennzeichnet wird, dass aus ihm die Dokumentation automatisch generiert werden kann.)

- Klassenansichten (z.B. in Editoren) werden diese Eigenschaften nicht darstellen können.

Aber die gute Nachricht ist: Sie sind für solche Zwecke ja gar nicht auf diese neue Funktionalität angewiesen! Keiner hindert Sie, eine als `private` bzw. `protected` deklarierte Eigenschaft einzuführen und passend dazu entsprechende get- bzw set-Methoden zu definieren, die auf dieser Eigenschaft operieren. So etwas lässt sich dann auch sauber dokumentieren und ist nachvollziebar.

Es sind aber viele Anwendungen denkbar, in denen man `__get()` und `__set()` sinnvoll einsetzen kann – so können zum Beispiel Wrapper-Klassen von dieser neuen Funktionalität profitieren. In einem späteren Kapitel zu einer neuen XML-Funktionalilität (Simple-XML, Kapitel 13 auf Seite 77) wird eine weitere Anwendung gezeigt.

2.3.2. Zugriff auf Methoden mit `__call()`

Das, was das Duo `__get()` und `__set()` für nicht vorhandene Eigenschaften ist, ist `__call()` für nicht vorhandene Methoden. `__call()` erwartet zwei Argumente: Den Bezeichner der aufgerufenen (aber nicht vorhandenen) Methode und ein Array mit den angegebenen Parametern.

Im folgenden Beipiel wird die Methode `getHauptstadt` simuliert, wobei die Schreibweise beim Methodenaufruf keine Rolle spielt:

Listing 2.7: Beispiel: __call()

```
<?php
class Hauptstadt
{
    protected $hiddenArray = array(
        "de" => "Berlin",
        "fr" => "Paris",
        "gb" => "London"
    );
    function __call($bezeichner, $paramArray)
```

```
    {
        switch (strtolower($bezeichner)) {
            case "gethauptstadt":
                if ( isset($paramArray[0]) && isset($this->hiddenArray
                    [strtolower($paramArray[0])]) ) {
                    return $this->hiddenArray[strtolower($paramArray
                        [0])];
                } else {
                    return FALSE;
                }
            break;
        }
        return FALSE;

    }
}

$Objekt = new Hauptstadt();
echo $Objekt->GetHauptStadt("DE") . "\n";
echo $Objekt->GETHAUPTSTADT("fR") . "\n";
echo $Objekt->getHauptstadt("gb") . "\n";
?>
```

```
Berlin
Paris
London
```

Das zu `__get()` und `__set()` Gesagte kann analog auch zu `__call()` wiederholt werden: Es ist kein Ersatz für »gemeine« Methoden, denn sonst würde der Code unübersichtlich, wäre schlecht zu dokumentieren und Klassenansichten würden die Segel streichen. Aber sinnvoll eingesetzt ist auch diese Methode eine echte Bereicherung. So gibt es zum Beipiel im PEAR eine Klasse `PEAR_Autoloader`, die es ermöglicht, eine Klasse transparent auf mehrere Klassen aufzuteilen, welche dann nur noch bei Bedarf geladen und kompiliert werden müssen – was einen deutlichen Performancevorteil mit sich bringen kann. Dafür nutzt `PEAR_Autoloader` die Methode `__call()`.

3. Auf- und Abbau von Objekten sowie Angriff der Klon-Krieger

3.1. Ein Bezeichner für alle Konstruktoren

Jede Klasse in PHP kann (genau) eine Methode enthalten, die automatisch aufgerufen wird, wenn ein Objekt instanziert wird. In einer solchen sogenannten Konstruktor-Methode werden üblicherweise Initalisierungen vorgenommen, die das entstehende Objekt in den richtigen Ausgangszustand versetzen.

In PHP 4 mussten Konstruktor-Methoden den gleichen Bezeichner wie die Klasse tragen, damit die Zend-Engine sie als Konstruktoren erkannte. Wollte man aus einer abgeleiteten Klasse den Konstruktor der Elter-Klasse aufrufen, dann musste man den Namen der Elter-Klasse kennen. Das machte es zu Beispiel schwierig, eine Klasse in ihrer Hierarchie zu verschieben. Dieses Problem wurde in PHP 5 durch einen einheitlichen Bezeichner für die Konstruktor-Methode gelöst: `__construct()`

Listing 3.1: Beispiel: __construct()

```
<?php
class KlasseEins
{
    function __construct()
    {
        echo "Konstruktor Klasse Eins\n";
    }
}
class KlasseZwei extends KlasseEins
{
    function __construct()
    {
        echo "Konstruktor Klasse Zwei\n";
```

```
        parent::__construct();
    }
}
$Obj = new KlasseZwei();
?>
```

Dies gibt folgendes aus:

```
Konstruktor Klasse Zwei
Konstruktor Klasse Eins
```

Findet PHP 5 in einer Klasse keine Methode namens `__construct()`, dann wird automatisch nach einer Methode gesucht, die den gleichen Bezeichner wie die Klasse trägt. Dies stellt sicher, dass bestehender PHP 4-Code weiterhin funktioniert – natürlich nur, wenn sie keine eigene Methode mit dem Bezeichner `__construct()` versehen haben. (Generell ist es nie eine gute Idee, die Bezeichner eigener Elemente mit zwei Unterstrichen zu beginnen!)

3.2. Endlich: Destruktoren!

Das, was der Konstruktor für den Aufbau eines Objektes ist, ist der Destruktor für den Abbau. Im Destruktor können zum Beispiel Aufräumarbieten erledigt (geöffnete Dateien oder Datenbank-Verbindungen schließen, sonstige Ressourcen freigeben etc.) und Nachrichten zur Fehler-Suche geschrieben werden.

Analog zum Konstruktor wird eine Klassen-Methode dann als Destruktor angesehen, wenn sie mit dem Bezeichner `__destruct()` versehen ist.

```
Konstruktor
Vor Zerstörung des Objekts A
Nach Zerstörung des Objekts A
Vor Zerstörung des Objekts B
Destruktor
Nach Zerstörung des Objekts B
```

Wie man an diesem Bespiel gut erkennen kann, wird der Destruktor erst dann aufgerufen, wenn keine Referenz auf das Objekt mehr existiert. Jedes Objekt

führt einen Referenzenzähler mit sich; wird eine Referenz auf das Objekt erzeugt, dann wird dieser Zähler um eins erhöht (im Beispiel jeweils in den Zeilen 13 und 14), wird eine Referenz zerstört, dann wird der Zähler um eins erniedrigt (hier in den Zeilen 16 und 19). Erreicht dieser Zähler Null, dann wird der Destruktor aufgerufen und anschließend das Objekt aus dem Speicher entfernt.

Destruktoren von eventuell vorhandenen übergeordneten Klassen werden nicht automatisch aufgerufen. Um den Destruktor der Elter-Klasse auszuführen, muss man im Destruktor der abgeleiteten Klasse explizit `parent::__destruct()` aufrufen. Auch hier entspricht das Verhalten des Destuktors also dem des Konstruktors.

3.3. Klonen von Objekten

3.3.1. Das Schlüsselwort `clone`

Bisweilen ist es erwünscht, dass man eine »echte« Kopie eines Objektes erzeugt, welches man unabhängig vom Ausgangsobjekt weiter manipulieren kann. Bei der einfachen Zuweisung `$Obj2=$Obj1` würden wir ja kein neues Objekt erhalten, sondern nur eine weitere Referenz auf das alte erzeugen. Daher führt PHP 5 das Schlüsselwort `clone` ein:

Listing 3.2: Klonen eines Objekts

```
<?php
class Schaf
{
    public $name = "";
    function __construct($name)
    {
        $this->name = $name;
    }
}

$Dolly = new Schaf("Dolly");
$Dolly2 = $Dolly;
$Dolly2->name = "Dolly, die Zweite!";
echo "Ohne 'clone':\n";
print_r($Dolly);
print_r($Dolly2);
```

```
$Dolly = new Schaf("Dolly");
$Dolly2 = clone $Dolly;
$Dolly2->name = "Dolly, die Zweite!";
echo "Mit 'clone':\n";
print_r($Dolly);
print_r($Dolly2);
?>
```

```
Ohne 'clone':
Schaf Object
(
    [name] => Dolly, die Zweite!
)
Schaf Object
(
    [name] => Dolly, die Zweite!
)
Mit 'clone':
Schaf Object
(
    [name] => Dolly
)
Schaf Object
(
    [name] => Dolly, die Zweite!
)
```

3.3.2. Den Kopiervorgang beeinflussen

Das in Listing 3.2 auf der vorherigen Seite durch Klonen entstandene Objekt war eine 1:1-Kopie des Ausgangsobjekts. Dies ist häufig ausreichend und auch das Standardverhalten in PHP 5. Es gibt aber auch Fälle, da möchte man auf den Kopiervorgang Einfluss nehmen. Stellen Sie sich vor, dass eine Eigenschaft des zu kopierenden Objekts wiederum ein Objekt ist. Nach dem Klonen ist die gleiche Eigenschaft des neuen Objekts nur eine weitere Referenz auf das selbe (Unter-)Objekt – was vielleicht gar nicht erwünscht ist. Man hat aber zum Glück die Möglichkeit, in jeder Klasse eine `__clone()`-Methode zu definieren, die während des Kopiervorgangs aufgerufen wird. Und zwar nachdem eine 1:1-Kopie vorliegt, die wir nur noch an den gewünschten Stellen manipulieren müssen.

Listing 3.3: Das Klonen beeinflussen

```
<?php
class Fell
{
    public $farbe = "";
    function __construct($farbe)
    {
        $this->farbe = $farbe;
    }
}

class Schaf_1
{
    public $name = "";
    public $fell = NULL;
    function __construct($name)
    {
        $this->name = $name;
        $this->fell = new Fell("beige");
    }
    public function sagWas()
    {
        echo "Ich bin Schaf " . $this->name . "\n";
        echo "    und mein Fell ist " . $this->fell->farbe . "\n";
    }
}
class Schaf_2 extends Schaf_1
{
    function __clone()
    {
        $this->fell = clone $this->fell;
    }
}

$Dolly = new Schaf_1("Dolly");
$Dolly2 = clone $Dolly;
$Dolly2->name = "Dolly, die Zweite,";
$Dolly2->fell->farbe = "schwarz";
$Dolly->sagWas();
$Dolly2->sagWas();

echo "\n";

$Dolly = new Schaf_2("Polly");
$Dolly2 = clone $Dolly;
$Dolly2->name = "Polly, die Zweite,";
$Dolly2->fell->farbe = "schwarz";
```

```
$Dolly->sagWas();
$Dolly2->sagWas();
?>
```

```
Ich bin Schaf Dolly
    und mein Fell ist schwarz
Ich bin Schaf Dolly, die Zweite,
    und mein Fell ist schwarz

Ich bin Schaf Polly
    und mein Fell ist beige
Ich bin Schaf Polly, die Zweite,
    und mein Fell ist schwarz
```

Wie man sehen kann, haben beide Polly-Schafe unabhängige Fell-Objekte, während die beiden Dollys sich das selbe Fell-Objekt teilen.[1]

Bitte beachten Sie, dass der `$this`-Zeiger in der `__clone()`-Methode schon auf das *neue, 1-zu-1 kopierte* Objekt zeigt.

[1] Ja, ich weiß, bei geklonten Schafen erreicht man eine unterschiedliche Fellfarbe eigentlich nur mit einer Farbsprühpistole...

4. Familienfest und andere Schwierigkeiten

Die objektorientierte Programmierung wäre bei weitem nicht so mächtig, wenn sie nicht das Konzept der Vererbung verfolgte und die damit eng verbundende polymorphe (mehr- bzw. vielgestaltige) Erscheinung der Objekte.

Unter Polymorphie von Objekten versteht man Folgendes: Ist ein Objekt vom Typ Klasse B und Klasse B ist direkt oder indirekt von Klasse A abgeleitet, so kann man das Objekt auch als ein Objekt vom Typ Klasse A ansehen. Ruft man eine Methode des Objekts auf, die in Klasse A definiert ist und in Klasse B überschrieben wurde, dann wird die überschriebene Version der Klasse B ausgeführt.

Dazu ein Beispiel:

Listing 4.1: Polymorphie

```
<?php
class Viereck
{
    public function ausgabe()
    {
        echo "Ich bin ein Viereck...\n";
    }
    public function ausgabeEcken()
    {
        echo "    ...und ich habe 4 Ecken!\n";
    }
}
class Rechteck extends Viereck
{
    public function ausgabe()
    {
        echo "Ich bin ein Rechteck...\n";
```

```
    }
}
class Quadrat extends Rechteck
{
    public function ausgabe()
    {
        echo "Ich bin ein Quadrat...\n";
    }
}
$viereckArray = Array(
    new Viereck(),
    new Rechteck(),
    new Quadrat()
);
foreach ($viereckArray as $viereck) {
    $viereck->ausgabe();
    $viereck->ausgabeEcken();
}
?>
```

```
Ich bin ein Viereck...
    ...und ich habe 4 Ecken!
Ich bin ein Rechteck...
    ...und ich habe 4 Ecken!
Ich bin ein Quadrat...
    ...und ich habe 4 Ecken!
```

In diesem Beispiel kann jedes Element im `$viereckArray` als ein Objekt der Klasse `Viereck` angesehen werden und man kann sich darauf verlassen, dass jedes Element, dass in der Klasse `Viereck` definiert ist, auch in den abgeleiteten Klassen vorhanden ist – allein *weil* sie von der Klasse `Viereck` abgeleitet sind! Und dem Aufrufer ist es eigentlich auch egal, welche Klasse nun welches Element letztendlich definiert hat – Hauptsache, es *ist* definiert (na gut, zur Klasse passen sollte es natürlich auch).[1] Das Verhältnis der abgeleiteten Klasse zu seinen Vorfahren wir häufig als »IST_EIN«-Beziehung (IS_A) bezeichnet: `Rechteck` IST_EIN `Viereck`, `Quadrat` IST_EIN `Viereck` und `Quadrat` IST_EIN `Rechteck`.

[1] In PHP scheint es irgendwie normal, dass – sofern in abgeleiteten Klassen eine Methode überschrieben wurde – automatisch die überschriebene Methode aufgerufen wird. In anderen (meist stärker typisierten) Programmiersprachen ist dies nicht immer so selbstverständlich und wird dort als »späte Bindung« bezeichnet.

