Jan Teriete

Objektorientiertes PHP7

Band 1: Grundlagen der OOP

Ein Webmasters Press Lernbuch

Version 10.2.0 vom 23.04.2019

Autorisiertes Curriculum für das Webmasters Europe Ausbildungs- und Zertifizierungsprogramm

Inhaltsverzeichnis

Virtuelle Attribute

6

In dieser Lektion lernen Sie

- wie flexibel Getter und Setter sind und welche Vorteile das hat.
- wie Sie mit virtuellen Attributen arbeiten.

6.1 Das Problem

In vielen Programmier-Projekten gibt es zwei Anforderungen an die Daten.

- Sie sollen möglichst effizient und nicht redundant abgelegt werden.
- Es soll möglichst einfach sein, mit ihnen zu arbeiten.

Leider schließen sich die beiden Bedingungen meistens aus:

Junior-Programmierer	Senior-Programmierer
Können wir nicht ein Attribut Name anlegen, das den Vornamen und den Nachnamen enthält? Das brauchen wir ständig und es ist aufwendig, das jedes Mal zusammenzubauen.	Wir dürfen auf keinen Fall Daten an mehreren Stellen abspeichern. Das macht uns später garantiert Ärger.
Wir brauchen dieselben Texte einmal unformatiert und einmal mit HTML-Tags. Können wir nicht beides abspeichern?	Dieselben Informationen in mehreren Formaten abzuspeichern erzeugt garantiert Inkonsistenzen. Sucht euch ein Format aus und konvertiert die Daten bei Bedarf!
Mit den Daten, die ihr liefert, kann man unmöglich vernünftig arbeiten!	Würden wir die Daten so ablegen, wie ihr es wollt, gäbe es ein heilloses Durcheinander!

Tabelle 6.1 *Konflikte*

6.2 Virtuelle Attribute

6.2.1 Konzept

Es gibt also einen Konflikt, wie Daten abzulegen sind. Einerseits möchten wir sie möglichst gut strukturiert und wiederverwendbar abspeichern, andererseits sollte man auch gut mit ihnen arbeiten können. Die objektorientierte Programmierung hat für dieses Problem eine praktische Lösung parat: Sie lügt!

Das ist weniger schlimm, als es sich anhört. Wir gaukeln einfach die Existenz von Attributen vor, die es in Wirklichkeit gar nicht gibt. Dies erreichen wir, indem wir Getter-Methoden schreiben, die sich die Daten aus anderen Attributen holen und aufbereiten.

Diese Vorgehensweise ist als **Uniform Access Principle** (siehe https://de.wikipedia.org/wiki/Prinzipien_objektorientierten_Designs#Uniform_Access_Prinzip) bekannt. Das Prinzip besagt, dass für den Zugriff von außen eine gleichartige Notation (Schreibweise) verwendet werden soll. Diese Notation gibt jedoch nicht preis, ob sie über direkte Datenzugriffe oder »Berechnungen« umgesetzt wurde. Die Notwendigkeit einer solchen Vorgehensweise wurde übrigens zuerst von *Bertrand Meyer* erkannt und beschrieben.

Beispiel

```php
<?php

class Buch
{
    protected $titel = '';
    protected $preis = 0; // Nettopreis

    public function getTitel()
    {
        return $this->titel;
    }

    public function getPreis()
    {
        return $this->preis;
    }

    public function getBruttoPreis()
    {
        $bruttoPreis = $this->getPreis() * 1.07;
        return $bruttoPreis;
    }
}
```

Codebeispiel 54 *buch.php (Version 1)*

Im Beispiel sehen Sie eine Klasse `Buch`, die über zwei echte Attribute verfügt. Auf Setter habe ich hier bewusst verzichtet, um das Beispiel kurz zu halten. Es existieren aber drei Getter, wobei einer, nämlich `getBruttoPreis()`, kein reales Attribut ausgibt. Diese Methode nimmt das Attribut `$preis` und rechnet sieben Prozent Mehrwertsteuer dazu. Den errechneten Wert gibt sie dann zurück. Von außen scheint es also, als gäbe es ein Attribut `$bruttoPreis`, das durch diesen Getter verwaltet wird. Dieses Vorgaukeln von Attributen bezeichnen wir als ***virtuelle Attribute***.

