

Objektorientiertes PHP8

Zusammenfassung und Ergänzungen **WBT**

Bitte die Lizenzbedingungen am Ende des Dokuments beachten!

Inhaltsverzeichnis

- 1. Klassen und Objekte
 - 1.1. Klassen erzeugen
 - 1.2. Objekt aus eigener Klasse instanziiieren
 - 1.3. Attribute
 - 1.3.1. Sichtbarkeit von Attributen
 - 1.4. Methoden
 - 1.4.1. Zusammenfassung
 - 1.5. Zugriff auf Attribute und Methoden innerhalb einer Klasse
 - 1.6. Die Variable `$this`
 - 1.6.1. Namenskonventionen
 - 1.7. Getter- und Setter-Methoden
 - 1.7.1. Kapselung
 - 1.7.2. Sichtbarkeit
 - 1.7.3. Getter-Methoden (Getter)
 - 1.7.4. Setter-Methoden (Setter)
 - 1.8. Arbeiten mit Objekten
 - 1.8.1. Methoden innerhalb eines Objektes aufrufen
 - 1.8.2. Objekte als lokale Variablen
 - 1.8.3. Objekte als Parameter übergeben
 - 1.8.4. Verhalten von Kopien von Objekten
 - 1.8.5. Objekte in superglobalen Variablen
 - 1.8.6. Objekte in Attributen ablegen
 - 1.8.7. Informationen zu Objekten abfragen
 - 1.8.8. Typdeklarationen (früher type hinting bzw. type hints)
 - 1.9. Virtuelle Attribute
 - 1.10. Magische Methoden
 - 1.10.1. Magische Methode `__toString()`
 - 1.10.2. Magische Methode `__construct()`
 - 1.10.3. Funktion `method_exists()`
 - 1.10.4. Variablenfunktionen
 - 1.10.5. `$_POST` oder `$_GET` in `__construct()`
 - 1.10.6. Sinnvolle Namensgebung
- 9. Zu dieser Dokumentation / Lizenzbedingungen
 - 9.1. Lizenzangaben

1. Klassen und Objekte

Dreh- und Angelpunkt der objektorientierten Programmierung (kurz OOP) bilden Objekte. In der realen Welt bezeichnet man oftmals lediglich greifbare Gegenstände als Objekte. Aus Sicht der Sprache PHP sind Objekte in ihrer Begrifflichkeit nicht so stark begrenzt und erst einmal nur ein weiterer Datentyp wie beispielsweise Arrays. In PHP ist also ein Lebewesen wie eine Person oder ein Tier durchaus als Objekt darstellbar.

 [Quelle tw. WBT 3.2](#)

Jedes Objekt übernimmt bei der sogenannten **Instanziierung** sämtliche **Attribute** und **Methoden** einer Klasse. Die Klasse ist somit eine Art Prototyp bzw. ein Bauplan, in dem Sie alles definieren, was später von den Objekten benötigt wird. Eigene Klassen werden mit dem Schlüsselwort **class** definiert. Mithilfe eines solchen Bauplans und des Schlüsselworts **new** lassen sich Objekte als individualisierbare Varianten einer Klasse instanziiieren.

```
<?php

$test = new stdClass();
echo gettype($test); // "object"
```

(Beispiel: oop01.php)

In PHP gibt es die Standardklasse **stdClass**, die leer ist und weder Attribute noch Methoden enthält. Sie eignet sich sehr gut für ein kleines Beispiel zur Instanziierung, wird im weiteren Verlauf jedoch nicht weiter relevant sein.

Im Gegensatz zu beispielsweise einem assoziativen Array kann ein Objekt nicht nur Daten in seinen Attributen sondern zusätzlich auch noch Methoden enthalten. **Methoden sind Funktionen die an Klassen gebunden sind**. Methoden lassen sich auf Objekten aufrufen, die aus einer Klasse erzeugt wurden, die diese Methode enthält. Die Methoden von Objekten der gleichen Klasse sind identisch die Werte der Attribute können sich hingegen unterscheiden.

 [Quelle ursprünglich WE Buch Objektorientiertes PHP7 Band 2: MySQL und Doctrine 2](#)

kurz - Objekte in PHP:

- haben den Datentyp **object**
- werden mit dem Schlüsselwort **new** erzeugt.
- können aus der allgemeinen Objektgruppe **stdClass** instanziiert werden.
- sind gleich sofern sich Objektgruppe und Eigenschaften (bzw. genauer deren Werte) nicht unterscheiden.
- sind identisch sofern sie die gleiche Identität haben.

 [Quelle WBT 3.2](#)

1.1. Klassen erzeugen

Klassen in PHP werden durch das Schlüsselwort `class` gefolgt von dem **Klassenbezeichner** deklariert. Danach folgt in geschweiften Klammern (`{}`) der Klassenkörper.

```
<?php
class EigeneKlasse {
    // ...
}
```

1.2. Objekt aus eigener Klasse instanziiieren

Dies erfolgt wie bei der Standardklasse mittels des Schlüsselworts `new` und dem Klassennamen

```
<?php
$einObjekt = new EigeneKlasse();
```

- Objekte in PHP beschreiben nicht zwingend einen greifbaren Gegenstand der realen Welt.
- Klasse ist der Fachbegriff für die Gruppe bzw. den Bauplan eines Objektes.
- Eigene Klassen werden mit dem Schlüsselwort `class` definiert.
- Klassennamen werden immer in der Einzahl geschrieben und es hat sich eingebürgert, den ersten Buchstaben eines Klassennamens großzuschreiben.
- Es macht Sinn, die Definition jeder einzelnen Klasse in eine eigene Datei auszulagern um eine optimale Wiederverwendbarkeit zu ermöglichen.

 [Quelle WBT 3.2](#)

1.3. Attribute

Die Eigenschaften von Klassen bzw. daraus instanziierten Objekten bezeichnet man als **Attribute**. Mittels des Pfeiloperators kann auf die Attribute zugegriffen (lesend oder schreibend).

```
<?php
$einObjekt = new EigeneKlasse();
$einObjekt->attribut1 = "Wert"; // Wert einem Attribut zuweisen
echo $einObjekt->attribut1; // Wert eines Attributs auslesen
```

Achtung: Ein schreibender Zugriff auf ein Attribut, das noch nicht existiert erzeugt dieses Attribut ⇒ das kann u. U. nicht vorhersehbaren Folgen haben, z. B. wenn man sich versehentlich beim Attributsbezeichner verschreibt, dann wird eine neues Attribut mit diesem 'fehlerhaften' Namen erzeugt und der Wert diesem Attribut und nicht dem eigentlich angedachten zugewiesen.