Die meisten der neuen Möglichkeiten, die in diesem Kapitel nun vorgestellt werden, sollen helfen, diese Vererbungen zu steuern oder Zugehörigkeiten sicherzustellen.

4.1. Vorlagen erzeugen: `abstract`

4.1.1. Abstrakte Klassen

Wird eine Klasse als abstract deklariert, so kann man von dieser Klasse keine Objekte instanzieren; sie dient immer nur als Ausgangsklasse für Ableitungen.

Natürlich können Sie in vielen Fällen auf abstrakte Klassen verzichten und gleich »konkretete« Klasse bereitstellen, die ebenso als Vorlage für Ableitungen dienen können. Wichtig ist, dass Ihre Klassen-Hierarchien immer von »allgemein« nach »spezialisiert« aufgebaut sind. Und manchmal sind die Basis-Klassen so allgemein, dass Objekte von Ihnen keinen Sinn machten. Das heißt nicht, dass mit Ihrer Klasse etwas nicht stimmt (oft bedeutet es genau das Gegenteil), sondern nur, dass diese Klassen als abstrakt deklariert werden sollten.

Listing 4.2: Abstrakte Klassen

```
<?php
abstract class KlasseVorlage
{
    public function ausgabe()
    {
        echo "Standard-Ausgabe!\n";
    }
}
class KlasseEins extends KlasseVorlage
{
    public function test()
    {
        $this->ausgabe();
    }
}
$Obj1 = new KlasseEins();
$Obj1->test();
```

```
//! dies wird nicht funktionieren:
$Obj2 = new KlasseVorlage();
?>
```

```
Standard-Ausgabe!

Fatal error: Cannot instantiate abstract class KlasseVorlage
in /htdocs/php5neu/abstract.php5 on line 19
```

4.1.2. Abstrakte Methoden

Häufig möchte man in einer abstrakten Klasse betimmte Methoden noch gar nicht implementieren. Man weiß zwar, dass man sie haben möchte, aber die konkrete Implementierung möchte man den ableitenden Klassen überlassen.

Stellen Sie sich zum Beispiel vor, dass Ihre abstrakte Klasse `Fahrzeuge` die Methode `fahren()` beinhalten soll. Aber in einer davon abgeleitete Klasse `Kfz` werden Sie diese Methode wahrscheinlich ganz anders implementieren als in einer Klasse `Fahrrad`. Und statt jetzt irgendeinen überflüssigen Dummy-Code in Ihre abstrakte Klasse einzufügen, können Sie in PHP 5 die Möglichkeit nutzen, Methoden als abstrakt zu kennzeichnen und die ableitenden Klassen dazu zwingen, eine passende Methode zu definieren.

Eine abstrakte Methode deklariert nur die Signatur[2] der Methode, enthält aber nicht ihre Implementierung. Diese muss zwingend in der ersten nicht als abstrakt deklarierten, ableitenden Klasse vorgenommen werden.

Eine Klasse, die mindestens eine abstrakte Methoden enthält, muss ebenfalls als abstrakt deklariert werden.

Listing 4.3: Abstrakte Methoden

```
1 <?php
2 abstract class KlasseVorlage
3 {
4     abstract public function ausgabe();
```

[2] Zur Signatur einer Methode gehört der Name (auch Bezeichner genannt) und die Parameterliste.

```
5  }
6  class KlasseEins extends KlasseVorlage
7  {
8      public function ausgabe()
9      {
10         echo "Ich bin in Klasse Eins!\n";
11     }
12 }
13 class KlasseZwei extends KlasseVorlage
14 {
15     public function test()
16     {
17         echo "Ich bin in Klasse Zwei!\n";
18     }
19 }
20 $Obj1 = new KlasseEins();
21 $Obj1->ausgabe();
22 //! KlasseZwei implementiert nicht die Methode ausgabe(),
23 //! daher wird dies nicht funktionieren:
24 $Obj2 = new KlasseZwei();
25 $Obj2->test();
26 ?>
```

```
Ich bin in Klasse Eins!

Fatal error: Cannot instantiate abstract class KlasseZwei in
/htdocs/php5neu/abstractmethod.php5 on line 24
```

Bitte beachten Sie, dass die Prüfung auf korrekte Implementierung der Methoden in der abgeleiteten Klasse erst zur Laufzeit stattfindet. Werden im letzten Listing die Zeilen 24 und 25 auskommentiert, dann wird das Skript ohne Murren ausgeführt.

4.2. Rien ne va plus: `final`

`final` stellt das Gegenteil von `abstract` dar: Während `abstract` ein Überschreiben von Methoden bzw. Ableiten von Klassen erzwingt, erreicht `final` genau das Gegenteil.[3]

[3] In z.B. Java ist es auch möglich, Eigenschaften als `final` zu kennzeichnen und damit vor Wertänderungen zu schützen. Dies ist in PHP 5 nicht vorgesehen. Stattdessen werden

4.2.1. Überschreiben von Methoden in ableitenden Klassen verhindern

Möchte man vermeiden, dass eine Methode in einer abgeleiteten Klasse überschrieben wird, dann ist die betreffende Methode als `final` zu kennzeichnen.

Listing 4.4: Eine Methode vor Überschreibung schützen

```
1  <?php
2  class KlasseEins
3  {
4      final public function ausgabe()
5      {
6          echo "In Klasse Eins\n";
7      }
8  }
9
10 class KlasseZwei extends KlasseEins
11 {
12     //! dies wird nicht kompilieren, da
13     //! ausgabe() in der Elter-Klasse als
14     //! final gekennzeichnet wurde:
15     public function ausgabe()
16     {
17         echo "In Klasse Zwei\n";
18     }
19 }
20 ?>
```

```
Fatal error: Cannot override final method
KlasseEins::ausgabe() in
/htdocs/php5neu/finalmethode.php5 on line 19
```

4.2.2. Eine Klasse vor Ableitung schützen

Kennzichnet man eine Klasse als `final`, dann ist es nicht mehr möglich, von dieser Klasse abzuleiten.[4]

aber Klassen-Konstanten eingeführt (siehe Abschnitt 5.2 auf Seite 41).

[4] In PHP ist der Vorteil von finalen Klassen (noch?) gering. In anderen objektorientierten Sprachen erhält man beim Einsatz final gekennzeichneter Klassen bisweilen einen deutli-

Listing 4.5: Eine Klasse vor Ableitung schützen

```
<?php
final class KlasseEins
{
    public function ausgabe()
    {
        echo "In Klasse Eins\n";
    }
}
//! dies wird nicht kompilieren, da
//! die Elter-Klasse als final
//! gekennzeichnet wurde:
class KlasseZwei extends KlasseEins
{
    public function ausgabe()
    {
        echo "In Klasse Zwei\n";
    }
}
?>
```

```
Fatal error: Class KlasseZwei may not inherit from final class
(KlasseEins) in /htdocs/php5neu/finalklasse.php5 on line 18
```

4.3. Das Versprechen: Interfaces

Eng verwandt mit abstrakten Klassen sind die Interfaces (Schnittstellen). Etwas vereinfacht kann man ein Interface als eine abstrakte Klasse mit ausschließlich abstrakten Methoden ansehen. Eine Klasse, die ein Interface einbaut (implementiert), verpflichtet sich, alle in dem Interface deklarierten (aber dort nicht definierten!) Methoden zu definieren. Ein Interface ist sozusagen ein Versprechen: »Wenn ich als Klasse ein Interface implementiere, dann verspreche ich, alle geforderten Methoden so zu bauen, dass sie erwartungsgemäß funtionieren und nutzbar sind!«

Der Name kommt ja nicht von ungefähr: Wenn Sie einen Computer kaufen, der mit einer USB-Schnittstelle (USB-*Interface*) beworben wird, dann erwarten Sie ja auch, dass dort Ihr USB-Speicher-Stift passt und ordnungsgemäß funktioniert. Übersetzt heißt das: Wenn eine Klasse ein Interface implementiert, dann

chen Geschwindigkeitsvorteil, weil Compiler den Code besser optimieren können.

können Sie erwarten, dass beliebiger, auf dieses Interface abgestimmter Code ohne Probleme funktioniert.

Meist werden Klassen durch Interfaces um irgendwelche speziellen, von außen nutzbare Fähigkeiten erweitert, was auch daran zu erkennen ist, dass die (englischen) Namen vieler Interfaces mit einem »-able« enden. In der SPL (der Standard-PHP-Library, die in Kapitel 10 auf Seite 60 beschrieben wird) ist zum Beispiel das Interface `Traversable` definiert, welches ein kontrolliertes Durchlaufen einer Klasse mit Hilfe der `foreach`-Schleife sicherstellt.

Das Schöne an den Interfaces ist, dass eine Klasse eine beliebige Zahl von ihnen implementieren darf und so um eine beliebige Menge an standarisierter Funktionalität erweitert werden kann.

Die Definition eines Interface gleicht der einer abstrakten Klasse mit ausschließlich abstrakten Methoden, nur das statt `class` das Schlüsselwort `interface` verwendet und auf das Schlüsselwort `abstract` verzichtet wird – letzteres steckt ja implizit schon in `interface`:

```
interface EinInterface
{
    public function meineInterfaceMethode();
    //...
}
```

Möchte eine Klasse ein Interface implementieren, dann wird der Klassenkopf nach dem Klassennamen (bzw. bei abgeleiteten Klassen nach dem Klassennamen der Elterklasse) um das Schlüsselwort `implements` und den Bezeichner des Interfaces erweitert. Bei Einsatz mehrerer Interfaces werden die Bezeichner durch Kommata getrennt aufgelistet.

```
class MeineKlasse extends DeineKlasse implements EinInterface,
    NochEinInterface
{
    //...
}
```

Dazu ein ausführlicheres Bespiel:

Listing 4.6: Implementation von Interfaces

```
<?php
interface Zuruecksetzbar
{
    public function zuruecksetzen();
}
interface Abfragbar
{
    public function abfrage();
}

class KlasseEins
{
    protected $a = 0;
    public function setzeA($wert)
    {
        $this->a = $wert;
    }
}
class KlasseZwei extends KlasseEins implements Zuruecksetzbar,
    Abfragbar
{
    public function zuruecksetzen()
    {
        $this->a = 0;
    }
    public function abfrage()
    {
        echo "a hat den Wert " . $this->a . "\n";
    }
}

$Objekt = new KlasseZwei();
$Objekt->abfrage();
$Objekt->setzeA(4711);
$Objekt->abfrage();
$Objekt->zuruecksetzen();
$Objekt->abfrage();
?>
```

```
a hat den Wert 0
a hat den Wert 4711
a hat den Wert 0
```

Interfaces werden häufig als ein Ersatz für Mehrfachvererbungen (bei der eine

Klasse von mehreren Elterklassen ableiten darf) betrachtet. Eine Mehrfachvererbung ist in PHP (aber zum Beispiel auch in Java und C#) nicht vorgesehen[5].

4.4. Der neue Operator `instanceof` (oder: Gehörst Du zur Familie?)

PHP ist eine schwach typisierte Sprache und wird es wohl auch – zumindest mittelfristig – bleiben. Bei einfachen Typen wie Integer, Float, String etc. hat man damit auch nur selten Probleme, denn was (PHP) nicht passt, wird (implizit zur Laufzeit) passend gemacht und entspricht danach meist sogar der Intention des Programmierers. Bei komplexen Typen (Arrays und Objekte) ist dies nicht möglich: Erwartet zum Beispiel ein Programm, dass ein Objekt ein bestimmte Methode bereitstellt, so gerät man in echte Schwierigkeiten, wenn dem nicht so ist. Daher ist es wichtig, prüfen zu können, ob ein Objekt von einem bestimmten Typ ist. Noch einmal zur Erinnerung: Ein Objekt kann nicht nur als Typ der eigenen Klasse, sondern auch als Typ jeder Vorfahr-Klasse angesehen werden. Und implementiert die eigene oder eine der Vorfahr-Klassen ein Interface, dann ist das Objekt auch vom Typ des Interfaces.

PHP 5 führt den Operator `instanceof` ein, der genau dieses prüft. Als Linkswert wird eine Objektvariable erwartet, als Rechtswert entweder eine Objektvariable oder eine Klassenbezeichnung, die – zumindest in der geteseten Version – direkt mit dem Bezeichner und nicht mit einer Zeichenkette, die den Bezeichner enthält, angegeben werden muss.

Ein Beispiel zur Verdeutlichung:

Listing 4.7: Beispiel: instanceof

```
1 <?php
2 interface Nonsens
3 {
4     public function nix();
```

[5] Beim Entwurf der Zend-Engine 2 wurde noch über die Implementation der Mehrfachvererbung nachgedacht, später aber zugunsten von Interfaces verworfen.

```
5  }
6  class KlasseEins
7  {
8      public $name = "Klasse Eins!\n";
9      public function ausgabe()
10     {
11         echo "Ich bin " . $this->name;
12     }
13 }
14 class KlasseZwei extends KlasseEins implements Nonsens
15 {
16     public $name = "Klasse Zwei!\n";
17     public function nix()
18     {
19         echo "Ich mache nix!\n";
20     }
21 }
22 class KlasseDrei extends KlasseZwei
23 {
24     public $name = "Klasse Drei!\n";
25 }
26 class AndereKlasse
27 {
28     public $name = "eine andere Klasse!\n";
29 }
30
31 $Objekt = new KlasseDrei();
32
33 if ( $Objekt instanceof KlasseDrei ) {
34     echo "Ich bin vom Typ 'Klasse Drei'\n";
35 }
36 if ( $Objekt instanceof KlasseZwei ) {
37     echo "Ich bin vom Typ 'Klasse Zwei'\n";
38 }
39 if ( $Objekt instanceof KlasseEins ) {
40     echo "Ich bin vom Typ 'Klasse Eins'\n";
41 }
42 if ( $Objekt instanceof Nonsens ) {
43     echo "Ich bin vom Typ 'Nonsens'\n";
44 }
45 if ( $Objekt instanceof AndereKlasse ) {
46     echo "Ich bin vom Typ 'AndereKlasse'\n";
47 } else {
48     echo "Ich bin _nicht_ vom Typ 'AndereKlasse'\n";
49 }
50 ?>
```

```
Ich bin vom Typ 'Klasse Drei'
Ich bin vom Typ 'Klasse Zwei'
Ich bin vom Typ 'Klasse Eins'
Ich bin vom Typ 'Nonsens'
Ich bin _nicht_ vom Typ 'AndereKlasse'
```

Mit dem Operator `instanceof` wird die erst in PHP 4.2 eingeführte Funktion `is_a()` hinfällig.