6.2.2 Setter für virtuelle Attribute

Natürlich ist es genauso möglich, virtuelle Attribute mit Settern auszustatten. Sie müssen in diesem Fall allerdings beachten, dass Sie nun unter Umständen mehrere Setter für ein reales Attribut haben. Diese dürfen sich nie in die Quere kommen. Davon abgesehen müssen Sie nur aufpassen, dass Sie die korrekten Werte erzeugen.

Beispiel

```php
<?php

class Buch
{
    protected $titel = '';
    protected $preis = 0; // Nettopreis

    public function getTitel()
    {
        return $this->titel;
    }

    public function setTitel($titel)
    {
        $this->titel = $titel;
    }

    public function getPreis()
    {
```

```
20          return $this->preis;
21      }
22
23      public function setPreis($preis)
24      {
25          $this->preis = $preis;
26      }
27
28      public function getBruttoPreis()
29      {
30          $bruttoPreis = $this->getPreis() * 1.07;
31          return $bruttoPreis;
32      }
33
34      public function setBruttoPreis($bruttoPreis)
35      {
36          $this->setPreis($bruttoPreis / 1.07);
37      }
38  }
```

Codebeispiel 55 *buch.php (Version 2)*

Beim Setter des virtuellen Attributs `$bruttoPreis` müssen Sie dieses Mal durch 1,07 teilen, um den Nettopreis zu erhalten.

6.3 Testen Sie Ihr Wissen!

1. Was unterscheidet virtuelle Attribute von echten Attributen?
2. Unter welchen Umständen sind virtuelle Attribute nützlich?
3. Was ist bei Settern von virtuellen Attributen zu beachten?

6.4 Übungen

Übung 18:

Schreiben Sie eine Klasse `Person` mit den Attributen `$vorname`, `$nachname` und `$geburtTimestamp` mit den entsprechenden Gettern und Settern. Nutzen Sie eine extra Datei *index.php*, um ein Objekt zu erstellen, dieses zu befüllen und den kompletten Namen auszugeben. Zum Testen können Sie der Einfachheit halber den heutigen Timestamp verwenden.

Übung 19:

`Person` soll das virtuelle Attribut `$name` mit dem Getter `getName()` enthalten, das den Vor- und Nachnamen enthält. Ändern Sie die Getter-Aufrufe in der *index.php* entsprechend ab.

6.5 Optionale Übungen

Übung 20:

`Person` soll das virtuelle Attribut `$geburtstag` mit Getter enthalten, das den Geburtstag im Format `tt.mm.` zurückgibt. Ergänzen Sie die Ausgabe des Geburtstags in Ihrer *index.php*.

Übung 21:

Schreiben Sie die Methode `setName()`, der Sie den Vor- und Nachnamen als einen String übergeben können. Die Methode soll den String in den Vor- und den Nachnamen zerlegen und den entsprechenden echten Attributen zuweisen. Verwenden Sie hierfür die PHP-Funktion `explode()`. Nehmen Sie der Einfachheit halber an, dass der Name nur ein Leerzeichen enthalten kann. Testen Sie den neuen Setter.

Übung 22:

Schreiben Sie eine Methode `setGeburtstag()`, der Sie ein Datum im Format `tt.mm.jjjj` übergeben können. Diese erzeugt daraus den ***Timestamp*** und speichert ihn in `$geburtTimestamp`. Verwenden Sie hierfür die PHP-Funktionen `strtotime()`. Ziehen Sie für die genaue Syntax http://php.net/ zu Rate. Testen Sie auch diesen Setter.

Magische Methoden

7

In dieser Lektion lernen Sie

- was eine magische Methode ist.
- wann die magischen Methoden `__toString()` und `__construct()` aktiviert werden.
- was beim Aufruf von `new` eigentlich passiert.
- wie Konstruktoren Ihnen das Erzeugen von Objekten erleichtern.