1.3.1. Sichtbarkeit von Attributen

Attribute können bereits in der Klassendefinition deklariert werden. Es empfiehlt sich, diese bereits mit einem Standardwert zu initialisieren und somit quasi den Datentyp zu bestimmen (natürlich kann sowohl der Wert als auch der Datentyp ähnlich wie bei Variablen später wieder geändert werden).

Die Schlüsselwörter `public` und `protected` legen den Gültigkeitsbereich der Attribute fest und steuern somit die Sichtbarkeit der Attribute, d. h. die Zugriffsmöglichkeiten von außerhalb der Instanzen.

- `public` bedeutet es kann von außerhalb mittels des Pfeiloperators `->` zugegriffen werden
- `protected` bedeutet, dass kein Zugriff von außen möglich ist - das Attribut ist somit nicht sichtbar wie wenn es nicht deklariert wäre

1.4. Methoden

Methoden sind Funktionen, die an Klassen gebunden sind. Methoden können auf Objekten aufgerufen werden, die aus einer Klasse erzeugt wurden, die diese Methode enthält.

Methoden sind quasi die Funktionen der Objekte die aus den Klassen instanziiert werden. Auf sie kann analog den Attributen mittels des Pfeiloperators `->` zugegriffen werden.

Die Sichtbarkeit von Methoden, d. h. die Zugriffsmöglichkeiten von außerhalb der Instanzen kann ebenfalls mittels der Schlüsselwörter `public` und `protected` gesteuert werden.

```
<?php
class EigeneKlasse {
    public function eigeneMethode1($parameter1, $parameter2, $parameter3) {
        // Code der Methode 1
    }
    protected function eigeneMethode2($parameter1, $parameter2, $parameter3) {
        // Code der Methode 2
    }
}

$object = new EigeneKlasse();

$object->eigeneMethode1(/*Argumente*/); // ruft Methode 1 auf
$object->eigeneMethode2(/*Argumente*/); // erzeugt einen Fehler - Methode 2 ist
protected also von außen nicht sichtbar
```

1.4.1. Zusammenfassung

Jedes Objekt einer Klasse übernimmt bei der Instanziierung sämtliche Attribute und Methoden dieser Klasse. Die Klasse ist somit eine Art Prototyp bzw. ein Bauplan, in dem Sie alles definieren, was später benötigt wird. Mit Hilfe dieses Bauplans können dann individualisierbare Varianten (die sogenannten Objekte) instanziiert werden.

Die Attribute, etwa der Preis, können und werden sich zukünftig von Instanz zu Instanz unterscheiden. Beachten Sie, dass der Begriff Attribut nicht nur die Eigenschaft bezeichnen kann, sondern auch den hierin abgelegten Wert. Hier meine ich den Wert.

Außerdem können zwei verschiedene Klassen identische Bezeichner für Attribute und Methoden verwenden. Sie können zwei Methoden, die das Gleiche tun, also gleich benennen, sofern diese in unterschiedlichen Klassen definiert werden. Durch ihre Bindung an die Klasse bleiben sie trotzdem eindeutig.

 [Quelle WBT 3.4.2](#)

1.5. Zugriff auf Attribute und Methoden innerhalb einer Klasse

```
<?php
class EigeneKlasse {
    public $attr1 = "Wert1";
    protected $attr2 = "Wert2";
    public function eigeneMethode1() {
        echo $attr1; // erzeugt einen Fehler
    }
    protected function eigeneMethode2() {
        echo $attr1; // erzeugt einen Fehler
    }
}
```


Ein direkter Zugriff auf die Attribute einer Klasse ist auch innerhalb der Klasse nicht möglich!

1.6. Die Variable `$this`

Die Pseudovariablen `$this` ist verfügbar, falls eine Methode aus einem Objektkontext heraus aufgerufen wird. `$this` ist eine Referenz auf das aufrufende Objekt (üblicherweise das Objekt, zu dem die Methode gehört, aber möglicherweise ein anderes Objekt, falls die Methode statisch aus dem Kontext eines zusätzlichen Objektes aufgerufen wird).

 [php.net](#)

Die Variable `$this` ermöglicht es innerhalb einer Klasse auf die Attribute und die Methoden der Klasse zuzugreifen ohne dass bereits ein Objekt aus der Klasse instanziiert wurde. Es stellt also eine Referenz auf die Klasse bzw. auf das später instanziierte Objekt dar. Sie ist insofern notwendig, da wir bei der Klassendefinition noch gar nicht wissen, wie die Objekte, die hieraus instanziiert werden, später mal heißen werden.

Wann immer Sie in einer Klasse auf ein Attribut oder eine Methode eines aus dieser Klasse instanziierten Objektes zugreifen wollen, müssen Sie die spezielle Variable `$this` verwenden. Also müssen Sie `$this->attribut` oder `$this->method()` schreiben.  [Quelle WBT 3.4.3](#)

1.6.1. Namenskonventionen

1. **Klassen** - Klassennamen verwenden den **UpperCamelCase**, beginnen also mit einem Großbuchstaben. Bei zusammengesetzten Wörtern wird jedes Wort großgeschrieben. Klassennamen sind i. R. Nomen und werden immer in Einzahl geschrieben.
2. **Attribute** - Attribute verwenden den **lowerCamelCase**, werden also kleingeschrieben. Bei zusammengesetzten Wörtern wird ab dem 2-ten Wort jedes Wort großgeschrieben. Also analog der Namensgebung von Variablen.
3. **Methoden** - Für Methoden gelten dieselben Regeln wie für Attribute sowie für die Namensgebung von Funktionen, der Name sollte immer mit einem Verb beginnen.

1.7. Getter- und Setter-Methoden

Der Sinn von Getter- und Setter-Methoden ist es, die internen Klassenvariablen gegen Zugriff von außen zu schützen.

Getter- und Setter-Methoden sind 'normale' Methoden, durch die man von Außen auf die Attribute lesend oder schreibend zugreifen und diese ggf. dabei noch ändern kann.

1.7.1. Kapselung

Als Datenkapselung (englisch encapsulation, nach David Parnas auch bekannt als information hiding) bezeichnet man in der Programmierung das **Verbergen** von *Daten* oder *Informationen* vor dem Zugriff von außen. Der direkte Zugriff auf die interne Datenstruktur wird unterbunden und erfolgt stattdessen über **definierte Schnittstellen** (*Black-Box-Modell*).

 [Wikipedia](#)

Ein grundlegendes Prinzip der objektorientierten Programmierung ist die **Kapselung**. Kapselung bedeutet, die tatsächliche Funktionalität eines Codeblocks in Funktionen oder in Methoden einer Klasse zu verbergen.

 [Quelle WBT 4.2](#)

Bei der sogenannten **Datenkapselung** werden die Attribute nach außen hin versteckt und der Zugriff auf sie über Methoden ermöglicht. Man spricht hier auch vom **Interface** einer Klasse, was nichts anderes als die Sicht von außen meint.