4.5. Klassentyp-Angabe bei Methodenparametern

Wenn Sie ein Objekt selbst erzeugen und anschließend damit arbeiten, dann kennen Sie den Typ und werden wohl keine Elemente aufrufen, die das Objekt nicht besitzt. Anders liegt der Fall schon bei Methoden[6], die Objekte als Parameter erwarten. Wird so einer Methode ein Objekt vom »falschen« Typ übergeben, dann gibt es natürlich Probleme, sobald auf ein Objekt-Element zugegriffen wird, dass dieses »unpassende« Objekt nicht bereitstellt. Daher ist es für eine Methode wichtig, die ihr übergebenen Objekte zu validieren. Dies kann mit dem in Abschnitt 4.4 auf Seite 34 vorgestellten Operator `instanceof` bewerkstelligt werden:

Listing 4.8: Klassentyp-Überprüfung eines Methodenparameters mit instanceof

```
public function erwarteObjektZwei($Objekt)
{
    if ( ! ( $Objekt instanceof KlasseZwei ) ) {
        die("Fataler Fehler: Parameter 1 muss muss vom Typ KlasseZwei
            sein! (Datei " . __FILE__ . " in Zeile " . __LINE__ . ")
            ");
    }
    //...
}
```

[6] Alles in diesem Abschnitt beschriebene gilt natürlich ebenso für Funktionen.

Da dies aber aufwändig und wenig elegant ist, gibt es in PHP 5 die Möglichkeit, den Objekttyp in der Signatur einer Methode anzugeben:

Listing 4.9: Klassentyp-Überprüfung eines Methodenparameters durch classtype-hinting

```
public function erwarteObjektZwei(KlasseZwei $Objekt)
{
    //...
}
```

Wird nun ein Objekt übergeben, dessen Typ nicht (auch) `KlasseZwei` ist, beendet sich das Skript mit folgender Meldung:

```
Fatal error: Argument 1 must be an instance of KlasseZwei in
/htdocs/php5neu/type_hinting.php5 on line 28
```

Diese Art der Typangabe (auch classtype-hinting genannt) in der Signatur einer Methode funktioniert in PHP 5 nur bei Objekten; mit anderen nativen Typen wie Integer, String etc. ist das nicht möglich. Außerdem wird die Typüberprüfung nicht während der Kompilation (wie bei stark typisierten Programmiersprachen üblich), sondern immer erst zur Laufzeit vorgenommen; somit enspricht der Code in Listing 4.9 tatsächlich dem Code in Listing 4.8 (abgesehen natürlich von dem unterschiedlichen Wortlaut der Fehlermeldung).

Noch ein ausführliches Beispiel:

Listing 4.10: Eine Klasse vor Ableitung schützen

```
1  <?php
2  interface Nonsens
3  {
4      public function nix();
5  }
6  class KlasseEins
7  {
8      public $name = "Klasse Eins!\n";
9      public function ausgabe()
10     {
11         echo "Ich bin " . $this->name;
12     }
13 }
14 class KlasseZwei extends KlasseEins implements Nonsens
15 {
```

```
16      public $name = "Klasse Zwei!\n";
17      public function nix()
18      {
19          echo "Ich mache nix!\n";
20      }
21  }
22  class KlasseDrei extends KlasseZwei
23  {
24      public $name = "Klasse Drei!\n";
25  }
26  class AndereKlasse
27  {
28      public function erwarteKlasseZwei(KlasseZwei $Objekt)
29      {
30          $Objekt->ausgabe();
31      }
32      public function erwarteInterfaceNonsens(Nonsens $Objekt)
33      {
34          $Objekt->nix();
35      }
36  }
37
38  $HauptObjekt     = new AndereKlasse();
39  $ParamObjektEins = new KlasseEins();
40  $ParamObjektZwei = new KlasseZwei();
41  $ParamObjektDrei = new KlasseDrei();
42
43  $HauptObjekt->erwarteKlasseZwei($ParamObjektZwei);
44  $HauptObjekt->erwarteInterfaceNonsens($ParamObjektZwei);
45  $HauptObjekt->erwarteKlasseZwei($ParamObjektDrei);
46  $HauptObjekt->erwarteInterfaceNonsens($ParamObjektDrei);
47  //! Dies gibt einen Fehler:
48  $HauptObjekt->erwarteKlasseZwei($ParamObjektEins);
49  ?>
```

```
Ich bin Klasse Zwei!
Ich mache nix!
Ich bin Klasse Drei!
Ich mache nix!

Fatal error: Argument 1 must be an instance of KlasseZwei in
/htdocs/php5neu/type_hinting.php5 on line 28
```

5. Die Schlüsselwörter `static` und `const`

5.1. Statische Klassen-Elemente

Wenn Sie ein Objekt mit `new` instanzieren oder mit `clone` kopieren, dann erzeugen Sie einen kompletten, unabhängigen Satz an Objekt-Eigenschaften (Variablen einer Instanz) im Speicher. Und mit Hilfe der Objekt-Methoden, die für jede Klasse – unabhängig von der Anzahl der von ihr instanzierten Objekte – nur einmal im Speicher existieren, können Sie dierekt auf diesen Objekt-Variablen arbeiten. Diese Eigenschaften »gehören« also dem Objekt.

5.1.1. Statische Eigenschaften

Es können aber auch Variablen erzeugt werden, die der Klasse »gehören« und die für jede Klasse genau einmal existieren. Diese Klassen-Variablenb bezeichnet man als statisch und deklariert sie mit dem Schlüsselwort `static`. Auf sie wird mit Hilfe des Sichtbarkeitsoperators (`::`) zugegriffen, wobei links der Klassenbezeichner oder das Schlüsselwort `self` (kennzeichnet die eigene Klasse) steht.

Listing 5.1: Beispiel: Statische Klassenvariable

```
<?php
class KlasseEins
{
    public static $counter = 0;
    public function erhoehenUndAusgabe()
    {
        ++self::$counter;
```

```
        echo "Der Klassenzähler steht auf " . self::$counter . "!\n";
    }
}

$ObjektA = new KlasseEins();
$ObjektB = new KlasseEins();

$ObjektA->erhoehenUndAusgabe();
$ObjektB->erhoehenUndAusgabe();
KlasseEins::$counter += 10;
$ObjektA->erhoehenUndAusgabe();
?>
```

```
Der Klassenzähler steht auf 1!
Der Klassenzähler steht auf 2!
Der Klassenzähler steht auf 13!
```

5.1.2. Statische Methoden

In PHP 5 kann man auch Methoden mit dem Schlüsselwort `static` als statisch deklarieren, was zur Folge hat, dass man sie ausschließlich als Element einer Klasse (und nicht als Element eines Instanz) aufrufen kann. Daher können statische Methoden mit der Variablen `$this`, die die eigene Instanz beinhaltet, nichts anfangen und nicht direkt auf Instanz-Elemente zugreifen– wohl aber auf staische Eigenschaften und Methoden. Der Zugriff aus statische Methoden erfolgt genau wie bei den statischen Eigenschaften über den Klassenbezeichner (bzw. `self` für die eigene Klasse) und den Sichtbarkeitsoperator.

Ein klassisches Beispiel für statische Klassenelemente ist das Singleton-Entwurfsmuster, welches eine statisch aufrufbare Methode bereitstellt, die beim erstmaligen Aufruf genau eine Instanz der eigenen Klasse erzeugt und immer nur diese eine Instanz zurückgibt[1]:

[1] Die Definition des Singleton-Entwurfsmusters laut Gamma et al: »Sichere ab, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.«

Listing 5.2: Beispiel für statische Klassenelemente: Das Singleton-Pattern

```
<?php
class Singleton
{
    static private $meineEinzigeInstanz = NULL;

    static public function get()
    {
        if ( self::$meineEinzigeInstanz == NULL ) {
            self::$meineEinzigeInstanz = new self();
        }
        return self::$meineEinzigeInstanz;
    }

    public function ausgabe()
    {
        echo "ich bin ein Singleton!";
    }
}

Singleton::get()->ausgabe();

?>
```

5.2. Klassen-Konstanten

In PHP 5 ist es mit dem Schlüsselwort `const` für jede Klasse möglich, eigene Konstanten zu definieren. Wie auch bei den globalen Konstanten hat der Bezeichner kein vorausgehendes Dollar-Zeichen und der Zugriff erfolgt (wie bei den statischen Elementen) immer mit Hilfe des Sichtbarkeitsoperators in der Form `Klassenname::Konstantenname`, in der Klasse, in der sie definiert sind, auch mit Hilfe des Schlüsselwortes `self` (`self::Konstantenname`).

Listing 5.3: Klassen-Konstanten

```
<?php
class KlasseEins
{
    const MEINE_KONST = "Ich bin konstant!";
    public function ausgabe()
    {
```

```
        echo KlasseEins::MEINE_KONST;
    }
}
$Obj = new KlasseEins();
$Obj->ausgabe();
?>
```

Eine Konstante muss immer initialisiert werden – alles andere wäre auch sinnlos, da spätere Zuweisungen nicht möglich sind. Dabei wird der Wert während der Kompilation ermittelt, also kommen nur Literale und konstante Ausdrücke in Frage. Ausdrücke, die auf Informationen zurückgreifen, die erst zur Laufzeit verfügbar sind, können nicht verwendet werden.

Die Sichtbarkeit einer Klassen-Konstante kann (noch?) nicht über die Modifizierer `public`, `protected` und `private` gesteuert werden; sie ist von überall sichtbar.

6. Sonstige objektorientierte Änderungen

6.1. Klassen automatisch laden mit `__autoload()`

Wenn Sie objektorientiert programmieren, dann werden Sie schnell merken (oder schon gemerkt haben), dass die Menge Ihrer Klassen zügig wächst. Und wenn Sie die (durchaus empfehlenswerte) Angewohnheit besitzen, jede Klasse in einer eigenen, nach dem Klassennamen bezeichneten Datei zu speichern, dann wird Ihnen sicher auch edine schier unendliche `include()`-Liste am Anfang eines Skriptes nicht unbekannt sein. Doch genau diese Liste können Sie jetzt – ein wenig Organisation und sinnvolle Benennung vorausgesetzt – durch die Definition einer einzigen (globalen) Funtion mit dem Namen `__autoload()` ersetzen.

`__autoload()` wird immer dann aufgerufen, wenn während der Ausführung des Skripts auf eine (noch) nicht definierte Klasse zugegriffen wird. Als ersten und alleinigen Parameter wird dieser Funktion der Bezeicher der fehlenden Klasse übergeben, und wenn `__autoload()` vernünftig implementiert und der Bezeichner aussagekräftig ist, dann wird der Quellcode für diese Klasse wie von Geisterhand inkludiert.

Die Bezeichner von Klassen sollten also mit (noch mehr) Bedacht ausgewählt werden und sich im Namen der Datei wiederfinden. Die vom PEAR praktizierte Art und Weise der Klassenbenennung, in der sich der Dateipfad im Klassennamen widerspiegelt (Ordner durch Unterstriche getrennt), ist eine gute und nachahmenswerte Idee. Also: Halten Sie die Ordnernamen kurz und vermeiden Sie allzu viele Ordner-Ebenen.

Datei: BSP/KlasseEins.php

```
<?php
class BSP_KlasseEins
{
    public function ausgabe()
    {
        echo "Ich bin Klasse Eins!\n";
    }
}
?>
```

Datei: BSP/KlasseZwei.php

```
<?php
class BSP_KlasseZwei
{
    public static function ausgabe()
    {
        echo "Ich bin Klasse Zwei!\n";
    }
}
?>
```

Listing 6.1: Beispiel: __autoload()

```
<?php
function __autoload( $bezeichner )
{
    if ( file_exists( str_replace("_", "/", $bezeichner) . ".php") ) {
        include_once( str_replace("_", "/", $bezeichner) . ".php" );
        return;
    }
    die ("Konnte Klasse " . $bezeichner . " nicht laden!");
}

$ObjektEins = new BSP_KlasseEins();
$ObjektEins->ausgabe();
BSP_KlasseZwei::ausgabe();
?>
```

```
Ich bin Klasse Eins!
Ich bin Klasse Zwei!
```

6.2. Die Pseudo-Konstante __METHOD__

Zu den altbekannten Pseudo-Konstanten __FILE__ und __LINE__ sowie den neueren __CLASS__ und __FUNCTION__ (sie wurden in PHP 4.3 eingeführt) gesellt sich nun __METHOD__. Wird sie innerhalb eines Klassen-Kontexts aufgerufen, enthält sie den Namen der Klasse und der Methode (entspricht also __CLASS__."::".__FUNCTION__). In einer Funktion außerhalb einer Klasse enthält sie den Funktionsnamen (ist dort identisch mit __FUNCTION__).

Listing 6.2: Beispiel: __METHOD__

```
<?php
class KlasseEins
{
    public function meineMethode()
    {
        echo "__FILE__     : " . __FILE__ . "\n";
        echo "__LINE__     : " . __LINE__ . "\n";
        echo "__CLASS__    : " . __CLASS__ . "\n";
        echo "__FUNCTION__ : " . __FUNCTION__ . "\n";
        echo "__METHOD__   : " . __METHOD__ . "\n\n";
    }
}

function meineFunction()
{
    echo "__FILE__     : " . __FILE__ . "\n";
    echo "__LINE__     : " . __LINE__ . "\n";
    echo "__CLASS__    : " . __CLASS__ . "\n";
    echo "__FUNCTION__ : " . __FUNCTION__ . "\n";
    echo "__METHOD__   : " . __METHOD__ . "\n\n";
}

echo "__FILE__     : " . __FILE__ . "\n";
echo "__LINE__     : " . __LINE__ . "\n";
echo "__CLASS__    : " . __CLASS__ . "\n";
echo "__FUNCTION__ : " . __FUNCTION__ . "\n";
echo "__METHOD__   : " . __METHOD__ . "\n\n";

$Obj = new KlasseEins();
$Obj->meineMethode();

meineFunction();
?>
```

```
__FILE__      : /htdocs/php5neu/__METHOD__.php5
__LINE__      : 24
__CLASS__     :
__FUNCTION__  :
__METHOD__    :

__FILE__      : /htdocs/php5neu/__METHOD__.php5
__LINE__      : 7
__CLASS__     : KlasseEins
__FUNCTION__  : meineMethode
__METHOD__    : KlasseEins::meineMethode

__FILE__      : /htdocs/php5neu/__METHOD__.php5
__LINE__      : 17
__CLASS__     :
__FUNCTION__  : meineFunction
__METHOD__    : meineFunction
```

7. Catch me if you can: Ausnahmen erzeugen und abfangen

In diesem Kapitel wird gezeigt, auf welche Weise in PHP 5 Fehler abgefangen werden. Ach – Sie machen gar keine Fehler? Nun, bevor Sie jetzt empört weiterblättern (denn dieses Kapitel ist ja offensichtlich nur für doofe Anfänger), lassen Sie mich den vorigen Satz anders formulieren: In diesem Kapitel wird gezeigt, wie man in PHP 5 seinen Code auf mögliche Probleme vorbereiten und gegebenfalls darauf reagieren lässt. Ein solches Problem kann zum Beispiel eine nicht lesbare Datei (»Welcher Trottel hat die Rechte geändert?«), ein fehlerhaftes Passwort beim Datenbank-Connect oder ein zickiger Mailserver sein.

