PEARSON Education

Markt+Technik

jetzt lerne ich

PHP 5 & MySQL 4.1

Der schnelle Einstieg in die objekt-orientierte Webprogrammierung

SVEN LETZEL FRIEDHELM BETZ

Markt+Technik

KAPITEL 4

jetzt lerne ich

Objektorientierte Programmierung mit PHP

In den letzten zwei Kapiteln haben Sie einiges über PHP gelernt und sich mit den Grundlagen vertraut gemacht. In diesem Kapitel wenden wir uns einer fortgeschrittenen Programmiertechnik zu: dem objektorientierten Programmieren (**O**bjekt **O**rientierte **P**rogrammierung) mit PHP. Spätestens, wenn Sie Programmcode in unterschiedlichen Projekten wieder verwenden wollen, werden Sie diese Technik zu schätzen wissen.

Sie werden Klassen und Objekte, ihre Eigenschaften und Methoden kennen lernen. Anhand von Beispielen werden Sie erfahren, wie Sie Klassen modellieren, Objekte erzeugen und diese in der Programmierung einsetzen.

Dieses Kapitel konzentriert sich auf PHP 5. Unserer Meinung nach bietet erst diese Version umfangreiche und sinnvolle Möglichkeiten, um objektorientiert zu programmieren. Die Konzepte und Möglichkeiten der Objektorientierung in PHP sind stark an Java angelehnt. Falls Sie die Programmierung mit Java beherrschen, werden Ihnen viele Dinge bekannt vorkommen.

4.1 Klassen und Objekte

Ein Objekt wird aus einer Klasse erzeugt. Um Objekte verwenden zu können, brauchen Sie eine Klasse, nach deren Vorschriften ein Objekt erzeugt werden kann. Ein Klasse ist ein Bauplan mit Vorschriften nach dessen Vorlage Sie beliebig viele Objekte erzeugen können. Sie können das mit einem gewöhnlichen Plan für ein Reihenhaus vergleichen: Der Bauplan beschreibt genau, wie ein Reihenhaus gebaut werden soll, wie viele Fenster, Türen, welche Heizung

und viele weitere Details. Der Plan für das Reihenhaus ist die Klasse, das reale Reihenhaus ist ein Objekt vom Typ Reihenhaus. Der Bauvorgang ist die Instanziierung, mit `new` wird ein Objekt »gebaut«, das heißt erzeugt und einer Variablen zugewiesen. Unter dem Namen dieser Variablen können Sie das Objekt ansprechen.

Klasse: abstrakter Bauplan für ein Objekt, wird mit dem Schlüsselwort `class` definiert: `class KlassenName`

Objekt: mit dem Schlüsselwort `new` wird von einer Klasse eine konkrete Ausprägung erzeugt, eine Instanz. Dieses Objekt weisen Sie einer Variablen zu. Diese Variable verweist als Referenz auf das Objekt: `$objekt = new KlassenName()`. Ein Objekt wird auch als Instanz einer Klasse bezeichnet.

4.1.1 Eigenschaften und Methoden

In einer Klasse definieren Sie die Eigenschaften und Methoden eines möglichen Objekts. Eigenschaften sind dabei ganz wörtlich zu verstehen. Um bei unserem Beispiel mit dem Reihenhaus zu bleiben, könnten die Eigenschaften vielleicht die Anzahl der Stockwerke, die Art der Heizung oder die Farbe des Hausanstrichs sein.

Die Methode einer Klasse oder eine Klassenmethode ist nichts anderes als eine gewöhnliche Funktion, die innerhalb einer Klasse definiert wird.

Stellen Sie sich die Instanz einer Klasse als Blackbox vor, die über Methoden mit der Außenwelt kommuniziert. Für den Umgang mit einem Objekt benutzen Sie die gegebenen Methoden, ohne sich jedoch darüber Gedanken machen zu müssen, wie Ihre Blackbox im Inneren funktioniert. Halten Sie sich das Bild einer Waschmaschine vor Augen. Eine Waschmaschine benutzen Sie um Ihre Wäsche zu waschen und es stehen Ihnen verschiedene Einstellmöglichkeiten zur Verfügung. Ohne im Detail wissen zu müssen, wie genau im Innern eine Waschmaschine funktioniert, befüllen Sie die Maschine, wählen ein Waschprogramm, geben Waschmittel hinzu und stellen die Temperatur ein, schalten die Maschine an und können nach einer gewissen Zeit Ihre saubere Wäsche nach dem Abschalten der Maschine entnehmen. Unabhängig von der Herstellerfirma Ihrer Waschmaschine wird dieser Vorgang immer ähnlich sein. Aus Sicht der Programmierung können wir folgende Eigenschaften und Methoden der Klasse *Waschmaschine* identifizieren:

1. Temperatur: zwischen 30° und 90°
2. Waschprogramm: Buntwäsche, Feinwäsche, Kochwäsche
3. Enthält Wäsche: ja, nein

Eigenschaften der Klasse Waschmaschine und ihre möglichen Werte

1. Auswahl der Temperatur
2. Auswahl des Waschprogramms
3. Einfüllen des Waschmittels
4. Einschalten der Waschmaschine
5. Waschen der Wäsche
6. Abstellen der Waschmaschine

Methoden der Klasse Waschmaschine

Bei der Modellierung einer Klasse handelt es sich immer um eine Abstraktion der Realität nach Ihren eigenen Vorstellungen. Es existieren keine Vorschriften über richtig oder falsch. Wie Sie persönlich eine Klasse modellieren, hängt vom Einsatzzweck und Ihrer persönlichen Sichtweise ab. Beachten Sie bitte, dass die Eigenschaften einer Klasse innerhalb der sie definierenden Klasse global verfügbar sind.

Beispiel für die Klasse *Waschmaschine* in PHP:

waschmaschine.php

```
<?php
class Waschmaschine
{
  /* Definieren der Eigenschaften */
  public $temperatur;
  public $waschprogramm;
  public $waesche;
  /* Methoden */
  public function auswahlTemperatur($temp)
  {
  }

  public function auswahlWaschprogramm($programm)
  {
  }

  public function einfuellenWaschmittel($wmittel)
  {
  }
```

```
  public function einschaltenWaschmaschine()
  {
  }

  public function abschaltenWaschmaschine()
  {
  }

  public function waschen()
  {
  }

  protected function pruefenWaschmaschine()
  {
  }
}
/* Erzeugen einer Instanz mit new */
$miele = new Waschmaschine();
?>
```

Dieses Beispiel hat keinen praktischen Nutzen. Es soll Ihnen lediglich veranschaulichen, wie Sie Eigenschaften und Methoden einer Klasse identifizieren und in PHP modellieren können.

Standardwerte für Eigenschaften

Bei der Definition von Eigenschaften einer Klasse können Sie diesen Standardwerte zuweisen. Allerdings können Sie nur konstante Werte zuweisen, das heißt kein Ergebnis eines Funktionsaufrufs oder eine andere Variable. Außerdem dürfen Sie keine Operatoren verwenden. Wollen Sie die Eigenschaften mit nicht konstanten Werten initialisieren, steht Ihnen diese Funktionalität mit einem Konstruktor zur Verfügung. Konstruktoren werden Sie im nächsten Abschnitt kennen lernen.

Eigenschaften.php

Beispiel zur Initialisierung von Eigenschaften mit Standardwerten:

```
<?php
class MeineKlasse
{
  /* Falsche Art der Initialisierung, erzeugt einen Fehler */
  public $datum = date('d-m-Y');
  /* Nur konstante Werte sind erlaubt */
  public $datum = '01-10-2004';
}
?>
```