7.1 Das Problem

Je häufiger Sie mit Objekten arbeiten, desto mehr werden Sie feststellen, dass Sie bestimmte Dinge mit jedem Objekt einer Klasse anstellen. Jedes Mal, wenn Sie ein Person-Objekt erzeugen, rufen Sie danach die Methoden `setVorname()` und `setNachname()` auf.

Beispiel

```php
<?php

class Person
{
    protected $vorname = '';
    protected $nachname = '';

    public function getVorname()
    {
        return $this->vorname;
    }

    public function getNachname()
    {
        return $this->nachname;
    }

    public function setVorname($vorname)
    {
        $this->vorname = $vorname;
    }

    public function setNachname($nachname)
    {
        $this->nachname = $nachname;
    }
}
```

Codebeispiel 56 *person.php*

```php
<?php

require_once 'person.php';

$remolt = new Person();
$remolt->setVorname('Marc');
$remolt->setNachname('Remolt');

var_dump($remolt);
```

Codebeispiel 57 *neue_person.php (Version 1)*

Gegen diese Schreibweise ist nichts einzuwenden, nur wäre es sinnvoll, wenn Dinge, die jedes Mal beim Erzeugen eines Objekts gemacht werden sollen, auch automatisch gemacht werden. Es ist einfach unnötig, drei Zeilen Code zu schreiben, wo theoretisch nur eine notwendig ist.

7.2 Magische Methoden

7.2.1 Konzept

Keine Sorge, wir driften jetzt nicht in die Esoterik ab. Eine ***magische Methode*** (engl.: magic method) ist eine Methode, die Sie nicht aufrufen müssen, sondern die selbst weiß, wann sie gebraucht wird. Daher auch magisch - plötzlich tut sie etwas, ohne dass Sie es sagen mussten.

Diese Methoden haben einen festgelegten Namen und einen Auslöser. Wann immer dieser Auslöser aktiviert wird, sieht PHP nach, ob Sie die passende magische Methode definiert haben. Wenn ja, wird sie ausgeführt. Einige mögliche Auslöser sind:

- Sie versuchen, ein Objekt mit `echo` auszugeben.
- Ein neues Objekt einer bestimmten Klasse wird angelegt.
- Sie versuchen, eine nicht existierende Methode aufzurufen.
- Sie versuchen, lesend auf ein nicht existierendes Attribut zuzugreifen.

Für eine komplette Liste aller magischen Methoden und ihrer Auslöser in PHP möchte ich Sie auf die PHP-Referenz verweisen: http://php.net/de/language.oop5.magic.

7.2.2 Die Methode __toString()

Bevor wir uns an die Konstruktoren heranwagen, lassen Sie uns erst ein einfaches Beispiel für eine magische Methode betrachten, nämlich `__toString()`. Was als Erstes auffällt, ist der seltsame Name, der mit zwei Unterstrichen (engl.: Underscore) beginnt. Das ist ein Kennzeichen für eine magische Methode: Jede beginnt mit zwei Unterstrichen `__`.

Diese Methode wird immer dann aktiv, wenn Sie versuchen, ein Objekt mit `echo` auszugeben, was normalerweise fehlschlägt.

Beispiel

```
1 <?php
2
3 require_once 'person.php';
4
5 $remolt = new Person();
6 $remolt->setVorname('Marc');
7 $remolt->setNachname('Remolt');
8
9 echo $remolt; //Erzeugt einen Fehler
```

Codebeispiel 58 *neue_person.php - Fehler (Version 2)*

In Zeile 9 passiert genau das, was wir erwarten, es wird ein Fehler erzeugt. Es ist auch ziemlich abwegig, ein Objekt als Text ausgeben zu wollen. Wenn wir allerdings die magische Methode `__toString()` in der Klasse definieren, sieht die Sache ganz anders aus.