7.1. Fehler in PHP 4 abfangen

In PHP 4 musste man jedem Funtions-Aufruf durch ein vorangestelltes @-Zeichen die Fehlerausgabe abgewöhnen und penibel prüfen, ob der Rüchgabewert einen Fehler anzeigt. Das sah dann häufig so aus:

Listing 7.1: Fehler in PHP 4 abfangen (Beipiel 1)

```
1 <?php
2 if ( ! ($fh = @fopen("meineDatei.txt", "r")) ) {
3     die("konnte Datei nicht öffnen!");
4 }
5 if ( ($text=fread($fh, 1024)) === false ) {
6     die("konnte Datei nicht lesen");
7 }
8 @fclose($fh);
9 echo $text;
```

```
10 ?>
```

In Zeile 5 benötigen Sie übrigens das dreifache Gleichheitszeichen, um den Fall einer leeren Datei abzufangen, denn auch dann hat ja `fread()` erfolgreich gelesen – wenn auch nur ein EOF.

Wollte man nach einem sochen Fehler weiteren Code ausführen lassen, dann führte das in PHP 4 zu `if-else`-Kaskaden oder zu folgendem seltsamen Konstrukt:

Listing 7.2: Fehler in PHP 4 abfangen (Beipiel 2)

```
1  <?php
2  $error = "";
3  switch (TRUE) { default:
4      if ( ! ($fh = @fopen("meineDatei.txt", "r")) ) {
5          $error = "konnte Datei nicht öffnen!";
6          break;
7      }
8      if ( ($text=fread($fh, 1024)) === false ) {
9          $error = "konnte Datei nicht lesen";
10         break;
11     }
12     @fclose($fh);
13 }
14 if ($error) {
15     echo $error;
16     $error = "";
17 } else {
18     echo $text;
19 }
20 echo "weiterer Code...";
21 ?>
```

7.2. Das Ausnahme-Modell von PHP 5

PHP 5 führt ein Ausnahme-Modell ein, dass vergleichbar mit dem anderer objektorientierter Programmiersprachen ist: Auszuführender Code kann in einem `try`-Block gekapselt werden. Wird nun innerhalb dieses Blocks mit Hilfe des Schlüsselworts `throw` ein Objekt »geworfen«, dann wird der restliche Code des

Blocks übersprungen und Code in einem anschließenden, mit `catch` gekennzeichneten und zum geworfenden Objekt passenden Block ausgeführt. Puh..., ein Beipiel wird das hoffentlich klarer machen:

Listing 7.3: Das Ausnahme-Modell von PHP 5

```
<?php
try {
    if ( ! ($fh = @fopen("meineDatei.txt", "r")) ) {
        throw new Exception("konnte Datei nicht öffnen!");
    }
    if ( ($text=fread($fh, 1024)) === false ) {
        throw new Exception("konnte Datei nicht lesen");
    }
    @fclose($fh);
    echo $text;
} catch (Exception $e) {
    echo $e->getMessage();
}
echo "weiterer Code...";
?>
```

Es spielt keine Rolle, in welche Ebene innerhalb des `try`-Blocks eine Ausnahme geworfen wird – gerade dadurch enthält dieses Konzept seine Mächtigkeit. Im folgenden Beipiel wird das Ausnahme-Objekt innerhalb einer Funktion geworfen, aber nicht der Code der Funktion ist durch `try-catch` gekapselt, sondern der Aufruf der Funktion:

Listing 7.4: Ausnahmen in Funktionen werfen

```
<?php
function leseDatei( $dateiname )
{
    if ( ! ($fh = @fopen($dateiname, "r")) ) {
        throw new Exception("konnte Datei ".$dateiname." nicht
            öffnen!");
    }
    if ( ($text=fread($fh, 1024)) === false ) {
        throw new Exception("konnte Datei ".$dateiname." nicht
            lesen");
    }
    @fclose($fh);
    return $text;
}
```

```
13 try {
14     echo leseDatei("MeineDatei.txt");
15 } catch (Exception $e) {
16     echo $e->getMessage();
17 }
18 echo "weiterer Code...";
19 ?>
```

Wird ein Ausnahme-Objekt geworfen, dann »läuft« dieses Objekt so lange die Ebenen hoch, bis es durch ein passendes `catch` abgefangen wird. Wichtig ist nur, *dass* es abgefangen wird, denn sonst kann es zu einer unschönen Fehlermeldung kommen:

Listing 7.5: Geworfene Ausnahmen müssen abgefangen werden

```
1  <?php
2  function leseDatei( $dateiname )
3  {
4      if ( ! ($fh = @fopen($dateiname, "r")) ) {
5          throw new Exception("konnte Datei ".$dateiname." nicht
               öffnen!");
6      }
7      if ( ($text=fread($fh, 1024)) === false ) {
8          throw new Exception("konnte Datei ".$dateiname." nicht
               lesen");
9      }
10     @fclose($fh);
11     return $text;
12 }
13 //! DeineDatei.txt gibt es nicht!
14 echo leseDatei("DeineDatei.txt");
15 ?>
```

```
Fatal error: Uncaught exception 'Exception' with message 'konnte Datei
DeineDatei.txt nicht öffnen!' in /htdocs/php5neu/errorHandling3.php5:5
Stack trace:
#0 /htdocs/php5neu/errorHandling3.php5(14): leseDatei('DeineDatei.txt')
#1 {main}
  thrown in /htdocs/php5neu/errorHandling3.php5 on line 5
```

7.3. Die Mutter(-Klasse) aller Ausnahmen: `Exception`

(TODO)

7.4. Eigene Ausnahme-Klassen erzeugen

(TODO)

8. Weitere Änderungen der Syntax

8.1. Referenzen in der `foreach`-Schleife

```
Array
(
    [DE] => DE: Berlin
    [FR] => FR: Paris
    [GB] => GB: London
)
```

In PHP 4 konnte man den Umweg über das Array-Element – direkt über den Index angesprochen – gehen, um den gleichen Effekt zu erzielen:

Listing 8.1: Änderung von Werten in der foreach-Schleife unter PHP 4

```
<?php
$array = array(
    "DE" => "Berlin",
    "FR" => "Paris",
    "GB" => "London"
);
foreach ($array as $key => $val) {
    $array[$key] = $key . ": " . $val;
}
print_r($array);
?>
```

8.2. Defaultwerte für Funktionsparametern, die als Referenz übegeben werden

Es war in PHP auch bislang schon möglich, die Argumente einer Funktion bzw. Methode mit Standardwerten zu belegen. Dieses Feature war und ist auch umso bedeutsamer, da PHP das Überladen [1] von Funktionen nicht unterstützt und wohl auch in Zukunft nicht unterstützen wird. In PHP 4 konnten aber nur Argument-Variablen, die den Wert als Kopie übernahmen, mit diesen Defaultwerten belegt werden. Argument-Variablen, die nur eine Referenz auf die übergebende Variable speicherten, blieben außen vor. Dies war auch insofern einleuchtend, da es ohne übergebende Variable ja nichts gibt, was referenziert werden kann. PHP 5 zaubert nun im Hintergrund und bietet auch für Funktionsparametern, die als Referenz übegeben werden, die Möglichkeit von Standardwerten. Da Objekte in PHP 5 aber bei solchen Gelegenheiten sowieso nicht mehr kopiert werden (siehe Abschnitt 1.1 auf Seite 2), ist dieses Feature nicht mehr so wichtig wie es in PHP 4 noch gewesen wäre.

Listing 8.2: Sichtbarkeit von Methoden

```
1  <?php
2  function testFunktion( &$testParam = "" )
3  {
4      if ( $testParam == "" ) {
5          echo "Bitte geben Sie einen Text an.\n";
6      } else {
7          echo "Sie gaben an: " . $testParam;
8      }
9  }
10 testFunktion();
11 $a = "Ein Beispiel-Text.";
12 testFunktion($a);
13 //! dies funktioniert nicht:
14 //! testFunktion("Ein anderer Text!");
15 ?>
```

[1] Überladen von Funktionen heißt, dass es mehrere Funktionen mit gleichem Namen gibt, die sich in ihrer Signatur unterscheiden. Je nach übergebender Parameter wird die richtige Funktion aufgerufen. Da PHP nur eine schwach typisierte Sprache ist, wäre ein solches Überladen kaum sinnvoll zu implementieren.
Das Überladen von Funktionen darf nicht mit dem Überschreiben in abgeleiteten Klassen verwechselt werden.

```
Bitte geben Sie einen Text an.
Sie gaben an: Ein Beispiel-Text.
```

Es ist aber weiterhin nicht möglich, Literale als Parameter zu übergeben, wie man in Zeile 14 sehen kann (sofern man sie einkommentiert...):

```
Fatal error: Only variables can be passed by reference in
/htdocs/php5neu/defaultValue.php5 on line 14
```

8.3. Das Error-Reporting-Level `E_STRICT`

PHP 5 führt die Error-Reporting-Level-Konstante `E_STRICT` ein, die dem Dezimalwert 2048 entspricht. Diese Konstante kann zum Beispiel in der `php.ini` der Einstellung `error_reporting` zugewiesen werden, um eine noch genauere Analyse des PHP-Codes zu erreichen. Die Hinweise gehen dann über die mit der Einstellung `E_ALL` ausgegebenen Hinweise hinaus; so wird zum Beispiel auf veraltete Konstrukte hingewiesen und PHP versucht, dem Programmierer beratend zur Seite zu stehen, um den Code so zukunftssicher wie möglich zu machen. Folgendes Listing

Listing 8.3: Beispiel: E_STRICT

```
<?php
class KlasseAlt
{
    var $zaehler;
}
$Objekt = new KlasseAlt();
?>
```

führt zu folgender Ausgabe:

```
Strict Standards: var: Deprecated. Please use the
public/private/protected modifiers in
/htdocs/php5neu/E_STRICT.php5 on line 4
```

9. Reflection-API

Endlich ist es in PHP möglich, Informationen über die verwendeten Funktionen, Klassen und Klassenelemente sowie Extensions abzufragen. Dabei spielt es keine Rolle, ob diese Elemente benutzerdefiniert oder standardmäßig eingebaut sind.

Es gab zwar schon seit längerem Funktionen wie `get_class_methods()`, `get_parent_class()` etc., aber nur mit einer strukturierten, einfach zu benutzenenden und durchdachten Schnittstelle lassen sich auch die neuen Fähigkeiten von PHP 5 vernünftig abbilden. Diese bietet das Relection-API.

9.1. Schnellübersicht mit `Reflector::export`

Wenn Sie einen schnellen Überblick über eine Funktion, Klasse oder Extension gewinnen möchten, dann reicht eine Zeile Code, wie das folgende Beispiel zeigt, dass Informationen über die Klasse `Exception` anzeigt:

Listing 9.1: Übersicht über eine Klasse

```
<?php
Reflection::export( new ReflectionClass("Exception") );
?>
```

```
Class [ <internal> class Exception ] {

  - Constants [0] {
  }

  - Static properties [0] {
  }
```

```
  - Static methods [0] {
  }

  - Properties [6] {
    Property [ <default> protected $message ]
    Property [ <default> private $string ]
    Property [ <default> protected $code ]
    Property [ <default> protected $file ]
    Property [ <default> protected $line ]
    Property [ <default> private $trace ]
  }

  - Methods [9] {
    Method [ <internal> final private method __clone ] {
    }

    Method [ <internal> <ctor> method __construct ] {
    }

    Method [ <internal> final public method getMessage ] {
    }

    Method [ <internal> final public method getCode ] {
    }

    Method [ <internal> final public method getFile ] {
    }

    Method [ <internal> final public method getLine ] {
    }

    Method [ <internal> final public method getTrace ] {
    }

    Method [ <internal> final public method getTraceAsString ] {
    }

    Method [ <internal> public method __toString ] {
    }
  }
}
```

Wie Sie sehen können, erhalten Sie nicht nur die Bezeichner der Elemente, sondern auch die Sichtbarkeits- und andere Modifizierer.

Um Informationen zu einer Funktion zu erhalten, müssen Sie ein Objekt der Klasse `RelectionFunktion` erzeugen und übergeben:

Listing 9.2: Übersicht über eine Funktion

```
<?php

/**
 * Sinnlose Beispielfuntion: Erhöht eine Ganzzahl um 1.
 *
 * @param  integer
 * @return integer
 */
function erhoehe($a)
{
    return $a+1;
}

Reflection::export( new ReflectionFunction("erhoehe") );

?>
```

```
/**
 * Sinnlose Beispielfuntion: Erhöht eine Ganzzahl um 1.
 *
 * @param  integer
 * @return integer
 */
Function [ <user> function erhoehe ] {
  @@ /htdocs/php5neu/Reflector_ReflectionFunction.php5 9 - 12

  - Parameters [1] {
    Parameter #0 [ $a ]
  }
}
```

Wow! Bei benutzerdefinerten Elementen werden also zusätzlich die Datei und die Zeilen angegeben, in der das Element definiert wurde. Und existiert eine Dokumentation im DocComment-Format[1], dann wird diese auch ausgegeben. (Wenn Sie's nicht eh schon machen, ist dies ein verdammt guter Zeitpunkt, um Ihren Code so zu dokumentieren!)