 [Quelle WBT 4.2](#)

Versuchen Sie immer, das Interface einer Klasse stabil zu halten. Jede Änderung an Methodenaufrufen zieht immer mehr Arbeit nach sich, je größer die Projekte werden.

 [Quelle WBT 4.2](#)

1.7.2. Sichtbarkeit

Die **Sichtbarkeit** einer **Eigenschaft**, **Methode** oder (von PHP 7.1.0 an) einer **Konstante** kann definiert werden, indem man der Deklaration eines der Schlüsselwörter **public**, **protected** oder **private** voranstellt.

- Auf **public** deklarierte Elemente kann von überall her zugegriffen werden.
- **Protected** beschränkt den Zugang auf Elternklassen und abgeleitete Klassen (sowie die Klasse, die das Element definiert).
- **Private** grenzt die Sichtbarkeit einzig auf die Klasse ein, die das Element definiert.

 [php.net](#)

⇒ Klasseneigenschaften (**Attribute**) und **Methoden** müssen als **public**, **private** oder **protected** definiert werden, anderenfalls sind sie immer public (gilt insbesondere auch bei Deklarationen mit dem Schlüsselwort **var** (veraltet))!

1.7.3. Getter-Methoden (Getter)

Der Bezeichner einer Methode, die den Wert eines Attributs zurückgibt, beginnt mit `get`, gefolgt vom Namen des Attributs. Es wird die übliche Namenskonvention für Methoden verwendet.



Quelle WBT 4.3


```
<?php
class User {
    protected $email = 'max@mustermann.de';
    // direkter Zugriff von außerhalb nicht möglich

    public function getEmail() {
        return $this->email;
    }
}

$user = new User();
echo $user->getEmail(); //=> max@mustermann.de
?>
```

Mittels des Getters ist es möglich für die Ausgabe noch Änderungen vorzunehmen (z. B. alles in Groß- oder Kleinbuchstaben umwandeln, Berechnungen durchführen etc.). So können auch später noch Änderungen vorgenommen werden, ohne dass die Schnittstelle an sich verändert wird! ⇒ Vorteil Wartbarkeit und Pflege.

1.7.4. Setter-Methoden (Setter)

Der Bezeichner einer Methode, die den Wert eines Attributs direkt (oder indirekt mittels virtuellen Attributen) verändert (setzt) beginnt mit **set**, gefolgt vom Namen des Attributs. Es wird die übliche Namenskonvention für Methoden verwendet.  [Quelle WBT 4.4](#)

```
<?php
class User
{
    protected $email = '';
    // Attribut mit leerem String initialisiert, direkter Zugriff von außerhalb
    nicht möglich

    public function getEmail()
    {
        return $this->email;
    }

    public function setEmail($email)
    {
        $this->email = $email;

        return $this;
    }
}

$user = new User();
$user->setEmail('erika@mustermann.de');
echo $user->getEmail(); //=> erika@mustermann.de
```

(Beispiel: oop02.php)

Versuchen Sie – wo immer möglich – nur im Inneren von Klassen etwas zu ändern. Auf diese Weise müssen Sie den Code, der mit den Objekten arbeitet nicht anpassen. Sie sparen viel Zeit und Ärger.

 [Quelle WBT 4.4](#)

1.8. Arbeiten mit Objekten

1.8.1. Methoden innerhalb eines Objektes aufrufen

```
<?php
class User
{
    protected $email = '';
    protected $name = 'Erika Mustermann'; // hier im Beispiel kein Setter sondern
    direkt initialisiert

    public function getEmail()
    {
        return $this->email;
    }

    public function getName()
    {
        return $this->name;
    }

    public function setEmail($email)
    {
        $this->email = $email;

        return $this;
    }

    public function gebeEmailInfoAus()
    {
        $ausgabe = 'Die E-Mail-Adresse von ' . $this->getName() . ' lautet: ' .
        $this->getEmail();

        echo $ausgabe;
    }
}

$user = new User();
$user->setEmail('erika@mustermann.de');
$user->gebeEmailInfoAus(); //=> Die E-Mail-Adresse von Erika Mustermann lautet:
erika@mustermann.de
```

(Beispiel: oop03.php)

Auch der interne Zugriff auf Attribute aus Methoden sollte idealerweise mittels der Getter-Methode erfolgen.

1.8.2. Objekte als lokale Variablen

Objekte lassen sich innerhalb von Methoden instanziiieren und wie eine lokale Variable verwenden.

```
<?php
class User
{
    // ... wie im Beispiel zuvor

    public function gebeEmailInfoAus()
    {
        $ausgabe = 'Die E-Mail-Adresse von ' . $this->getName() . ' lautet: ' .
$this->getEmail();

        return $ausgabe; // echo durch return ersetzt
    }
}

class Adresse
{
    protected $strasse = 'Heidestraße 17';
    protected $ort = 'München';

    public function gebeKontaktAus()
    {
        $user = new User();
        $user->setEmail('erika@mustermann.de');
        $ausgabe = 'Die Anschrift von ' . $user->getName() . ' lautet: ' . "\n";
        $ausgabe .= $this->strasse . ' in ' . $this->ort . "\n";
        $ausgabe .= $user->gebeEmailInfoAus();

        echo $ausgabe;
    }
}
$adresse = new Adresse();
$adresse->gebeKontaktAus();

//=> Die Anschrift von Erika Mustermann lautet:
//=> Heidestraße 17 in München
//=> Die E-Mail-Adresse von Erika Mustermann lautet: erika@mustermann.de
```

(Beispiel: oop04.php)

1.8.3. Objekte als Parameter übergeben

Objekte können auch Argument, d. h. als Wert für einen Parameter an eine Methode (oder auch Funktion) übergeben. Objekte verhalten sich hierbei genau so wie die anderen Datentypen.

```
<?php
class User
{
    // ... wie im Beispiel zuvor
}

class Adresse
{
    protected $strasse = 'Heidestraße 17';
    protected $ort = 'München';

    public function gebeKontaktAus($benutzer)
    {
        $ausgabe = 'Die Anschrift von ' . $benutzer->getName() . ' lautet: ' .
"\n";
        $ausgabe .= $this->strasse . ' in ' . $this->ort . "\n";
        $ausgabe .= $benutzer->gebeEmailInfoAus();

        echo $ausgabe;
    }
}

$user = new User();
$user->setEmail('erika@mustermann.de');

$adresse = new Adresse();
$adresse->gebeKontaktAus($user);

//=> Die Anschrift von Erika Mustermann lautet:
//=> Heidestraße 17 in München
//=> Die E-Mail-Adresse von Erika Mustermann lautet: erika@mustermann.de
```