Beispiel

```php
<?php

class Person
{
    protected $vorname = '';
    protected $nachname = '';

    public function __toString()
    {
        return $this->getVorname() . ' ' . $this->getNachname();
    }

    public function getVorname()
    {
        return $this->vorname;
    }

    public function getNachname()
    {
        return $this->nachname;
    }

    public function setVorname($vorname)
    {
        $this->vorname = $vorname;
    }

    public function setNachname($nachname)
    {
        $this->nachname = $nachname;
    }
}
```

Codebeispiel 59 *person_to_string.php*

```php
<?php

require_once 'person_to_string.php';

$remolt = new Person();
$remolt->setVorname('Marc');
$remolt->setNachname('Remolt');

echo $remolt; //Gibt "Marc Remolt" aus
```

Codebeispiel 60 *neue_person.php (Version 2b)*

Sobald das Objekt `$remolt` im Zusammenhang mit `echo` aufgerufen wird, ruft PHP die Methode `__toString()` auf und verwendet den Rückgabewert an der Stelle im Code als Text.

Sie können die Methode `__toString()` also verwenden, um eine für Menschen lesbare Repräsentation des Objekts zu erhalten. Meistens nutzt man ohnehin eine derartige Methode, und so erhält man noch den Magie-Bonus.

Sie müssen allerdings beachten, dass `__toString()` immer einen String zurückgeben muss. Im Grunde ist das zwar logisch, es wird aber leider immer wieder vergessen.

7.3 Konstruktoren

7.3.1 Konzept

Als ***Konstruktor*** bezeichnet man eine spezielle Methode, die beim Erzeugen eines Objekts aufgerufen wird. In ihr sollten Sie alles abarbeiten, was zur Verwendung des Objekts notwendig ist. Möglichkeiten gibt es hier viele, daher nur ein paar Beispiele:

- Es werden Standardwerte für die Attribute gesetzt.
- Die Attribute werden auf Basis von Formulardaten befüllt.
- Das Objekt benötigt andere Objekte zum Betrieb. Diese werden im Konstruktor erzeugt.

7.3.2 Die Methode __construct()

Der Konstruktor könnte im Prinzip eine beliebige Methode sein, die Sie sofort aufrufen, nachdem Sie ein Objekt mit `new` erzeugt haben. Dieser Weg hat allerdings zwei Nachteile:

- Sie benötigen zwei Zeilen Code.
- Sie könnten den Methoden-Aufruf vergessen.

Es ist besser, sich die Arbeit von PHP abnehmen zu lassen. Wenn Sie Ihren Konstruktor `__construct()` nennen, wird diese Methode automatisch beim Aufruf von `new` ausgeführt.

Beispiel

```php
<?php

class TestKlasse
{
    public $text;

    public function __construct()
    {
        $this->text = 'Hallo Welt';
    }
}

$test = new TestKlasse();
echo $test->text; //Gibt "Hallo Welt" aus
```

Codebeispiel 61 *test.php (Version 1)*

Wann immer Sie also ein neues Objekt aus der Klasse `TestKlasse` erzeugen, wird das Attribut `$text` mit einem festen String befüllt. Das mag kein sonderlich sinnvolles Beispiel sein, aber es zeigt das Konzept.

7.3.3 Parameter an __construct() übergeben

Wie aber können Sie schon beim Erzeugen des Objekts Werte übergeben? Sie könnten einer Person schon beim Erzeugen einen Namen geben oder einen variablen Text in unserem vorherigen Beispiel nutzen.

Das zu erreichen ist nicht schwer, und Sie erfahren auch endlich, warum Sie an den Klassennamen beim Aufruf von `new` immer Klammern anhängen, als ob Sie eine Methode aufrufen würden. Die Antwort müssten Sie inzwischen schon kennen: Zwar rufen nicht Sie eine Methode auf, aber PHP tut es, nämlich `__construct()`.

Alles, was Sie als Parameter in die Klammern hinter dem Klassennamen schreiben, landet direkt als Parameter in der magischen Methode.