[1] Das DocComment-Format ... (TODO)

Extensions werden analog abgefragt:

Listing 9.3: Übersicht über eine Extension

```
<?php
Reflection::export( new ReflectionExtension("zlib") );
?>
```

Die führt zu folgender Ausgabe, die hier nur gekürzt wiedergegeben wird:

```
Extension [ <persistent> extension #15 zlib version 1.1 ] {

 - INI {
    Entry [ zlib.output_compression <ALL> ]
      Current = ''
    }
    Entry [ zlib.output_compression_level <ALL> ]
      Current = '-1'
    }
    Entry [ zlib.output_handler <ALL> ]
      Current = ''
    }
  }

 - Functions {
    Function [ <internal> public function readgzfile ] {
    }
    Function [ <internal> public function gzrewind ] {
    }
    Function [ <internal> public function gzclose ] {
    }

    ...
```

9.2. Detailierte Informationen über eine ...

9.2.1. ... Klasse

Der oben gezeigte Weg, Informationen über ein Element mit `Reflector::export()` auszugeben, ist für eine Schnellübersicht gut geeig-

net. Möchte man jedoch gezielt Informationen (nicht zuletzt, um sie während der Laufzeit zu verwenden), ...

(TODO)

9.2.2. ... Funktion

(TODO)

9.2.3. ... Extension

(TODO)

10. Standard PHP Library (SPL)

Die Standard PHP Library (SPL) stellt einen Satz an Klassen und Interfaces bereit, die das Leben mit Datenstrukturen in PHP vereinfachen sollen (und es auch tun). Die SPL war bislang nur im PECL zu finden und wurde mit der neuen PHP-Version in den Stand eines »regulären«, standardmäßig einkompilierten Moduls erhoben.

SPL support	enabled
Interfaces	RecursiveIterator, SeekableIterator
Classes	ArrayObject, ArrayIterator, CachingIterator, CachingRecursiveIterator, DirectoryIterator, FilterIterator, LimitIterator, ParentIterator, RecursiveDirectoryIterator, RecursiveIteratorIterator, SimpleXMLIterator

Abbildung 10.1.: Standard PHP Library: Auschnitt aus der php_info()-Ausgabe

10.1. Datenstrukturen als `Iterator` gekapselt

Zentrales Element der SPL, deren Namen wohl nicht von ungefähr an die in C++ genutzte STL (Standard Template Library) erinnert, ist das Interface `Iterator`. `Iterator` selbst implementiert das Interface `Traversable`, ein leeres Interface, welches anzeigt, dass ein Objekt mit Hilfe der `foreach`-Schleife gezielt[1] durchlaufen werden kann.

Klassen, die das Interface `Iterator` implementieren, müssen folgenden Methoden definieren:

[1] Nicht nur Arrays, auch beliebige Objekte können mit `foreach` durchlaufen werden, wobei die Eigenschaften als Werte zurückgegeben werden. Auf dieses *generelle* Durchlaufen kann jedoch nicht weiter Einfluss genommen werden.

```
mixed current()
```

Gibt den aktuellen (durch den internen Zeiger gekennzeichneten) Wert zurück.

```
mixed key()
```

Diese Methode gibt den aktuellen Schlüssel zurück.

```
void next()
```

Setzt den internen Zeiger auf das nächste Element.

```
void rewind()
```

Setzt den Zeiger auf das erste Element.

```
boolean valid()
```

Zeigt an, ob es einen »aktuellen« Wert gibt.

Es folgt ein Beispiel, in dem eine Klasse das Interface `Iterator` implmentiert. Von dieser Klasse abgeleitete Objekte können wie Arrays durchlaufen werden können, obwohl weder Arrays noch verschiedene Eigenschaften im Spiel sind:

Listing 10.1: Beispiel: Iterator

```
<?php
class KlasseEins implements Iterator
{
    private $i = 0;
    public function current()
    {
        return $this->i*$this->i;
    }
    public function key()
    {
        return $this->i;
    }
    public function next()
    {
        ++$this->i;
    }
    public function rewind()
    {
```

```
        $this->i = 0;
    }
    public function valid()
    {
        return ($this->i<5) ? TRUE : FALSE;
    }
}

$Objekt = new KlasseEins();
foreach ($Objekt as $key => $value) {
    echo $key . ": " . $value . "\n";
}
?>
```

```
0: 0
1: 1
2: 4
3: 9
4: 16
```

10.2. Ein Verzeichnis durchlaufen: `DirectoryIterator`

In der SPL existieren aber auch schon fertige Klassen, die das Interface `Iterator` implementieren. Eine interessante Klasse ist `DirectoryIterator`, mit der die Einträge eines Verzeichnisses durchlaufen werden können:

Listing 10.2: Beispiel: DirectoryIterator

```
<?php
$DIter = new DirectoryIterator(".");
foreach ($DIter as $key => $file) {
    echo $key . ": " . $file . "\n";
}
?>
```

```
0: .
1: ..
```

```
2: buch_php5_neues.php5
3: info.php
4: info.php5
5: phpMyAdmin
```

Interessant ist, dass `current()` ein Objekt vom Typ `DirectoryItrator` zurückgibt – und genau zu sein: Es ist das Objekt selbst, also `$this`. Dass man trotzdem den Dateinamen erhält, liegt an einem Zauber im Hintergrund, der immer dann den Verzeichniseintrag als Zeichenkette zurückgibt, sobald das Objekt (implizit oder explizit) zum String gewandelt wird.[2]

Die Klasse wäre aber nicht wirklich brauchbar, würde sie nicht noch einige andere Methoden implementieren, über die man weitere Informationen erfahren könnte:

Listing 10.3: Beispiel 2: DirectoryIterator

```
<?php
$DIter = new DirectoryIterator(".");
foreach ($DIter as $key => $file) {
    echo $key . ": " . $file;
    if ( $file->isDir() ) {
        echo " (Ordner, ";
    } else {
        echo " (" . $file->getSize() . " Byte, ";
    }
    echo date("Y-m-d", $file->getCTime()) . ")\n";
}
?>
```

```
0: . (Ordner, 2003-01-07)
1: .. (Ordner, 2003-01-01)
2: buch_php5_neues.php5 (2197 Byte, 2004-06-04)
3: info.php (21 Byte, 2003-01-06)
4: info.php5 (21 Byte, 2003-08-23)
5: phpMyAdmin (Ordner, 2003-01-07)
```

[2] Tja, dieser Zauber sollte in PHP 5 eigentlich allen Klassen zur Verfügung stehen. Es wurde eine Methode `__toString()` festgelegt, die immer dann zum Einsatz kommen sollte, wenn ein Objekt zum String gecastet wird. Leider gab's Probleme zusammen mit der internen Struktur von PHP, und daher wurde dieses Feature erst einmal wieder gestrichen. Wenn man den Entwicklern Glauben schenken darf, dann wird dies aber in einer zukünftigen Version (PHP 5.1?) implementiert.

Hier sieht man, wie wichtig es ist, dass `current()` das Objekt an sich zurückgibt. Es gibt noch eine ganze Reihe weiterer Methoden des `DirectoryItrator`-Objekts, die weitgehend selbsterklärend sind und die Sie bitte der PHP-Hilfe entnehmen.

10.3. Rekursiv Unterverzeichnisse durchlaufen

Die Klasse `RecursiveDirectoryIterator` ist abgeleitet von `DirectoryIterator` und führt folgende zwei Methoden ein:

```
boolean hasChildren()
```

Zeigt an, ob der aktuelle Wert ein Verzeichnis ist, wobei das übergeordnete Verzeichis (..) und das eigene Verzeichnis (.) ignoriert werden.

```
RecursiveDirectoryIterator getChildren()
```

Gibt ein Objekt vom Typ `RecursiveDirectoryIterator` zurück, welches das Unterverzeichnis repräsentiert.

Zur angenehmen Anwendung eines Objekts einer Klasse, dass das Interface `RecursivIterator` implementiert (wie es unter anderem `RecursiveDirectoryIterator` macht), stellt die SPL die Klasse `RecursiveIteratorIterator` bereit, die ein solches Objekt kapselt und es ermöglicht, mit einer einzigen `foreach`-Schleife rekursive alle Einträge zu durchlaufen.

Listing 10.4: Beispiel: RecursiveDirectoryIterator

```
<?php
$Iter = new RecursiveIteratorIterator(
            new RecursiveDirectoryIterator(".") );
foreach ($Iter as $key => $file) {
    echo $key . ": " . $file;
    echo " (" . $file->getSize() . " Byte, ";
    echo date("Y-m-d", $file->getCTime()) . ")\n";
}
?>
```

Dies führt zu folgender, wegen des Umfangs von PhpMyAdmin stark gekürzter Ausgabe:

```
./buch_php5_neues.php5: buch_php5_neues.php5 (2197 Byte, 2004-06-04)
./info.php: info.php (21 Byte, 2003-01-06)
./info.php5: info.php5 (21 Byte, 2003-08-23)
./phpMyAdmin/.cvsignore: .cvsignore (48 Byte, 2003-01-07)
./phpMyAdmin/ANNOUNCE.txt: ANNOUNCE.txt (6182 Byte, 2003-01-07)
```

Interessant ist, dass der Schlüssel nun den komplette Pfad enthält[3], während im Wert der Dateiname gespeichert wird.

10.4. Filtern: `FilterIterator`

Die abstrakte Klasse `FilterIterator` kapselt ein beliebiges Objekt vom Typ `Iterator`. In einer von ihr abgeleiteten Klasse muss die abstrakte Methode `accept()` definiert werden, die entscheidet, ob ein Element herausgefiltert oder akzeptiert wird. Im folgenden Beipiel wird ein solcher FilerIterator auf einen RecursiveDirectoryIterator angewendet:

Listing 10.5: Beispiel: FilterIterator

```
<?php
class MyDirectoryFilterIterator extends FilterIterator
{
    protected $filterString;

    function __construct($path, $filterString)
    {
        $this->filterString = $filterString;
        parent::__construct(
            new RecursiveIteratorIterator(
                new RecursiveDirectoryIterator($path) ) );
    }

    function accept()
    {
```

[3] Im Gegensatz zu einem Array kann also bei einem Iterator-Objekt der Schlüssel auch Punkte enthalten.

```
        return stristr($this->current(), $this->filterString);
    }
}

$Iter = new MyDirectoryFilterIterator(".", "my");
foreach ($Iter as $key => $file) {
    echo $key . ": " . $file;
    echo " (" . $file->getSize() . " Byte, ";
    echo date("Y-m-d", $file->getCTime()) . ")\n";
}
?>
```

```
./phpMyAdmin/libraries/mysql_wrappers.lib.php: mysql_wrappers.lib.php
(4203 Byte, 2003-01-07)
```

Wenn Sie sich wundern, dass nur ein einziger Eintrag ausgegeben wird, obwohl der Ordner »php**my**admin« heißt, dann bedenken Sie, dass im Beipiel in der Methode `accept()` die gesuchte Zeichenkette mit dem Wert von `current()` verglichen wird, der gesammte Pfad aber im Schlüssel (durch `key()` abfragbar) steckt.

10.5. Ein Array zu einem Iterator-Objekt machen

Natürlich ist es auch möglich, ein Array zu einem Objekt zu kapseln, um daraus ein Iterator-Objekt zu gewinnen. Dazu erzeugt man ein neues Objekt der Klasse `ArrayObject` und gibt als Parameter das Array an. Man kann statt eines Arrays übrigens auch ein beliebiges Objekt (oder, wie die die SPL-Dokumentation sagt, alles, was eine Hash-Tabelle hat) übergeben.

Mit der Methode `getIterator()` ist es möglich, aus dem Array-Objekt einen Iterator zu gewinnen und diesen weiter zu verwenden – zum Beispiel, um ihn zu filtern wie in Abschnitt 10.4 auf der vorherigen Seite beschrieben.

Im folgenden Beispiel wird die Anwendung eines Array-Objekts vorgestellt.

Listing 10.6: Beispiel: ArrayObject

```
1  <?php
2  $arr = array(
3      "de" => "Berlin",
4      "fr" => "Paris"
5  );
6  $ArrObj = new ArrayObject( $arr );
7  foreach ($ArrObj as $key => $file) {
8      echo $key . ": " . $file . "\n";
9  }
10 $ArrObj->append("London");
11 $ArrObj["nl"] = "Amsterdam";
12 $Iter = $ArrObj->getIterator();
13 foreach ($Iter as $key => $file) {
14     echo $key . ": " . $file . "\n";
15 }
16 echo "Anzahl: ".$ArrObj->count()."\n";
17 echo $ArrObj->offsetGet("de")."\n";
18 echo $ArrObj["fr"]."\n";
19 ?>
```

```
de: Berlin
fr: Paris
de: Berlin
fr: Paris
0: London
nl: Amsterdam
Anzahl: 4
Berlin
Paris
```

Wie man in den Zeilen 11 und 18 sieht, besteht die Möglichkeit, mit dem Indizierungs-Operator ([]) auf Werte des Array-Objekts zuzugreifen oder Elemente zu ändern oder hinzuzufügen. Dies scheint sogar mehr Sinn zu machen als die dafür vorgesehenen Methoden (wie zum Beispiel `append()` oder `offsetGet()`), da nur so Schlüsselwerte kontrolliert werden können.

11. XML

XML wurde eine zeitlang so gehyped, dass man es für eine – in der Programmierwelt gar nicht so seltene – Mode halten konnte, doch inzwischen ist es mehr als nur »in«: XML ist heute ein allgemein akzeptierter und vor allen Dingen genutzter Standard.

Bislang war XML in PHP zwar vorhanden, aber richtig glücklich wurde man damit nicht: SAX (Simple API for XML) gab es schon in PHP 3, aber viel mehr als einfaches Parsen war nicht möglich. In PHP 4 kam DOM (Document Object Model) hinzu, aber erst mit PHP 4.3, in der das Interface komplett überarbeitet wurde, war es ernsthaft nutzbar. XSLT-Unterstützung existierte auch (baserend auf Sablotron), war aber standardmäßig nicht aktiviert und wenn doch, stolperte man ständig über irgendein seltsames Verhalten.

libXML support	active
libXML Version	2.6.9
libXML streams	enabled

Abbildung 11.1.: Die komplette XML-Funktionalität beruht nun auf der libxml2: Ausschnitt aus der php_info()-Ausgabe

XML wurde in PHP 5 nicht nur entrümpelt, sondern in weiten Teilen komplett abgerissen und neu aufgebaut. Statt mehrerer zugrunde liegender Bibliotheken stützt man sich jetzt ausschließlich auf die libxml2-Bibliothek des GNOME-Projekts. Als Folge wird die Kompilation und Pflege des PHP-Source-Codes einfacher und Dokumente können intern ohne performanceschluckende Konvertierung zwischen den Modulen ausgetauscht werden.