(Beispiel: oop05.php)

1.8.4. Verhalten von Kopien von Objekten

```
<?php
class User
{
    // ... wie im Beispiel zuvor
}

class Adresse
{
    // ... wie im Beispiel zuvor
}

$user = new User();
$user->setEmail('erika@mustermann.de');

$neuuser = $user; // Erzeugung einer 'Kopie' des Objektes $user
$user->setEmail('maxfrau@mustermann.de'); // Ändern des Attributs im Original Objekt
$user

$adresse = new Adresse();
$adresse->gebeKontaktAus($neuuser); // Übergabe der 'Kopie' des Objektes $user

//=> Die Anschrift von Erika Mustermann lautet:
//=> Heidestraße 17 in München
//=> Die E-Mail-Adresse von Erika Mustermann lautet: maxfrau@mustermann.de
```

(Beispiel: oop06.php)

⇒ Die Wertzuweisung eines Objektes an eine Variable erzeugt keine Kopie des Objektes sondern es handelt sich um eine Referenz!

```
...
var_dump($user);
//=> object(User)#1 (2) {
//=>   ["email":protected]=>
//=>   string(22) "maxfrau@mustermann.de"
//=>   ["name":protected]=>
//=>   string(16) "Erika Mustermann"
//=> }

var_dump($neuuser);
//=> object(User)#1 (2) {
//=>   ["email":protected]=>
//=>   string(22) "maxfrau@mustermann.de"
//=>   ["name":protected]=>
//=>   string(16) "Erika Mustermann"
//=> }
```

(Beispiel: oop06.php)

⇒ es ist also ein und dasselbe Objekt in 2 Variablen gespeichert => Referenz (vgl. Index nach #)

Soll hingegen eine 'echte' Kopie eines Objektes als neues Objekt erzeugt werden, so kann das mit durch die Nutzung des Schlüsselwortes `clone` (welches wenn möglich die `__clone()`-Methode des Objektes aufruft) erfolgen. (Die `__clone()`-Methode eines Objektes kann nicht direkt aufgerufen werden!)

```
$kopie_des_objektes = clone $objekt;
```

```
<?php
class User
{
    // ... wie im Beispiel zuvor
}

class Adresse
{
    // ... wie im Beispiel zuvor
}

$user = new User();
$user->setEmail('erika@mustermann.de');

$neuuser = clone $user; // Erzeugung einer 'Kopie' des Objektes $user
$user->setEmail('maxfrau@mustermann.de'); // Ändern des Attributs im Original Objekt
$user

$adresse = new Adresse();
$adresse->gebeKontaktAus($neuuser); // Übergabe der 'Kopie' des Objektes $user

//=> Die Anschrift von Erika Mustermann lautet:
//=> Heidestraße 17 in München
//=> Die E-Mail-Adresse von Erika Mustermann lautet: erika@mustermann.de
```

(Beispiel: oop07.php)

```
...
var_dump($user);
//=> object(User)#1 (2) {
//=>   ["email":protected]=>
//=>   string(22) "maxsfrau@mustermann.de"
//=>   ["name":protected]=>
//=>   string(16) "Erika Mustermann"
//=> }

var_dump($neuuser);
//=> object(User)#2 (2) {
//=>   ["email":protected]=>
//=>   string(19) "erika@mustermann.de"
//=>   ["name":protected]=>
//=>   string(16) "Erika Mustermann"
//=> }
```

(Beispiel: oop07.php)

⇒ also zwei unterschiedliche Objecte (s. Index nach #)

1.8.5. Objekte in superglobalen Variablen

Bei jedem neuen Request (bspw. Aufruf der Folgeseite) müssen Objekte jedes mal neu erzeugt werden. Allerdings gibt es in PHP die Möglichkeit Objekte wie die andere Datentypen (Strings, Integers, Arrays... - alles außer dem der Datentyp Resource) Objekte in der Session abzulegen, d. h. in der Superglobal `$_SESSION` zu speichern.

1. Vorgängerseite - Speichern in der Session

```
<?php
class User
{
    // ... wie im Beispiel zuvor
}

session_start(); // bisher immer bevor eine Ausgabe gemacht wird!

$user = new User();
$user->setEmail('erika@mustermann.de');
# speichern in der Session
$_SESSION['erika'] = $user;
...
```

(Beispiel: `oop08a.php` - via Webserver aufrufen)

2. Folgeseite - Auslesen aus der Session

Wichtig: es müssen zuerst die Klassendefinitionen bereitgestellt werden, bevor die Session gestartet wird, sonst erfolgt ein Laufzeitfehler.

```
<?php
# erst die Klassen definieren bzw. externe Dateien mit den Klassendefinitionen
laden/einbinden (require_once)
class User
{
    // ... wie im Beispiel zuvor
}
# dann die Session starten
session_start();
# jetzt kann das Objekt aus der Session geladen werden
$user = $_SESSION['erika'];

$adresse = new Adresse();
$adresse->gebeKontaktAus($user);
...
```

(Beispiel: `oop08b.php`)

⇒ **Wichtig:** Das Objekt darf ausschließlich serialisierbare Daten gespeichert haben, also keine Ressourcen!

Selbstverständlich kann auch direkt auf das gespeicherte Objekt in der Superglobal zugegriffen werden
- dies ist aber eher unschön hinsichtlich der Lesbarkeit des Codes.

1.8.6. Objekte in Attributen ablegen

Objekte lassen sich analog den anderen Datentypen auch in den Attributen andere Objekte speichern (entweder durch eine direkte Zuweisung oder mittels eines Setters).

Eine Instanziierung bei der Initialisierung des Attributs geht nicht, da der zu setzende Wert ein konstanter Wert sein muss - d.h. dieser muss zum Kompilierungszeitpunkt ausgewertet werden können und darf nicht von Informationen abhängen, die erst zur Laufzeit zur Verfügung stehen!


```
<?php

class User
{
    protected $name = 'Oli';
}

class Name
{
    protected $name = 'Oli';
    // protected $name = new User(); # geht nicht da kein konstanter Wert!
    public $name2;
    protected $name3;

    public function setName3($name3)
    {
        $this->name3 = $name3;

        return $this;
    }

    public function gebeAus()
    {
        var_dump($this->name);
        var_dump($this->name2);
        var_dump($this->name3);
    }
}

$oli = new User();
$name = new Name();
$name->name2 = $oli;

$oliver = new User();
$name->setName3($oliver);

$name->gebeAus();
//=> string(3) "Oli"
//=> object(User)#1 (1) {
//=>   ["name":protected]=>
//=>   string(3) "Oli"
//=> }
//=> object(User)#3 (1) {
//=>   ["name":protected]=>
//=>   string(3) "Oli"
//=> }
```

(Beispiel: oop09.php)

1.8.7. Informationen zu Objekten abfragen

1.) Datentyp abfragen

a) mittels `gettype()`

```
gettype ( mixed $var ) : string
```

Liefert den Datentyp der Variablen var. Zur Typprüfung sollten die `is_*` Funktionen verwendet werden.