Beispiel

```php
<?php

class TestKlasse
{
    public $text;

    public function __construct()
    {
        $this->text = 'Hallo Welt';
    }
}

$test = new TestKlasse('Hallo Leute!');
echo $test->text; //Gibt immer noch "Hallo Welt" aus
```

Codebeispiel 62 *test.php (Version 2)*

Der Code in Zeile 13 führt dazu, dass in dem Objekt `$test` die Methode `__construct()` mit dem String `'Hallo Leute!'` als Parameter aufgerufen wird, also genauso, als hätten Sie `$test->__construct('Hallo Leute!')` von Hand aufgerufen. Letzteres ist technisch gesehen übrigens durchaus möglich, aber normalerweise durch den an einen Auslöser gebundenen automatisierten Aufruf unnötig. Durch den Aufruf ändert sich im Moment jedoch noch nichts im Objekt, da die Methode noch keine Parameter erwartet.

Beispiel

```php
<?php

class TestKlasse
{
    protected $text;

    public function __construct($eingabe)
    {
        $this->setText($eingabe);
    }

    public function setText($text)
    {
        $this->text = $text;
    }

    public function getText()
    {
        return $this->text;
    }
}

$test = new TestKlasse('Hallo Welt');
echo $test->getText(); //Gibt "Hallo Welt" aus

$test = new TestKlasse('Wie geht es euch denn so?');
echo $test->getText(); //Gibt "Wie geht es euch denn so?" aus
```

Codebeispiel 63 *test.php (Version 2b)*

Jetzt wird der String in der Methode `__construct()` im Attribut `$text` abgelegt und kann wie gewohnt weiterverarbeitet werden. Nebenbei haben wir auch noch den Code überarbeitet, indem wir einen passenden Getter und Setter geschrieben haben.

Lassen Sie uns nun das Problem vom Beginn dieser Lektion erneut aufgreifen. Der Aufruf könnte nun folgendermaßen aussehen:

Beispiel

```php
<?php

require_once 'person_konstruktor.php';

$remolt = new Person('Marc', 'Remolt');

echo $remolt; //Gibt "Marc Remolt" aus
```

Codebeispiel 64 *neue_person.php (Version 3)*

Natürlich müssen Sie noch die Klasse `Person` um einen Konstruktor erweitern, der den Vornamen und den Nachnamen als Parameter akzeptiert. Es ist also nicht wirklich eine Zeile, aber zumindest sehen Sie nur noch eine.

7.3.4 Ein assoziatives Array an den Konstruktor übergeben

Wenn Sie dem Konstruktor viele Werte übergeben müssen, bietet es sich an, dies als assoziatives Array zu tun. So müssen Sie nicht auf die Reihenfolge der Parameter achten oder können das Array auch vorbefüllen und dem Konstruktor als eine Variable übergeben.

Im Konstruktor weisen Sie dann die einzelnen Werte aus dem Array den Attributen des Objekts zu, indem Sie deren Setter verwenden.

Beispiel

```php
<?php

class Person
{
    protected $vorname = '';
    protected $nachname = '';

    public function __construct(array $daten = [])
    {
        if ($daten) {
            $this->setVorname($daten['vorname']);
            $this->setNachname($daten['nachname']);
        }
    }

    public function __toString()
    {
        return $this->getVorname() . ' ' . $this->getNachname();
    }

    public function getVorname()
    {
        return $this->vorname;
    }

    public function setVorname($vorname)
    {
        $this->vorname = $vorname;
    }

    public function getNachname()
    {
        return $this->nachname;
```

```
34      }
35
36      public function setNachname($nachname)
37      {
38          $this->nachname = $nachname;
39      }
40  }
```

Codebeispiel 65 *person_konstruktor_array.php*

```
1  <?php
2
3  require_once 'person_konstruktor_array.php';
4
5  $person = new Person(
6      [
7          'vorname' => 'Arthur',
8          'nachname' => 'Dent',
9      ]
10 );
11
12 echo $person;
```

Codebeispiel 66 *neue_person.php (Version 4)*

Der Konstruktor erhält als optionalen Parameter ein Array `$daten` und prüft über den TypeHint, ob er auch tatsächlich ein Array erhält. Die Methode selbst prüft, ob das Array Daten enthält. Wenn ja, werden die Setter der Attribute mit den passenden Schlüsseln des Arrays beliefert.