- SAX gibt es weitehin und alle Funktionen sehen unverändert aus, aber intern baut es nicht mehr auf der expat-, sondern auf der libxml2-Bibliothek auf. Es wird hier nicht weiter vorgestellt.

- Simple-XML ist das »kleine Besteck«, wenn es um das Lesen von XML-Dokumenten geht: Einfach in der Anwendung und für viele Zwecke völlig ausreichend (siehe Kapitel 13 auf Seite 77).
- DOM ist der standarisierte Weg, um Dokumente zu lesen, zu ändern oder zu erstellen. Unterstützt auch XPath und XPointer. Sehr mächtig, bisweilen aber auch mächtig kompliziert (siehe Kapitel 12 auf der nächsten Seite).
- XSLT, die Sprache zum Transformieren von XML-Dokumenten, wird nun von einer auf der libxml2-basierenden Erweiterung unterstützt (siehe Abschnitt 12.5 auf Seite 73). Die alte XSLT-Erweiterung, die auf der Sablotron-Bibliothek beruhte, wird nicht mehr unterstützt.
- SOAP, der Standard für Webdienste, wird sowohl für Client- als auch Serveranwendungen nativ unterstützt, so dass man nicht mehr auf langsamere, in PHP erstellte Bibliotheken zurückgreifen muss (siehe Kapitel 14 auf Seite 87).

11.1. Beispiel XML

Als Beispiel-XML wird in den nächsten Kapiteln folgende kleine Datei herhalten:

Listing 11.1: Beispiel-XML-Dokument

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<DVDArchiv>
    <DVD Disczahl="2" id="DVD1">
        <Titel>Fight Club</Titel>
        <Regisseur>David Fincher</Regisseur>
    </DVD>
    <DVD Disczahl="1" id="DVD2">
        <Titel>Familienfest und andere Schwierigkeiten</Titel>
        <Regisseur>Jodie Foster</Regisseur>
    </DVD>
    <DVD Disczahl="2" id="DVD3">
        <Titel>Catch me if you can</Titel>
        <Regisseur>Steven Spielberg</Regisseur>
    </DVD>
</DVDArchiv>
```

12. DOM

Das DOM (Document Objekt Model) ist eine Empfehlung des World-Wide-Web-Konsortiums (W3C), wie ein XML-Dokument analysiert und bearbeitet werden kann. Die DOM-Extension, die ein solches Arbeiten ermöglicht, wurde mit PHP 4 eingeführt. Leider lehnten sich die damals eingeführten Bezeichner für Eigenschaten und Methoden an die in PHP übliche Schreibweise mit Unterstrichen (also zum Beispiel `append_child()`) an, während das W3C das sogenannte camelCasing (oder auch studlyCaps genannt) benutzt (also `appendChild()`). Und obwohl das DOM des W3C rein objekt-orientiert ist, wurde davon abweichend vieles prozedural inplementiert. Abgesehen von diesen »Schönheitsfehlern« war die DOM-Extension in PHP 4 auch nie wirklich stabil und wurde dementsprechend wenig eingesetzt. Schon mit PHP 4.3 wurde die DOM-Extension auf reine Objekt-Orientierung umgestellt, und mit PHP 5 lehnt sie sich nun noch enger an die Empfehlung des W3C an. Außerdem ist sie nun standardmäßig einkompiliert; mit ihrer Verfügbarkeit kann also auch bei fremden Providern gerechnet werden.

DOM/XML	enabled
DOM/XML API Version	20031129
libxml Version	2.6.9
HTML Support	enabled
XPath Support	enabled
XPointer Support	enabled
Schema Support	enabled
RelaxNG Support	enabled

Abbildung 12.1.: DOM: Auschnitt aus der php_info()-Ausgabe

12.1. Ein XML-Dokument laden, speichern und ausgeben

Es folgt ein kleines Beispiel, in dem ein DOM-Objekt erzeugt, eine XML-Datei eingeladen und anschließend wieder ausgegeben wird:

Listing 12.1: Ein einfaches Beispiel mit DOM-XML

```
<?php
$DomObj = new DomDocument();
$DomObj->load("./dvdArchiv.xml");
echo $DomObj->saveXML();
?>
```

Die Methode `load()` bedient sich der Stream-Funktionalität, die in PHP 4.3 umfassend eingeführt wurde. Das bedeutet, dass nicht nur über das Datei-System, sondern über alle registrierten Streams-Sockets (zum Beispiel HTTP oder FTP) Daten geladen werden können. Dazu analog kann mit der Methode `save()` das XML-Dokument gespeichert werden. Soll das Dokument direkt ausgegeben werden, dann kann der Rückgabewert von `saveXML()` genutzt werden.

12.2. Ein XML-Dokument mit DOM-XML durchlaufen

Listing 12.2: Ein XML-Dokument mit DOM-XML durchlaufen

```
<?php
$DomObj = new DomDocument();
$DomObj->load("./dvdArchiv.xml");
foreach ($DomObj->documentElement->childNodes as $dvd) {
    if ($dvd->nodeType==1) {
        foreach ($dvd->childNodes as $dvdInfo) {
            if ($dvdInfo->nodeType==1) {
                echo $dvdInfo->nodeName . ": ";
                echo $dvdInfo->firstChild->data . "\n";
            }
        }
```

```
    }
}
?>
```

```
Titel: Fight Club
Regisseur: David Fincher
Titel: Familienfest und andere Schwierigkeiten
Regisseur: Jodie Foster
Titel: Catch me if you can
Regisseur: Steven Spielberg
```

Diese Art, ein XML-Dokument zu durchlaufen, ist äußerst flexibel, aber leider auch ein wenig kompliziert. Ein einfacherer Weg kann mit Simple-XML, das in Kapitel 13 auf Seite 77 vorgestellt wird, beschritten werden.

Ein wenig einfacher kann man sich das Leben aber auch DOM-Hausmitteln machen. So gibt die Methode `getElementsByTagName()` ein Objekt vom Typ DomNodeList zurück, dass mit `foreach` durchwandert werden kann. Natürlich sollte man darauf achten, dass dieser Tag-Name eindeutig ist, weil man sonst Knoten in der Rückgabe-Liste hat, die gar nicht erwünscht sind:

Listing 12.3: Beispiel: getElementsByTagName()

```
<?php
$DomObj = new DomDocument();
$DomObj->load("./dvdArchiv.xml");
foreach ($DomObj->getElementsByTagName("Titel") as $dvdTitel) {
    echo $dvdTitel->textContent . "\n";
}
?>
```

Die hier genutzte Eigenschaft `textContent` ist kein W3C-Standard. Sie ist in bestimmten Situationen aber äußerst hilfreich, weil sie den reinen Text-Inhalt eines Knoten (samt Unterknoten) zurückgibt.

```
Fight Club
Familienfest und andere Schwierigkeiten
Catch me if you can
```

12.3. XPath

(TODO)

12.4. XPointer

(TODO)

12.5. Der Transformer: XSLT

XSLT (Extensible Stylesheet Language Transformation) ist eine Sprache, die es ermöglicht, XML-Dokumente in andere XML-Dokumente zu überführen. XSLT selbst ist ein XML-Dialekt und ebenfalls vom World-Wide-Web-Konsortium (W3C) definiert worden.

Die alte XSLT-Extension, die auf der Sablotron-Bilbliothek basierte, hat den Sprung in die neue Version von PHP bislang nicht geschafft und es scheint auch nicht so, als würde das noch passieren. Statt dessen gibt es nun die XSL-Erweiterung, die wie alles XML-mäßige in PHP 5 auf der libxml2-Bibliothek (genauer gesagt: auf der libxslt-Erweiterung der libxml2-Bibliothek, die gegebenfalls nachinstalliert werden muss) basiert. Die XSL-Erweiterung muss (unter Unix/Linux) mit einem `--with-xsl` aktiviert bzw. (unter Windows) als DLL hinzugeladen werden.

XSL	enabled
libxslt Version	1.1.7
libxslt compiled against libxml Version	2.6.9
EXSLT	enabled
libexslt Version	1.1.7

Abbildung 12.2.: XSL: Auschnitt aus der php_info()-Ausgabe

Es folgt eine XSL-Datei, die zur Transformierung unserer XML-Datei genutzt werden soll:

Listing 12.4: Beispiel-XSLT-Dokument

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
    Transform">
    <xsl:output method="html" encoding="iso-8859-1" indent="no"/>
    <xsl:template match="/DVDArchiv">
        <html>
            <head>
                <title>DVD-Archiv</title>
            </head>
            <body>
                <h1>DVD-Archiv</h1>
                <xsl:apply-templates/>
            </body>
        </html>
    </xsl:template>
    <xsl:template match="/DVDArchiv/DVD">
        <h3><xsl:value-of select="Titel"/></h3>
        Regie: <xsl:value-of select="Regisseur"/>
    </xsl:template>
</xsl:stylesheet>
```

Die Tranformation ist in wenigen Zeilen PHP-Code erledigt:

Listing 12.5: Ein XML-Dokument mit XSL transformieren

```
<?php
$DomObj = new DomDocument();
$XslObj = new DomDocument();
$DomObj->load("./dvdArchiv.xml");
$XslObj->load("./dvdArchiv2Html.xsl");

$XsltProcObj = new XsltProcessor();
$XsltProcObj->importStylesheet($XslObj);

echo $XsltProcObj->transformToXML($DomObj);
?>
```

Zuerst werden zwei Objekte der Klasse `DomDocument` instanziert und die XML-Dokumente eingelesen (XSLT ist ja auch XML). Anschließend wird der XSLT-Prozessor (ein Objekt vom Typ `XsltProcessor`) erzeugt und mit `importStylesheet()` das XSLT geladen. Nun reicht es aus, den XSLT-Prozessor anzuweisen, aus der XML-Datei ein neues XML-Dokument zu erzeugen.

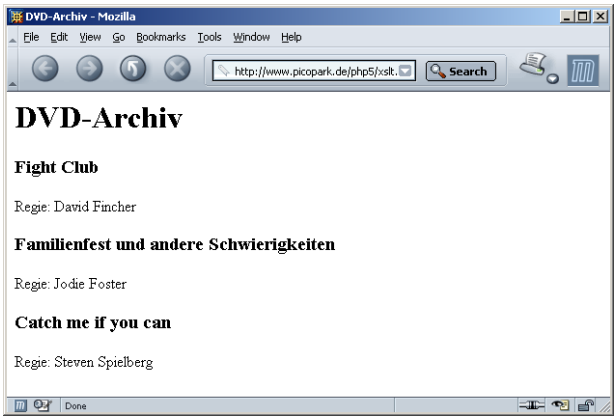


Abbildung 12.3.: Die Browser-Ausgabe des XML-Dokuments nach der XSL-Transformation

12.5.1. PHP-Funktionen aus dem Stylesheet aufrufen

Es ist möglich, in das XSLT-Dokument Aufrufe für PHP-Funktionen einzubauen. Das klingt ziemlich mächtig (und ist es auch!), aber man erkauft sich diese Funktionalität mit dem Verlust an Portabilität des XSLT-Dokuments auf Nicht-PHP-Systeme. Es empfiehlt sich also zweimal zu überlegen, ob es nicht doch einen standardkonformen Weg gibt, bevor man dieses Feature tatsächlich einsetzt.

Listing 12.6: Beispiel: XSLT-Dokument mit Aufruf einer PHP-Funktion

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
    Transform" xmlns:php="http://php.net/xsl">
    <xsl:output method="html" encoding="iso-8859-1" indent="no"/>
    <xsl:template match="/DVDArchiv">
        <html>
            <head>
                <title>DVD-Archiv</title>
            </head>
            <body>
                <h1>DVD-Archiv</h1>
                Stand: <xsl:value-of select="php:function('date', 'Y-m
                    -d H:i:s')"/>
                <xsl:apply-templates/>
            </body>
        </html>
```

```
    </xsl:template>
    <xsl:template match="/DVDArchiv/DVD">
        <h3><xsl:value-of select="Titel"/></h3>
        Regie: <xsl:value-of select="Regisseur"/>
    </xsl:template>
</xsl:stylesheet>
```

Nachdem der Namespace `php` deklariert wurde (hier in Zeile 2), kann jede eingebaute oder selbstdefinierte Funktion wie in Zeile 11 gezeigt aufgerufen werden, also mit dem Funktionsnamen als ersten und den zu übergebenden Argumenten als weitere Parameter.

Listing 12.7: PHP-Funktionen im XSLT-Dokument aufrufen

```
<?php
$DomObj = new DomDocument();
$XslObj = new DomDocument();
$DomObj->load("./dvdArchiv.xml");
$XslObj->load("./dvdArchiv2Html2.xsl");

$XsltProcObj = new XsltProcessor();
$XsltProcObj->registerPHPFunctions();
$XsltProcObj->importStylesheet($XslObj);

echo $XsltProcObj->transformToXML($DomObj);
?>
```

Damit das XSLT-Prozessor-Objekt die beschriebene Funktionalität bereitstellt, muss aus Sicherheitsgründen die Methode `registerPHPFunctions()` aufgerufen werden. So wird verhindert, dass jedes XSLT-Dokument, sobald es von PHP aufgerufen wird, beliebigen Code ausführen kann.

13. Simple-XML

Es wird nicht viel geben, was Sie mit einem XML-Dokument anstellen möchten und mit DOM nicht erreichen können. Häufig ist aber die Funktionsvielfalt erschlagend und in der Anwendung unnötig kompliziert (obwohl DOM den Schrecken verliert, wenn man sich einmal daran gewöhnt hat). Daher gibt es in PHP 5 das »kleine Besteck«: Simple-XML.

Simplexml support	enabled
Revision	$Revision: 1.139 $
Schema support	enabled

Abbildung 13.1.: Simple-XML: Auschnitt aus der php_info()-Ausgabe

Simple-XML ist eine Erweiterung, die standardmäßig zur Verfügung steht und die ebenfalls auf der libxml2-Bibliothek beruht. Gerade wenn Sie XML-Dokumente bekannter Struktur auslesen möchten, werden Sie die Einfachheit schnell zu schätzen lernen. Und sollten Sie an die Grenzen von Simple-XML stoßen, dann können Sie Ihr XML-Dokument mit DOM weiterbearbeiten – und alles, ohne große Klimmzüge machen zu müssen.

13.1. Ein XML-Dokument mit Simple-XML durchlaufen

Listing 13.1: XML-Dokument mit Simple-XML durchlaufen

```
<?php
$SimpleXmlObj = simplexml_load_file("./dvdArchiv.xml");
foreach ($SimpleXmlObj as $Dvd) {
    foreach ($Dvd as $tagname => $text) {
```