Beispiel:

```
<?php

class User
{
    protected $email = ' max@mustermann.de';

    public function getEmail()
    {
        return $this->email;
    }
}

$user = new User();
echo gettype($user); //=> object
```

(Beispiel: oop10.php)

b) mittels `is_object()`

```
is_object ( mixed $var ) : bool
```

Prüft ob die gegebene Variable ein Objekt ist.



Beispiel:

```
<?php

class User
{
    protected $email = ' max@mustermann.de';

    public function getEmail()
    {
        return $this->email;
    }
}

$user = new User();
echo is_object($user); //=> 1 #entspricht true
```

(Beispiel: oop11.php)

2.) Informationen abfragen

a) mittels `var_dump()`

```
var_dump ( mixed $expression [ mixed $... ] ) : void
```

Die Funktion gibt strukturierte Informationen über einen oder mehrere Ausdrücke aus, darunter auch den entsprechenden Typ und Wert. Arrays und Objekte werden rekursiv durchlaufen und die jeweiligen Werte eingerückt dargestellt, um die Struktur zu verdeutlichen.

Alle öffentlichen (**public**), privaten (**private**) und geschützten (**protected**) **Eigenschaften** eines Objekts werden in der Ausgabe dargestellt, außer wenn das Objekt eine `__debugInfo()` Methode implementiert.



Beispiel:

```
<?php

class User
{
    protected $email = ' max@mustermann.de';

    public function getEmail()
    {
        return $this->email;
    }
}

$user = new User();
var_dump($user);
//=> object(object(User)#1 (1) {
//=>    ["email":protected]=>
//=>    string(18) " max@mustermann.de"
//=> }
```

(Beispiel: `oop12.php`)

b) mittels `print_r()`

```
print_r ( mixed $expression [, bool $return = FALSE ] ) : mixed
```

`print_r()` zeigt Informationen über eine Variable in menschenlesbarer Form an.



Beispiel:

```
<?php
class User
{
    protected $email = ' max@mustermann.de';

    public function getEmail()
    {
        return $this->email;
    }
}
$user = new User();
print_r($user);
//=> User Object
//=> (
//=>     [email:protected] =>  max@mustermann.de
//=> )
```

(Beispiel: `oop13.php`)

c) weitere Möglichkeiten

- `var_export` — Gibt den Inhalt einer Variablen als parsbaren PHP-Code zurück, d. h. es liefert strukturierte Informationen zum Inhalt der übergebenen Variable. Das Verhalten ist ähnlich dem der `var_dump()`, allerdings ist hier das Ergebnis valider PHP-Code mit dem sich der Inhalt der Variable wieder herstellen lässt.
- `__debugInfo()` - Diese (magische) Methode wird von `var_dump()` aufgerufen, wenn ein Objekt ausgegeben wird, um die Eigenschaften auszulesen die gezeigt werden sollen. Wenn diese Methode in einem Objekt nicht definiert ist so werden alle **Eigenschaften** die **public**, **protected** oder **private** sind angezeigt.

3.) Informationen zur Klasse (Klassenzugehörigkeit) abfragen

a) mittels des **Typ-Operator** `instanceof`

Syntax:

```
(Objekt instanceof Klasse) : bool
```

`instanceof` wird dazu verwendet um festzustellen, ob ein gegebenes Objekt ein Objekt ist, das zu einer bestimmten Klasse gehört.



Beispiel:

```
<?php

class User
{
    protected $email = ' max@mustermann.de';

    public function getEmail()
    {
        return $this->email;
    }
}

$user = new User();
echo ($user instanceof User) ? 'ist Instanz von User' : 'ist keine Instanz von User';
#if-Statement mittels ternären Operator
//=> ist Instanz von User
```

(Beispiel: oop14.php)

b) mittels `get_class()`

```
get_class ([ object $object ] ) : string
```

Ermittelt den Klassennamen für das übergebene object.



Beispiel:

```
<?php

class User
{
    protected $email = ' max@mustermann.de';

    public function getEmail()
    {
        return $this->email;
    }
}

$user = new User();
echo get_class($user); //=> User
```

(Beispiel: oop15.php)

4.) weitere Informationen zum Objekt abfragen

a) mittels `get_object_vars`

```
get_object_vars ( object $object ) : array
```

Liefert die zugreifbaren nichtstatischen Eigenschaften des gegebenen Objekts object entsprechend des Gültigkeitsbereichs.



Beispiel:

```
<?php

class User
{
    public $vorname = 'Max';
    protected $nachname = 'Mustermann';
    private $email = 'max@mustermann.de';
    const Kontostand = 1;

    public function gibAttributeAus()
    {
        var_dump(get_object_vars($this)); #Rückgabe 'alle' Attribute, jedoch keine
        statischen
    }
}

$user = new User();

var_dump(get_object_vars($user)); #Rückgabe nur public Attribute

$user->gibAttributeAus();

//=> array(1) {
//=>   ["vorname"]=>
//=>   string(3) "Max"
//=> }
//=> array(3) {
//=>   ["vorname"]=>
//=>   string(3) "Max"
//=>   ["nachname"]=>
//=>   string(10) "Mustermann"
//=>   ["email"]=>
//=>   string(17) "max@mustermann.de"
//=> }
```

(Beispiel: oop16.php)

b) mittels `get_class_vars`

```
get_class_vars ( string $class_name ) : array
```

`get_class_vars` — Liefert die Vorgabeeigenschaften einer Klasse



Beispiel:

```
<?php
class User {

    public $vorname = 'vorname';
    protected $nachname = 'nachname';
    private $email = 'email@domain.de';
    const Kontostand = 5E-6;

    public function __construct($vorname, $nachname, $email) {
        $this->vorname = $vorname;
        $this->nachname = $nachname;
        $this->email = $email;
    }

    public function gibKlassenAttributeAus() {
        var_dump(get_object_vars($this)); #Rückgabe 'alle' Attribute der Klasse,
        jedoch keine statischen
    }
}

$user = new User('Oli', 'Vogt', 'oli@home.de');

var_dump(get_class_vars(get_class($user))); #Rückgabe nur public Attribute der Klasse

$user->gibKlassenAttributeAus();

//=> array(1) {
//=>   ["vorname"]=>
//=>   string(7) "vorname"
//=> }
//=> array(3) {
//=>   ["vorname"]=>
//=>   string(7) "vorname"
//=>   ["nachname"]=>
//=>   string(8) "nachname"
//=>   ["email"]=>
//=>   string(15) "email@domain.de"
//=> }
```

(Beispiel: `oop17.php`)

⇒ Liefert ein assoziatives Array von deklarierten Eigenschaften, die im aktuellen Geltungsbereich sichtbar sind, mit ihren Vorgabewerten. Die Arrayeinträge haben dabei die Form `varname => value`. Im Fehlerfall wird `FALSE` zurückgegeben.