Ein weiterer Vorteil dieses Konstruktors ist übrigens, dass Sie ihn sehr gut zusammen mit Formularen verwenden können. Formulardaten liegen normalerweise in `$_POST` in Form eines assoziativen Arrays. Na, klingelt es schon? Mit einem derartig aufgebauten Konstruktor können Sie folgenden Code schreiben:

```
$person = new Person($_POST);
```

Das ist doch mal praktisch, oder?

Wir können die ganze Sache sogar noch ein wenig weiter treiben. Oftmals existiert ein namentlicher Zusammenhang zwischen den Array-Schlüsseln, den Attributen der Klasse und den Settern. So gehören zum Array-Schlüssel `vorname` das Attribut `$vorname` und der Setter `setVorname`.

Beispiel

```
1  <?php
2
3  class Person
4  {
5      protected $vorname = '';
6      protected $nachname = '';
7
8      public function __construct(array $daten = [])
9      {
10         // wenn $daten nicht leer ist, rufe die passenden Setter auf
11         if ($daten) {
12             foreach ($daten as $k => $v) {
13                 $setterName = 'set' . ucfirst($k);
14                 // pruefe ob ein passender Setter existiert
15                 if (method_exists($this, $setterName)) {
16                     $this->$setterName($v); // Setteraufruf
17                 }
18             }
19         }
```

```
20      }
21
22      // gekuerztes Beispiel ohne __toString und Getter/Setter
23  }
```

Codebeispiel 67 *person_konstruktor_schleife.php*

Das Beispiel nutzt genau diesen namentlichen Zusammenhang, indem das Array `$daten` in einer `foreach`-Schleife durchlaufen wird. Von jedem Schlüssel `$k` wird der Anfangsbuchstabe in einen Großbuchstaben umgewandelt (`ucfirst`) und davor der String `set` ergänzt. Aus dem Schlüssel `vorname` wird so also der Methodenname `setVorname`.

Mit der Funktion `method_exists()` können Sie prüfen, ob eine Methode mit einem bestimmten Namen in einem Objekt existiert. In unserem Beispiel wird in Zeile 15 also bestätigt, dass es die Methode `setVorname` in dem aktuellen Objekt gibt.

Handelt es sich um eine gültige Methode, so rufen wir diese in Zeile 16 auf und übergeben den Array-Wert `$v` als Parameter (`$this->$setterName($v)`). Hierfür nutzen wir das Konzept der ***Variablenfunktionen***: Wenn Sie an das Ende einer Variablen Klammern hängen, versucht PHP eine Funktion aufzurufen, deren Name der aktuelle Wert der Variablen ist. Dies gilt natürlich auch für die Methoden eines Objektes, wenn wir noch `$this` und den Pfeil-Operator davorschreiben.

Die Konstruktor-Methode durchläuft also das Array `$daten`, und wenn es einen Setter passend zum Schlüssel gibt, wird dieser aufgerufen. Schlüssel, zu denen es kein Attribut im Objekt gibt, werden einfach ignoriert.

Allerdings löst unser schöner neuer Konstruktor nicht alle Probleme. Häufig muss man ein existierendes Objekt aktualisieren.

Beispiel

```
1  <?php
2
3  require_once 'person_konstruktor_array.php';
4
5  $person = new Person(
6      [
7          'vorname' => 'Marc',
8          'nachname' => 'Remolt',
9      ]
10 );
11
12 //hier passieren viele Dinge ...
13
14 $formularDaten = [
15     'vorname' => 'Jan',
16     'nachname' => 'Teriete',
17 ];
18
19 $person->setVorname($formularDaten['vorname']);
20 $person->setNachname($formularDaten['nachname']);
21
22 echo $person;
```

Codebeispiel 68 *neue_person.php (Version 5)*

Wie im Beispiel veranschaulicht, würde man für diese Aktualisierung weiterhin jeden Setter einzeln aufrufen. Nicht wirklich schön, oder?