```
        echo $tagname . ": " . $text . "\n";
    }
}
?>
```

Dieses Skript macht genau das gleiche und führt auch zur gleichen Ausgabe wie der Code des Listings 12.2 auf Seite 71:

```
Titel: Fight Club
Regisseur: David Fincher
Titel: Familienfest und andere Schwierigkeiten
Regisseur: Jodie Foster
Titel: Catch me if you can
Regisseur: Steven Spielberg
```

Man erkennt schnell, dass der Simple-XML-Code kürzer zu schreiben und auch einfacher zu lesen ist als das DOM-Äquivalent. Und wie Sie sich vielleicht schon denken können: Das Objekt `$SimpleXmlObj` im obigen Beispiel, das von der Funktion `simplexml_load_file()` zurückgegeben wird, ist ein Objekt, dass das Interface `Traversable` (siehe Seite 60) implementiert und daher mit `foreach` durchlaufbar ist. Dieses Objekt ist von Typ `SimpleXMLElement` und beinhaltet die Unterknoten des Wurzelelements. Diese Unterknoten sind wiederum vom Typ `SimpleXMLElement`, die ebenfalls Unterknoten enthalten können, und so weiter und so fort... Auf diese Unterknoten kann ebenfalls mit der Methode `children()` zugegriffen werden.

Auch folgende, fast schon selbsterklärende Schreibweise ist möglich:

Listing 13.2: XML-Dokument mit Simple-XML durchlaufen, 2. Beispiel

```
<?php
$SimpleXmlObj = simplexml_load_file("./dvdArchiv.xml");
foreach ($SimpleXmlObj as $Dvd) {
    echo "Titel: " . $Dvd->Titel . "\n";
    echo "Regie: " . $Dvd->Regisseur . "\n";
}
?>
```

Es ist also möglich, auf die Knoten zuzugreifen, indem man die Knotennamen wie Eigenschaften behandelt. Natürlich hat ein `SimpleXMLElement` weder die Eigenschaft Titel noch Regisseur – dass es trotzdem funktioniert, ist der

Overload-Extension, die im Abschnitt 2.3 auf Seite 12 vorgestellt wurde, zu verdanken.

Auf Elemente kann aber nicht nur zugegriffen werden, indem man sie in einer Schleife durchläuft; auch der direkte Zugriff durch einen Index ist möglich:

Listing 13.3: Simple-XML: Zugriff durch einen Index

```
<?php
$SimpleXmlObj = simplexml_load_file("./dvdArchiv.xml");
echo $SimpleXmlObj->DVD[1]->Titel;
?>
```

```
Familienfest und andere Schwierigkeiten
```

13.2. Zugriff auf Attribute

Attribut-Bezeichner eines XML-Knotens dürfen niemals nur aus Zahlen bestehen und müssen mit einem Buchstaben beginnen – und genau dies macht man sich in Simple-XML zu Nutze, um auf Attribute zuzugreifen. Während numerische Indices auf eventuell vorhandene Unterknoten verweisen, kann man auf Attribute zugreifen, indem man den Bezeicher als Index nutzt:

Listing 13.4: Simple-XML: Zugriff auf Attribute

```
<?php
$SimpleXmlObj = simplexml_load_file("./dvdArchiv.xml");
echo $SimpleXmlObj->DVD[1]["id"];
?>
```

```
DVD2
```

Ein `SimpleXMLElement` hat aber auch die Methode `attributes()`, die die Attribute eines Knotens wiederum als `SimpleXMLElement` zurückgibt:

Listing 13.5: Simple-XML: Zugriff auf Attribute mit `attributes()`

```
<?php
$SimpleXmlObj = simplexml_load_file("./dvdArchiv.xml");
foreach ($SimpleXmlObj as $Dvd) {
    echo $Dvd->Titel . ": ";
    foreach ($Dvd->attributes() as $attributename => $value) {
        echo $attributename . ": " . $value . " ";
    }
    echo "\n";
}
?>
```

```
Fight Club: Disczahl: 2 id: DVD1
Familienfest und andere Schwierigkeiten: Disczahl: 1 id: DVD2
Catch me if you can: Disczahl: 2 id: DVD3
```

13.3. Berücksichtigung von Namensräumen (namespaces)

Macht ein XML-Dokument Gebrauch von Namesräumen (namespaces), dann erlebt man beim Durchlaufen mit Simple-XML eine kleine Überraschung. Nehmen wir zum Beispiel folgendes XML-Dokument:

Listing 13.6: Ein XML-Dokument mit Namesräumen

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<DVDArchiv xmlns:eins="http://virtuelle-maschine.de/ns/eins" xmlns:
    zwei="http://virtuelle-maschine.de/ns/zwei">
    <DVD Disczahl="1" id="DVD4">
        <Titel>Was gibt's Neues, Pussycat?</Titel>
        <eins:Kritik>Die erste Kritik zu Pussycat.</eins:Kritik>
        <zwei:Kritik>Die zweite Kritik zu Pussycat.</zwei:Kritik>
    </DVD>
    <DVD Disczahl="1" id="DVD2">
        <Titel>Familienfest und andere Schwierigkeiten</Titel>
        <eins:Kritik>Die erste Kritik zu Familienfest.</eins:Kritik>
        <zwei:Kritik>Die zweite Kritik zu Familienfest.</zwei:Kritik>
    </DVD>
</DVDArchiv>
```

Wird dieses XML-Dokument in Simple-XML eingelesen und druchlaufen (vgl. Listing 13.1 auf Seite 77), dann erhält man folgende Ausgabe:

```
Titel: Was gibt's Neues, Pussycat?
Titel: Familienfest und andere Schwierigkeiten
```

Die Knoten, die Namensräumen zugeordnet sind, werden also nicht berücksichtigt. Dass es aber trotzdem möglich ist, auf solche Knoten zuzugreifen, ist der Methode `children()` zu verdanken, die weiter oben eher beiläufig erwähnt wurde:

Listing 13.7: Simple-XML und Namesräume

```
<?php
$SimpleXmlObj = simplexml_load_file("./namespaces.xml");
foreach ($SimpleXmlObj as $Dvd) {
    foreach ($Dvd->children("http://virtuelle-maschine.de/ns/eins") as
         $tagname => $text) {
        echo $tagname . ": " . $text . "\n";
    }
}
?>
```

```
Kritik: Die erste Kritik zu Pussycat.
Kritik: Die erste Kritik zu Familienfest.
```

Es ist zu beachten, dass die Methode `children()` den URL des Namenraums erwartet und nicht den Bezeichner.

13.4. Einlesen und Ausgabe eines XML-Dokuments

Neben `simplexml_load_file()`, das einen XML-Stream öffnet und einliest, gibt es noch die Funktion `simplexml_load_string()`, die eine Zeichenkette erwartet, als XML interpretiert und ein entsprechendes `SimpleXMLElement` zurückgibt.

Die Methode `asXML()` gibt das `SimpleXMLElement`-Objekt wieder als Zeichenkette zurück, die anschließend in einen Stream geschrieben (zum Beispiel mit der Funktion `file_put_contents()`, die im Abschnitt 16.2.2 auf Seite 97 vorgestellt wird) oder auch direkt ausgegeben werden kann:

Listing 13.8: Simple-XML: Zugriff auf Attribute mit `attributes()`

```
<?php
$SimpleXmlObj = simplexml_load_file("./dvdArchiv.xml");
echo $SimpleXmlObj->DVD[1]->asXML();
?>
```

```
<DVD Disczahl="1" id="DVD2">
        <Titel>Familienfest und andere Schwierigkeiten</Titel>
        <Regisseur>Jodie Foster</Regisseur>
    </DVD>
```

13.5. XPath-Ausdrücke nutzen

Wenn man etwas kompliziertere Abfragen an das XML-Objekt stellen möchte, dann wird man um XPath nicht herumkommen. Durch XPath-Ausdrücke hat man die Möglichkeit, durch spezielle (und bei Bedarf auch komplexe) Suchkriterien auf bestimmte Knoten zuzugreifen.[1] Folgender Code zeigt zum Beispiel, wie man aus dem DVD-Archiv alle Titel ausliest, die aus zwei DVDs bestehen:

Listing 13.9: Simple-XML: XPath-Ausdrücke nutzen

```
<?php
$SimpleXmlObj = simplexml_load_file("./dvdArchiv.xml");
$Titel = $SimpleXmlObj->xpath('/DVDArchiv/DVD[@Disczahl="2"]/Titel');
foreach ($Titel as $wert) {
    echo $wert . "\n";
}
?>
```

[1] XPath wird daher auch bisweilen als die SQL für XML bezeichnet.

```
Fight Club
Catch me if you can
```

Die Methode `xpath()` erwartet einen XPath-Ausdruck und gibt ein Array zurück, das aus Objekten vom Typ `SimpleXMLElement` besteht.

13.6. Änderung des XML-Dokuments

Mit Simple-XML können XML-Dokumente aber nicht nur ausgelesen, sondern auch verändert werden. Dies gilt sowohl für Textknoten als auch für Attribute:

Listing 13.10: Ein Simple-XML-Objekt verändern

```
<?php
$SimpleXmlObj = simplexml_load_file("./dvdArchiv.xml");
$SimpleXmlObj->DVD[1]->Titel = "Was gibt's Neues, Pussycat?";
$SimpleXmlObj->DVD[1]->Regisseur = "Blake Edwards";
$SimpleXmlObj->DVD[1]["id"] = "DVD4";
echo $SimpleXmlObj->asXML();
?>
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<DVDArchiv>
    <DVD Disczahl="2" id="DVD1">
        <Titel>Fight Club</Titel>
        <Regisseur>David Fincher</Regisseur>
    </DVD>
    <DVD Disczahl="1" id="DVD4">
        <Titel>Was gibt's Neues, Pussycat?</Titel>
        <Regisseur>Clive Donner</Regisseur>
    </DVD>
    <DVD Disczahl="2" id="DVD3">
        <Titel>Catch me if you can</Titel>
        <Regisseur>Steven Spielberg</Regisseur>
    </DVD>
</DVDArchiv>
```

Dagegen ist es nicht möglich, dem Objekt weitere Knoten hinzuzufügen:

Listing 13.11: Einem Simple-XML-Objekt Knoten direkt hinzuzufügen funktioniert nicht

```
<?php
$SimpleXmlObj = simplexml_load_file("./dvdArchiv.xml");
$SimpleXmlObj->DVD[3]->Titel = "Was gibt's Neues, Pussycat?";
$SimpleXmlObj->DVD[3]->Regisseur = "Clive Donner";
$SimpleXmlObj->DVD[3]["id"] = "DVD4";
echo $SimpleXmlObj->asXML();
?>
```

```
Fatal error: Objects used as arrays in post/pre
increment/decrement must return values by reference in
/htdocs/php5neu/simpleXml8.php5 on line 3
```

Es ist aber möglich, einem Knoten weitere Attribute hinzuzufügen – auch wenn die Anwendungsmöglichkeiten dafür meist beschränkt sein werden:

Listing 13.12: Simple-XML: Attribute hinzufügen

```
<?php
$SimpleXmlObj = simplexml_load_file("./dvdArchiv.xml");
$SimpleXmlObj->DVD[0]["Vertrieb"] = "20th Century Fox";
$SimpleXmlObj->DVD[1]["Vertrieb"] = "Paramount";
$SimpleXmlObj->DVD[2]["Vertrieb"] = "Dreamworks";
echo $SimpleXmlObj->asXML();
?>
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<DVDArchiv>
    <DVD Disczahl="2" id="DVD1" Vertrieb="20th Century Fox">
        <Titel>Fight Club</Titel>
        <Regisseur>David Fincher</Regisseur>
    </DVD>
    <DVD Disczahl="1" id="DVD2" Vertrieb="Paramount">
        <Titel>Familienfest und andere Schwierigkeiten</Titel>
        <Regisseur>Jodie Foster</Regisseur>
    </DVD>
    <DVD Disczahl="2" id="DVD3" Vertrieb="Dreamworks">
        <Titel>Catch me if you can</Titel>
        <Regisseur>Steven Spielberg</Regisseur>
    </DVD>
</DVDArchiv>
```

Will man also größere Veränderungen am XML-Objekt vornehmen, dann wird man wohl um DOM nicht herumkommen. Und der Weg von einem Simple-XML-Objekt zu einem DOM-Objekt ist kürzer als viele vielleicht vermuten ...

13.7. Einmal DOM und zurück

Sowohl die Simple-XML- als auch die DOM-Funktionalität beruht in PHP 5 auf der libxml2-Bibliothek. Und das ermöglicht einen erstaunlichen Kniff: Ein Objekt vom Typ `SimpleXMLElement` kann in ein Objekt vom Typ `DomElement` gewandelt werden und vice-versa. Dabei ist gewandelt sogar noch der falsche Begriff: Das zugrunde liegende Objekt ist ein und das selbe! Es müssen also keine resourcenschluckenden Konvertierungen stattfinden, sondern nur der Zugriff verändert sich.

Listing 13.13: Ein Simple-XML-Objekt in ein DOM-Objekt überführen und ausgeben

```
<?php
$SimpleXmlObj = simplexml_load_file("./dvdArchiv.xml");
$DomObj = dom_import_simplexml($SimpleXmlObj);
echo $DomObj->ownerDocument->saveXML();
?>
```

(Bitte beachten Sie, dass `dom_import_simplexml()` ein Objekt vom Typ `DomElement` zurückgibt, man aber nur ein `DomDokument` (das das Wurzelelement repräsentiert) als XML ausgeben werden kann. Daher muss im Beipiel mit der Eigenschaft `ownerDocument` der übergeordnete Knoten aufgerufen werden.)

Natürlich dienst die Ausgabe des Dom-Dokuments nur als Beispiel. Denn eine reine Ausgabe könnten Sie ja auch mit Simple-XML-Hausmitteln allein bewerkstelligen. Richtig spannend wird es erst, wenn nun die mächtigen Möglichkeiten des DOM benutzt werden, um das XML nach Lust und Laune (und Anforderung...) zu manipulieren.