1.8.8. Typdeklarationen (früher `type hinting` bzw. `type hints`)

Hinter dem Begriff **Typdeklaration** verbirgt sich u. a. eine weitere Möglichkeit, die Klasse eines Objekts zu überprüfen. Sie können bei der Definition einer Methode jedem Parameter den Namen einer Klasse als Typ mitgeben. (Das geht aber auch mit anderen Datentypen s. u.)

Typdeklarationen erlauben es, dass Funktionsparameter zur Aufrufzeit von einem bestimmten Typ sind. Ist der gegebene Wert von einem falschen Typ dann wird ein Fehler generiert.



Beispiel:

```
<?php
class KlasseA
{
    protected $ParA;
    // ... beliebiger Code
}

class KlasseB
{
    protected $ParB;
    // ... beliebiger Code
}

function gebKlassenNamenAus(KlasseA $klasse)
{
    echo get_class($klasse) . "\n";
}

$myClassA = new KlasseA();
$myClassB = new KlasseB();

gebKlassenNamenAus($myClassA); //=> KlasseA
gebKlassenNamenAus($myClassB); //=> PHP Fatal error:  Uncaught TypeError: Argument
1 ...
```

(Beispiel: `oop18.php`)

Typdeklarationen anderer Datentypen

Um eine **Typdeklaration** zu spezifizieren, sollte der **Typname** vor dem *Parameternamen* hinzugefügt werden. Wird der Vorgabewert des Parameters als **NULL** definiert, akzeptiert die Funktion die Übergabe von **NULL**.

Vorsicht: Bei skalaren Datentypen (**int**, **float**, **string** und **bool**) findet vor der Typprüfung eine dynamische (implizite) Typumwandlung statt was das Ergebnis u. U. verfälscht!

Gültige Typen:

Typ	Beschreibung
Klassen/Interface-Name	Der Parameter muss ein Exemplar (instanceof) der gegebenen Klasse oder des Interface sein.
self	Der Parameter muss ein Exemplar (instanceof) der selben Klasse sein für die die Methode definiert ist. Dies kann nur für Klassen- und Objektmethoden verwendet werden.
array	Der Parameter muss ein Array sein.
callable	Der Parameter muss ein gültiges Callable sein.
bool	Der Parameter muss ein Boolescher Wert (boolean) sein.
float	Der Parameter muss eine Gleitkommazahl (float) sein.
int	Der Parameter muss eine Ganzzahl (integer) sein.
string	Der Parameter muss ein String sein.
iterable	Der Parameter muss entweder ein Array oder ein instanceof Traversable sein.
object	Der Parameter muss ein Object sein.

Anmerkung/Wiederholung: In PHP gibt es vier grundlegende **Datentypen**: **Skalar**, **Array**, **Objekt** und **Ressource**. Wobei sich Skalare und Objekte jeweils noch in verschiedenen weitere Untertypen aufteilen lassen. Für **Skalare** sind dies **Boolean**, **Integer**, **Float** (**Double**) und **String**. Bei Objekten ist der Untertyp die jeweilige Objekt-Klasse.

1.9. Virtuelle Attribute

Obwohl PHP schwach typisiert ist und keine explizite Deklaration für eine verwendete Variable erforderlich ist, hat jedoch jede Variable zu jedem Zeitpunkt einen genau definierten Typ. Mittels der **Typdeklaration** (type hints) kann zwar in einem gewissen Maß typsicher programmiert werden, jedoch ist dieses Sprachfeature limitiert (implizite Typumwandlung bei skalaren Datentypen). Oftmals wünscht man sich aber ein wesentlich höheres Maß an Typsicherheit, besonders wenn es um die Attribute eines Objektes geht.

Aus diesem Grund empfiehlt es sich beim Arbeiten mit Objekten, die Attribute (Objektvariablen) nie direkt zu setzen, sondern immer mittels Setter-Methoden. Hierbei kann dann eine Typprüfung und ggf. Typumwandlung erfolgen.

Eine weitere Möglichkeit ist das Vorhalten von Attributen, die nicht wirklich definiert sind, mittels Getter- und Setter-Methoden ⇒ sogenannte **virtuelle Attribute**. Diese sind von Außen mittels der jeweiligen Getter- und Setter ansprechbar, intern jedoch werden sie aus den 'echten' Attributen und ggf. Methoden erst mit Aufruf der jeweiligen Methode erzeugt. ⇒ man gaukelt also die Existenz von Attributen vor.

```
<?php
```

```
class Buch {
    protected $titel = '';
    protected $preis = 0; // Nettopreis
    protected $ebook = false; // Bool, ob ebook (true) oder gedruckte Version (false)

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

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

    public function getEbook() {
        return $this->ebook;
    }

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

    public function setPreis(float $preis) {
        $this->preis = $preis;
    }

    public function setEbook($ebook){
        $this->ebook = $ebook;
    }
}
```

```
public function getBruttoPreis() {
    if ($this->ebook) {
        $bruttoPreis = (float) $this->getPreis() * 1.19; // Type-Casting hier
        notwendig sonst erfolgt ein Hinweis!

    } else {
        $bruttoPreis = (float) $this->getPreis() * 1.07; // Type-Casting hier
        notwendig sonst erfolgt ein Hinweis!
    }
    return round($bruttoPreis, 2) . ' €'; #auf 2 Nachkommastellen runden
}

public function setBruttoPreis($bruttoPreis) {
    if ($this->ebook) {
        $this->preis = $bruttoPreis / 1.19;
    } else {
        $this->preis = $bruttoPreis / 1.07;
    }
}

}

$perAnhalter = new Buch();
$perAnhalter->setTitle('Per Anhalter durch die Galaxy');
$perAnhalter->setPreis('13.99');
echo $perAnhalter->getBruttoPreis();

echo PHP_EOL;

$perAnhalterEbook = new Buch();
$perAnhalterEbook->setTitle('Per Anhalter durch die Galaxy');
$perAnhalterEbook->setPreis('11.99');
$perAnhalterEbook->setEbook(true);
echo $perAnhalterEbook->getBruttoPreis();

echo PHP_EOL;

$DirkGentley = new Buch();
$DirkGentley->setTitle('Dirk Gentley holistische Detektei');
$DirkGentley->setEbook(true);
$DirkGentley->setBruttoPreis('15.99');
echo $DirkGentley->getBruttoPreis();

# Ausgabe:
//=> 14.97 €
//=> 14.27 €
//=> 15.99 €
```

(Beispiel: oop19.php)

Vorsicht: Die Methode `setBruttoPreis()` ändert das Attribut `$preis` ⇒ das kann zu unerwünschten Ergebnissen führen!