Übung 23:

1. Sie haben in dieser Lektion gelernt, wie Sie alle Setter aufrufen können, ohne dass Sie jeden Aufruf einzeln notieren müssen. Überlegen Sie, wie das ging.
2. Probieren Sie es nun aus. Ist der entstandene Code lesbar?

Wenn Sie sich an den Abschnitt 7.3.3 zurückerinnern, so kennen Sie eine Schreibweise für den manuellen Aufruf einer magischen Methode. In diesem Fall benötigen Sie den Konstruktor, der dann automatisch jeden Setter aufruft, zu dem ein Array-Schlüssel existiert.

Allerdings wäre Ihr Code nicht besonders gut lesbar, wenn Sie für eine Aktualisierung von Objekt-Daten jedesmal den Konstruktor aufrufen würden. Zudem besteht die Möglichkeit, dass der Konstruktor weitere Aufgaben neben dem Aufruf der Setter wahrnimmt. Doch wie lässt sich dieses Problem lösen?

Beispiel

```php
<?php

class Person
{
    protected $vorname = '';
    protected $nachname = '';

    public function __construct(array $daten = [])
    {
        $this->setDaten($daten);
    }

    public function setDaten(array $daten)
    {
        // wenn $daten nicht leer ist, rufe die passenden Setter auf
        if ($daten) {
            foreach ($daten as $k => $v) {
                $setterName = 'set' . ucfirst($k);
                // pruefe ob ein passender Setter existiert
                if (method_exists($this, $setterName)) {
                    $this->$setterName($v); // Setteraufruf
                }
            }
        }
    }

    public function getVorname()
    {
        return $this->vorname;
    }

    public function setVorname($vorname)
    {
        $this->vorname = $vorname;
    }

    public function getNachname()
    {
        return $this->nachname;
    }

    public function setNachname($nachname)
```

```
43      {
44          $this->nachname = $nachname;
45      }
46  }
```

Codebeispiel 69 *person_set_daten.php*

Code, der eine klar definierte Aufgabe hat (z. B. Aufruf der Setter auf Basis des assoziativen Arrays), lässt sich normalerweise auslagern (Stichwort ***Single Responsibility Principle***). In diesem Fall soll diese Funktionalität fest an die Objekte der `Person`-Klasse gebunden sein, weswegen wir den Code in die Methode `setDaten()` auslagern und diese im Konstruktor in Zeile 10 aufrufen. Wann immer Sie die Daten eines bestehenden Objekts aktualisieren wollen, können Sie fortan diese Methode nutzen.

7.4 Testen Sie Ihr Wissen!

1. Was versteht man unter einer ***magic method***?
2. Wann wird die Methode `__toString()` einer Klasse automatisch aufgerufen?
3. Wann wird die Methode `__construct()` einer Klasse automatisch aufgerufen?
4. Benötigt jede Klasse eine Konstruktor-Methode?
5. Sie haben ein `Person`-Objekt in der Variable `$person` und möchten dessen Inhalt (oder zumindest einen Teil davon) als Text ausgeben. Die in dieser Lektion vorgestellten magischen Methoden sind vorhanden. Was können Sie aufrufen, wenn Sie keine Getter verwenden dürfen?

7.5 Übungen

Übung 24:

Erweitern Sie die Klasse `Fussball` aus den Aufgaben von Lektion 5, so dass sie über einen Konstruktor verfügt, dem die Attribute des Fussballs direkt übergeben werden können. Verwenden Sie mehrere Parameter. Erzeugen Sie mehrere Bälle und geben Sie deren Beschreibung mittels `beschreibeFussball()` aus.

Übung 25:

Ersetzen Sie die Methode `beschreibeFussball()` durch die magische Methode `__toString()`. Passen Sie auch die Ausgabe in der *index.php* an.

7.6 Optionale Übungen

Übung 26:

Ändern Sie die Klasse `Fussball` und verwenden Sie nun einen einzelnen Parameter als Array. Erweitern Sie hierfür die Klasse um die Methode `setDaten()`.