Natürlich kann ein `DomElement` auch wieder zu einem `SimpleXMLElement` werden; die entsprechende Funktion lautet `simplexml_import_dom()`:

Listing 13.14: Ein DOM-Objekt in ein Simple-XML-Objekt überführen und ausgeben

```
<?php
$DomObj = new DomDocument();
$DomObj->load("./dvdArchiv.xml");
$SimpleXmlObj = simplexml_import_dom($DomObj);
echo $SimpleXmlObj->asXML();
?>
```

13.8. Simple-XML und »gemischte« Knoten

Es gibt keine Möglichkeit, in Simple-XML auf Knoten zuzugreifen, die Text und Unterknoten mischen:

Listing 13.15: In Simple-XML kann man nicht auf gemischte Knoten zugreifen

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<DVD>
    <Beschreibung>Ein <kursiv>toller</kursiv> Film mit <fett>Holly
        Hunter</fett>.</Beschreibung>
</DVD>
```

In diesem Beispiel kann man zwar auf den Text der Unterknoten `kursiv` und `fett` zugreifen, der restliche Text des Knotens `Beschreibung` entzieht sich jedoch dem Zugriff. Wollen Sie ein solches XML-Dokument verarbeiten, dann müssen Sie in solchen Momenten auf DOM zurückgreifen (oder Sie begnügen sich mit der gemischen Form, die `asXML()` zurückgibt).

14. Übermut. Chaos. Seife. Webservices.

(TODO: Einleitung SOAP)

Im Unterschied zu fast allen anderen in diesem Buch vorgestellten Neuerungen steht die SOAP-Extension in PHP 5 standardmäßig *nicht* zur Verfügung. Sie muss durch ein `--enable-soap` explizit einkompiliert bzw. durch einen Eintrag in der php.ini dynamisch geladen werden.

Soap Client	enabled
Soap Server	enabled

Directive	Local Value	Master Value
soap.wsdl_cache_dir	/tmp	/tmp
soap.wsdl_cache_enabled	1	1
soap.wsdl_cache_ttl	86400	86400

Abbildung 14.1.: SOAP: Auschnitt aus der php_info()-Ausgabe

14.1. WSDL: Die Beschreibung des Dienstes in XML

Leider unterstützt die SOAP-Erweiterung (im Moment noch?) nicht die mehr oder weniger automatische Erstellung der WSDL-Datei[1]. Eine WDSL-Datei

[1] Die NuSoap-Bibliothek bieten zum Beispiel eine solche Funktion.

(WSDL steht für »**W**eb**S**ervice-**D**escription-**L**anguage«) beschreibt den Webdienst, also das Format der Ein- und Ausgabe, die genutzten Spezifikationen sowie den URL des Dienstes.

Listing 14.1: Die WSDL-Datei

```
<?xml version="1.0" ?>
<definitions name="Capital" targetNamespace="http://www.picopark.de/
    demo/soap/capital.wsdl" xmlns:tns="http://www.picopark.de/demo/
    soap/capital.wsdl" xmlns:xsd="http://www.w3.org/2000/10/XMLSchema
    " xmlns:xsd1="http://www.picopark.de/demo/soap/capital.xsd" xmlns
    :soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns="http://
    schemas.xmlsoap.org/wsdl/">
    <message name="getCapitalInput">
        <part name="tickerSymbol" element="xsd:string"/>
    </message>
    <message name="getCapitalOutput">
        <part name="result" type="xsd:string"/>
    </message>
    <portType name="CapitalPortType">
        <operation name="getCapital">
            <input message="tns:getCapitalInput"/>
            <output message="tns:getCapitalOutput"/>
        </operation>
    </portType>
    <binding name="CapitalBinding" type="tns:CapitalPortType">
        <soap:binding style="rpc" transport="http://schemas.xmlsoap.
            org/soap/http"/>
        <operation name="getCapital">
            <soap:operation soapAction="http://www.picopark.de/demo/
                soap/capital.wsdl"/>
            <input>
                <soap:body use="encoded" namespace="http://www.
                    picopark.de/demo/soap/capital.wsdl" encodingStyle
                    ="http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>
                <soap:body use="encoded" namespace="http://www.
                    picopark.de/demo/soap/capital.wsdl" encodingStyle
                    ="http://schemas.xmlsoap.org/soap/encoding/"/>
            </output>
        </operation>
    </binding>
    <service name="CapitalService">
        <documentation>Mein erster Web-Service.</documentation>
        <port name="CapitalPort" binding="tns:CapitalBinding">
            <soap:address location="http://localhost/buch_php5_neues/
```

```
                capital.php5"/>
        </port>
    </service>
</definitions>
```

14.2. Der SOAP-Server

Listing 14.2: Der SOAP-Server

```
<?php
function getCapital($land) {
    $stadtArray = array(
        "de" => "Berlin",
        "fr" => "Paris",
        "gb" => "London"
    );
    $land = strtolower(trim($land));

    if ( isset( $stadtArray[$land] ) ) {
        return $stadtArray[$land];
    } else {
        return "???";
    }return $quotes[$symbol];
}

//! Wichtig zum Testen: Den Cache ausschalten,
//! damit Änderungen an der WSDL-Datei unmittel-
//! bar übernommen werden:
ini_set("soap.wsdl_cache_enabled", "0");

$Server = new SoapServer("./capital.wsdl");
$Server->addFunction("getCapital");
$Server->handle();
?>
```

14.3. Der SOAP-Client

Listing 14.3: Der SOAP-Client

```
<?php
$Client = new SoapClient("./capital.wsdl");
echo $Client->getCapital("de") . "\n";
echo $Client->getCapital("FR");
?>
```

```
Berlin
Paris
```

15. SQLite: 100% SQL bei 0% Server

OK, die Kapitel-Überschrift ist ein wenig geschwindelt – aber sie zeigt, wohin die Reise geht. Doch der Reihe nach...

SQLite support	enabled
PECL Module version	1.1-dev $Id: sqlite.c,v 1.144 2004/05/13 14:19:10 stas Exp $
SQLite Library	2.8.11
SQLite Encoding	iso8859

Directive	Local Value	Master Value
sqlite.assoc_case	0	0

Abbildung 15.1.: SQLite: Auschnitt aus der php_info()-Ausgabe

15.1. SQLite: Übersicht

(TODO)

15.1.1. Was SQLite ist ...

(TODO)

15.1.2. ... und was nicht

(TODO)

15.2. Der SQL(ite)-Dialekt

(TODO)

15.3. SQLite in PHP 5 nutzen

(TODO)

16. Weitere neue Funktionen

Hier werden nun noch einige (aber bei weitem nicht alle) Funktionen vorgestellt, die im Zuge der Version 5 Einzug in PHP gehalten haben und die in den vorangegangenen Kapiteln nicht erwähnt wurden.

16.1. Neue Array-Funktionen

16.1.1. `array_combine()`

Syntax

```
array array_combine( array keys, array values )
```

`array_combine()` erwartet als Parameter zwei Arrays und gibt ein Array zurück, dass als Schlüssel (Keys) die Elemente des ersten und als Werte die Elemente des zweiten übergebenden Arrays benutzt.

Listing 16.1: Beispiel: array_combine()

```
<?php
$arrayKeys = array("DE", "FR", "GB");
$arrayVals = array("Berlin", "Paris", "London");
$arrayNeu  = array_combine($arrayKeys, $arrayVals);
print_r($arrayNeu);
?>
```

Dies führt zu folgender Ausgabe:

```
Array
(
```

```
    [DE] => Berlin
    [FR] => Paris
    [GB] => London
)
```

Haben die Ausgangs-Arrays eine unterschiedliche Zahl an Elementen oder sind sie leer, dann gibt `array_combine()` den boolschen Wert FALSE zurück.

Vorsicht ist angebracht, wenn in dem Array, das die Schlüssel bereitstellt, der selbe Wert mehrfach vorkommt:

Listing 16.2: array_combine(): Vorsicht bei gleichen Schlüssel-Bezeichner

```
<?php
$arrayKeys = array("DE", "FR", "GB", "DE");
$arrayVals = array("Berlin", "Paris", "London", "Bonn");
$arrayNeu  = array_combine($arrayKeys, $arrayVals);
print_r($arrayNeu);
?>
```

16.1.2. `array_diff_uassoc()`

`array_diff_uassoc()` ist vergleichbar mit der in PHP 4.3 eingeführten Funktion `array_diff_assoc()`, mit dem Unterschied, dass nun eine Callback-Funktion angegeben werden kann, die über Gleichheit der Schlüssel entscheidet.

Syntax

```
array array_diff_uassoc( array array1, array array2 [, array array3,
                            ...], callback function )
```

Es wird ein Array zurückgegeben, das alle Elemente aus `array1` enthält, deren Schlüssel-Wert-Kombination nicht in `array2` (und optional weiteren Arrays) vorkommen. Dabei wird der Vergleich der Schlüssel durch eine benutzerdefinierte Funktion vorgenommen, die zwei Argumente erwartet. Sind die beiden Argumente gleich, dann soll die Funktion 0 zurückgeben; ist das erste Argument größer als das zweite, dann eine Ganzzahl größer 0, ansonsten eine Ganzzahl kleiner 0. Deutlicher (oder erst deutlich?) wird es mit einem Beispiel:

Listing 16.3: Beispiel: array_diff_uassoc()

```
<?php
function schluesselVergleich($key1, $key2)
{
    $key1 = strtolower($key1);
    $key2 = strtolower($key2);
    if ( $key1 == $key2 ) {
        return 0;
    }
    return ( $key1 < $key2 ) ? -1 : 1;
}
$array1 = array(
    "DE" => "Berlin",
    "FR" => "Paris",
    "GB" => "London",
    "Rom"
);
$array2 = array(
    "eu" => "Brüssel",
    "fr" => "Paris",
    "gb" => "London",
    "Wien",
    "Rom"
);
$arrayNeu  = array_diff_uassoc($array1, $array2,
                                "schluesselVergleich");
print_r($arrayNeu);
?>
```

```
Array
(
    [DE] => Berlin
    [0] => Rom
)
```

Die hier eingesetzte Callback-Funktion wandelt die Schlüssel in Kleinbuchstaben und vergleicht sie erst dann (für einen 1:1-Vergleich hätte man ja auch auf das schnellere `array_diff_assoc()` zurückgreifen können). Das der Wert 'Rom' im Array verbleibt, liegt an den unterschiedlichen Schlüsseln: Im `array1` hat er den Schlüssel `0`, im `array2` aber den Schlüssel `1`.

16.1.3. array_udiff()

16.1.4. array_udiff_assoc()

16.1.5. array_udiff_uassoc()

16.1.6. array_walk_recursive()

Syntax

```
bool array_walk_recursive( array array, callback function [, mixed
                           userdata] )
```

Listing 16.4: Beispiel: array_walk_recursive()

```
<?php
function callbackFunc($value, $key, $bla)
{
    echo  $key . " => " . $value . "\n";
}
$array = array(
    "DE" => array(
        "Berlin",
        "Bonn",
        "Hamburg"
    ),
    "FR" => "Paris",
    "GB" => "London",
);
array_walk_recursive($array, "callbackFunc");
?>
```

Leider stürzt PHP 5 (sowohl RC 2 als auch RC 3) bei Benutzung dieser Funktion zuverlässig ab...

16.2. Sonstige neue Funktionen

16.2.1. debug_print_backtrace()

Syntax

```
void debug_print_backtrace( void )
```

16.2.2. `file_put_contents()`

Syntax

```
int file_put_contents( string dateiname, string data [, int flags [,
                              resource context]] )
```

Das, was seit PHP 4.3 mit `file_get_contents()` lesenderweise möglich ist, bietet nun `file_put_contents()` für das Schreiben: In einem Rutsch einen Stream öffnen, beschreiben und wieder schließen. Als erster Parameter muss die zu schreibende Zeichenkette angegeben werden, wobei auch binäre Daten erlaubt sind. Der als zweiter Parameter erlaubte Integerwert kann eine Kombination (durch ein binäres ODER »`&`« verknüpft) aus `FILE_USE_INCLUDE_PATH` und `FILE_APPEND` sein; ohne Angabe von `FILE_APPEND` wird eine eventuell vorhandene Datei überschrieben. Zurückgegeben wird die Anzahl der geschrieben Bytes.

16.2.3. `get_headers()`

Syntax

```
array get_headers( string url )
```

Die Funktion `get_headers()` erwartet einen URL und gibt die Headerzeilen der Antwort einer entsprechenden HTTP-GET-Anfrage als Array zurück.

Listing 16.5: Beispiel: get _headers()

```
<?php
$array = get_headers("http://www.virtuelle-maschine.de/");
print_r($array);
?>
```

```
Array
(
    [0] => HTTP/1.1 200 OK
    [1] => Date: Tue, 13 Jul 2004 09:38:22 GMT
    [2] => Server: Apache/1.3.31 (Unix) PHP/4.3.7
    [3] => X-Powered-By: PHP/4.3.7
    [4] => Connection: close
    [5] => Content-Type: text/html
)
```

16.2.4. `headers_list()`

Syntax

```
array headers_list( void )
```

`headers_list()` gibt ein Array mit den Header-Zeilen zurück, die PHP an den Client senden wird oder schon gesendet hat. Solange sie noch nicht gesendet wurden (überprüfbar mit `heders_sent()`), können sie mit den bekannten Funktionen `headers()` bzw. `setcookie()` sowie der neuen `setrawcookie()` manipuliert werden.

Listing 16.6: Beispiel: headers_list()

```
<?php
header("X-Header-Test: Dies ist ein Test");
setcookie('MeinTestCookie', 'Hallo Welt');

print_r(headers_list());
?>
```

```
Array
(
    [0] => X-Powered-By: PHP/5.0.0RC2
    [1] => X-Header-Test: Dies ist ein Test
    [2] => Set-Cookie: MeinTestCookie=Hallo+Welt
)
```

16.2.5. `http_build_query()`

Syntax

```
string http_build_query( array formdata [, string numeric_prefix] )
```

A. Schlüsselwörter

Neben den unten aufgeführten Schlüsselwörten dürfen Sie auch keine Bezeichner nutzen, die durch eingebaute Klassen oder Interafaces sowie Funktionen belegt sind . Die neuen Schlüsselwörter sind fettgedruckt.

```
__CLASS__
__FILE__
__FUNCTION__
__LINE__
__METHOD__
abstract
and
array()
as
break
case
catch
cfunction
class
clone
const
continue
declare
default
die()
do
echo()
else
elseif
empty()
enddeclare
endfor
endforeach
endif
endswitch
endwhile
eval
exit()
extends
final
for
foreach
function
global
if
implements
include()
include_once()
isset()
instanceof
list()
new
old_function
or
print()
```

private
protected
public
require()
require_once()
return()
static
switch
throw
try
unset()
use
var
while
xor

Listings

Abbildungsverzeichnis

Index

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z