1.10. Magische Methoden

In PHP gibt einige vordefinierte Methoden, die von PHP in speziellen Fällen, wenn ein spezielles Ereignis eintritt, automatisch aufgerufen werden. Jede Klasse besitzt diese automatisch, sie können aber bei der Deklaration auch mit einer neuen Funktionsweise und anderen Parametern überschrieben werden. Diese Methoden werden Magische Methoden (magic methods) genannt, da sie ohne expliziten Aufruf ausgeführt werden.

Die Funktionen `__construct()`, `__destruct()`, `__call()`, `__callStatic()`, `__get()`, `__set()`, `__isset()`, `__unset()`, `__sleep()`, `__wakeup()`, `__toString()`, `__invoke()`, `__set_state()`, `__clone()` und `__debugInfo()` sind in PHP-Klassen magisch. Man kann keine Funktionen gleichen Namens in einer seiner Klassen haben, wenn man nicht die magische Funktionalität, die sie mit sich bringen, haben will.

Hinweis: Alle magischen Methoden müssen als public deklariert werden!



1.10.1. Magische Methode `__toString()`

Die Magische Methode `__toString()` wird ausgeführt, wenn ein Objekt durch eine Funktion serialisiert werden soll. (z. B. bei `echo $objekt`, `print_r($objekt)`, ...)

Beispiel:

```
<?php

class Buch {
    protected $titel = 'Per Anhalter durch die Galaxy';
    protected $preis = 11.99;
    protected $ebook = false;

    public function __toString() {
        return $this->titel . ' ' . $this->preis . ' €' .
            (($this->ebook) ? ' (Ebook)' : ' (gedruckte Version)');
    }
}

class Zeitschrift {
    protected $titel = 'c`t';
    protected $preis = 5.20;
}

$perAnhalter = new Buch();
echo $perAnhalter;
//=> Per Anhalter durch die Galaxy 11.99 € (gedruckte Version)

$c_t = new Zeitschrift();
echo $c_t;
//=> PHP Recoverable fatal error:  Object of class Zeitschrift could not be
converted to string...
```

(Beispiel: `oop20.php`)

Die `__toString()` Methode erlaubt einer Klasse zu entscheiden, wie sie reagieren soll, wenn sie in eine Zeichenkette umgewandelt wird. Sie beeinflusst, beispielsweise, was `echo $obj`; ausgegeben wird. **Diese Methode muss eine Zeichenkette zurückgeben**, ansonsten wird ein `E_RECOVERABLE_ERROR` geworfen.



1.10.2. Magische Methode `__construct()`

Die Magische Methode `__construct()` wird ausgeführt, wenn ein neues Objekt instanziiert wird. Die Besonderheit dieser magischen Methode ist, dass man bei der Instanzierung dieser Methodener Parameter übergeben kann. Deshalb wird sie auch als **Konstruktor** bezeichnet, da in ihr alles durchgeführt (z. B. Werte von Attributen setzen) werden kann, was für die Verwendung eines Objekts notwendig ist ⇒ das Objekt wird quasi mit Hilfe dieser Methodener aus der Klasse konstruiert. Der Vorteil diese Methode zu nutzen, besteht darin, dass man sich für das Setzen der Attribute den externen Aufruf der jeweiligen Setter sparen kann!

Beispiel:

```
<?php

class Buch
{
    protected $titel = '';
    protected $preis = 0;
    protected $ebook = false;

    public function __construct($titel, $preis, $ebook = false)
    {
        $this->titel = $titel; #besser mittels Aufruf des entsprechenden Setter
        $this->preis = $preis; #besser mittels Aufruf des entsprechenden Setter
        $this->ebook = $ebook; #besser mittels Aufruf des entsprechenden Setter
    }

    public function __toString()
    {
        return $this->titel . ' ' . $this->preis . ' €' . (($this->ebook) ? '
(Ebook)' : ' (gedruckte Version)');
    }
}

//=> Per Anhalter durch die Galaxy 11.99 € (Ebook)
```

(Beispiel: oop21.php)

OPTIMIERUNG 1: Parameter als assoziatives Array an Konstruktor übergeben

Das vorangehende Beispiel lässt sich dahingehend verbessern, dass die einzelnen Parameter in Form eines assoziativen Arrays an Konstruktor übergeben wird.

Dabei ist es sinnvoll die Übergabe zu überprüfen:

- mittels Typdeklaration in der Methodendefinition
- mittels Abfrage der übergebenen Daten (ob und welche Key-Value-Paare übergeben wurden)

```
<?php

class Buch
{
    protected $titel = '';
    protected $preis = 0;
    protected $ebook = false;

    public function __construct(array $daten = [])
    //Typprüfung mittels Typdeklaration
    {
        if ($daten) {
            //Prüfen ob Parameter $daten vorhanden
            foreach ($daten as $key => $value) {
                switch (true) {
                    //Prüfen ob Key-Value-Paar übergeben und
                    //Aufruf des jeweiligen Setters
                    case ($key == 'titel'):
                        $this->setTitel($value); #jetzt mit Setter
                        break;
                    case ($key == 'preis'):
                        $this->setPreis($value); #jetzt mit Setter
                        break;
                    case ($key == 'ebook'):
                        $this->setEbook($value); #jetzt mit Setter
                        break;
                }
            }
        }
    }

    public function __toString()
    {
        return $this->titel . ' ' . $this->preis . ' €' .
            (($this->ebook) ? ' (Ebook)' : ' (gedruckte Version)');
    }

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

```
public function setPreis($preis)
{
    $this->preis = $preis;
}

public function setEbook($ebook)
{
    $this->ebook = $ebook;
}
}

$perAnhalter = new Buch([
    'titel' => 'Per Anhalter durch die Galaxy',
    'preis' => 11.99,
    'ebook' => true,
]);
echo $perAnhalter;
//=> Per Anhalter durch die Galaxy 11.99 € (Ebook)
```

(Beispiel: oop22.php)

OPTIMIERUNG 2: Setter automatisch auf Basis des Keynamen aufrufen

Voraussetzung: Funktion method_exists() und Variablenfunktionen

1.10.3. Funktion method_exists()

```
method_exists ( mixed $object, string $method_name ) : bool
```

Prüft ob eine Methode mit Namen `method_name` im Objekt `objekt` definiert ist.



[php.net](https://www.php.net/manual/en/function.method-exists.php)

1.10.4. Variablenfunktionen

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

Variablenfunktionen funktionieren **nicht** mit Sprachkonstrukten wie `echo`, `print`, `unset()`, `isset()`, `empty()`, `include` und `require`. Sie müssen Ihre eigenen Wrapperfunktionen verwenden um diese Konstrukte als Variablenfunktionen benutzen zu können.



[php.net](https://www.php.net/manual/en/functions.variable-functions.php)

Beispiel:

```
<?php

$myVar = 'Hallo Welt';

function myFunction()
{
    return 'Hola mundo';
}

echo $myVar . PHP_EOL; //=> Hallo Welt

$myVar = 'myFunction';

echo $myVar . PHP_EOL; //=> myFunction
echo $myVar() . PHP_EOL; //=> Hola mundo #hier Aufruf der Variablenfunktion
// d. h. der Wert von $myVar wird als Funktionsname verwendet
```

(Beispiel: oop23.php)

OPTIMIERUNG 2: Durchführung

```
<?php

class Buch
{
    protected $titel = '';
    protected $preis = 0;
    protected $ebook = false;

    public function __construct(array $daten = [])
    {
        if ($daten) {
            foreach ($daten as $key => $value) {
                $setterName = 'set' . ucfirst($key);
                // zusammensetzen (konkateneren) des Werts von $settername
                // nach Namenskonvention von Settern
                if (method_exists($this, $setterName)) {
                    // prüfen, ob ein passender Setter mit $settername existiert
                    $this->$setterName($value); // Setteraufruf
                }
            }
        }
    }

    public function __toString()
    {
        return $this->titel . ' ' . $this->preis . ' €' .
            (($this->ebook) ? ' (Ebook)' : ' (gedruckte Version)');
    }

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

    public function setPreis($preis) { $this->preis = $preis; }

    public function setEbook($ebook) { $this->ebook = $ebook; }
}

$perAnhalter = new Buch([
    'titel' => 'Per Anhalter durch die Galaxy',
    'preis' => 11.99,
    'ebook' => true,
]);
echo $perAnhalter;
//=> Per Anhalter durch die Galaxy 11.99 € (Ebook)
```

(Beispiel: oop24.php)

OPTIMIERUNG 3: das Datenarray als virtuelles Attribut anlegen

Vorteil: wird der Code zur Evaluierung und Ausführung des Settermethoden im Konstruktor in eine separate Settermethode für ein virtuelles Datenattribut ausgelagert, so kann jederzeit dieser Setter für die Änderungen der Attributswerte verwendet werden.

```
<?php

class Buch
{
    protected $titel = '';
    protected $preis = 0;
    protected $ebook = false;

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

    public function setDaten(array $daten = [])
    {
        if ($daten) {
            foreach ($daten as $key => $value) {
                $setterName = 'set' . ucfirst($key);
                // zusammensetzen (konkatenieren) des Werts von $settername
                // nach Namenskonvention von Settern
                if (method_exists($this, $setterName)) {
                    // prüfen, ob ein passender Setter mit $settername existiert
                    $this->$setterName($value); // Setteraufruf
                }
            }
        }
    }

    // ... Rest wie im Beispiel zuvor
}

$perAnhalter = new Buch([
    'titel' => 'Per Anhalter durch die Galaxy',
    'preis' => 11.99,
    'ebook' => true,
]);
echo $perAnhalter;
//=> Per Anhalter durch die Galaxy 11.99 € (Ebook)
```

(Beispiel: oop24b.php)

1.10.5. \$_POST oder \$_GET in __construct()

Formulardaten liegen normalerweise in \$_POST (oder \$_GET) in Form eines assoziativen Arrays vor und entsprechend können diese auch direkt dem Konstruktor übergeben werden (die optimierte Version 2 bzw. 3 wählt von alleine die passenden Setter zu den Keys aus - filtert quasi auch das übergebene)

Aufruf:

```
$perAnhalter = new Buch($_POST);
```

1.10.6. Sinnvolle Namensgebung

Wird bei der Benennung der Zusammenhang zwischen den Array-Schlüsseln, den Attributen der Klasse und den Settern berücksichtigt, so erleichtert sich vieles.

Bsp: Array-Schlüssel `titel`, Attribut `$titel` und Setter `setTitel()`

oder z. B. bei Übergabe `$_POST` an Konstruktor:

Bsp: Formularfeld Attribut `name="titel"`, Attribut `$titel` und Setter `setTitel()`.

living document \Rightarrow Fortsetzung folgt

9. Zu dieser Dokumentation / Lizenzbedingungen

Dieses Dokument ist eine begleitende Dokumentation zu den Workshops und wird fortlaufend kontextbezogen angepasst bzw. ergänzt → *Living Document*.

Es hat ausdrücklich nicht den Anspruch eines Lehrbuchs, sondern soll lediglich als Orientierungshilfe für die Durchführung und Wiederholung der Workshops sowie als Einstiegshilfe für eigene weitere Recherchen verstanden werden.

Das Dokument referenziert teilweise auf die Inhalte aus der entsprechenden Class im [Webmaster WBT](#) und/oder es wurden einige Formulierungen zwecks weiterer Ausführung direkt übernommen.

Alle weiteren Inhalte dieses Dokuments, sofern nicht selbst erstellt oder nicht anders angegeben, stammen aus Quellen mit frei zugänglichen Inhalte, die unter der Creative Commons Attribution-ShareAlike-Lizenz veröffentlicht sind, oder sind spezielle für die Erstellung von Unterrichts- bzw. Studienmaterial freigegeben.

Es sollen ausdrücklich alle diese Inhalte nicht als Eigene dargestellt und ausgegeben werden.

Sollte trotz Überprüfung eine Urheberrechtsverletzung vorliegen, bitte ich um kurze Information, damit diese umgehende beseitigt werden können.

Dieses Dokument ist ausschließlich für Schulungszwecke erstellt und teilweise urheberrechtlich geschützt bzw. unterliegt tw. urheberrechtlichen Einschränkungen. Somit sind die Weitergabe sowie die Vervielfältigung dieses Dokuments, als auch die Verwertung und Mitteilung seines Inhalts nicht erlaubt, soweit nicht ausdrücklich gestattet.

Die Nutzung dieses Dokuments ist ausschließlich Kursteilnehmern mit der Berechtigung des Zugriffs auf das [Webmaster WBT](#) gestattet.

9.1. Lizenzangaben

1: php.net

[Copyright PHP Manual](#) nach [Creative Commons Attribution 3.0 License](#)

2: Wikipedia

[Creative-Commons-Lizenz Namensnennung – Weitergabe unter gleichen Bedingungen 3.0 \(unported\)](#)
sowie aus Kompatibilitätsgründen weiterhin auch der [GFDL](#)

3: php-einfach.de

[n/a](#)