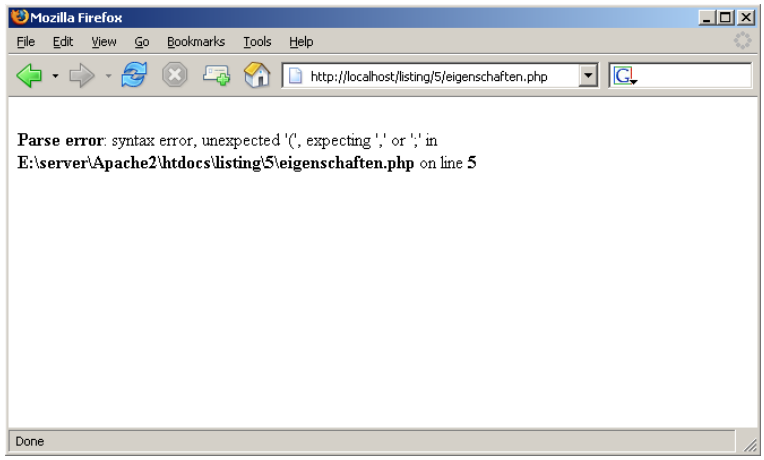

Abb. 4.1: Fehlermeldung bei der Zuweisung von Standardwerten

4.1.2 Konstruktoren und Destruktoren

In einer Klasse können Sie eine spezielle Methode definieren, die automatisch jedes Mal beim Erzeugen eines neuen Objekts ausgeführt wird: einen Konstruktor. Definiert wird ein Konstruktor durch die Angabe von `function __construct([mixed args [, ...]])`. Dieser Methode können Sie wie jeder anderen Funktion beliebige Parameter übergeben. Sehr praktisch ist dies, wenn Sie beim Erzeugen eines Objekts dessen Eigenschaften mit Werten initialisieren wollen oder sonstige Funktionalitäten implementieren wollen, die automatisch bei der Instanziierung ablaufen sollen.

Wir initialisieren die Eigenschaft Datum im Konstruktor mit Hilfe von `date()` und weisen damit `$datum` das tagesaktuelle Datum zu. Zur Kontrolle geben wir das aktuelle Datum sofort im Konstruktor wieder aus.

__construct.php

```
<?php
class MeineKlasse
{
  private $datum;
  public function __construct()
  {
    $this->datum = date('d-m-Y');
    echo 'Heute ist der ' . $this->datum . '<br>';
  }
}
$objekt = new MeineKlasse();
?>
```

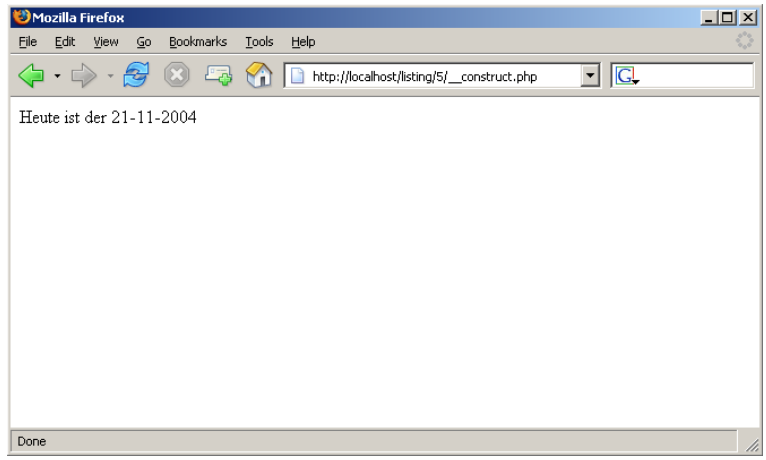

Abb. 4.2: Das tagesaktuelle Datum wird im Konstruktor der Eigenschaft `$datum` zugewiesen und angezeigt

Um Eigenschaften mit Werten zu initialisieren, müssten Sie ohne Konstruktor eine oder mehrere Methoden definieren, die den Eigenschaften Ihrer Klasse Werte zuweisen. Ohne Methode sollte es für ein Objekt keine Möglichkeit geben, auf eine Eigenschaft zuzugreifen.

Objekte werden nicht nur erzeugt, sondern auch zerstört, spätestens nach Beendigung eines Skripts. Möchten Sie in einer Klasse bestimmte Dinge automatisch erledigen lassen, sobald ein Objekt dieser Klasse zerstört wird, haben Sie die Möglichkeit, einen Destruktor in Ihrer Klasse zu definieren: `function __destruct()`. Ein Destruktor wird automatisch bei der Zerstörung eines Objekts aufgerufen, nimmt keine Parameter entgegen und kann als Gegenstück zum Konstruktor verstanden werden.

Ein Destruktor kann für bestimmte Aufräumarbeiten sinnvoll sein. Sie können nicht mehr benötigte temporäre Dateien, die ein Objekt angelegt hat, löschen oder für Informationszwecke die Laufzeit des Skripts in einer Datei oder Datenbank festhalten. Oder Sie könnten festhalten, zu welcher Uhrzeit ein Besucher Ihres Webshops sich ausloggt. Ihrer Fantasie sind dabei fast keine Grenzen gesetzt.

4.1.3 Konstanten in Klassen

In einer Klasse können Sie Konstanten nicht mit `define()` definieren. Stattdessen müssen Sie klasseneigene Konstanten mit dem Schlüsselwort **`const`** `NameDerKonstante = WertDerKonstanten` festlegen. Diese Konstanten gehören der Klasse selbst und können deshalb von einer Instanz, einem Objekt, nicht

angesprochen werden. Innerhalb einer Klasse können Sie Konstanten mit **`self::NameDerKonstante`** ansprechen. Allerdings können Sie auf klasseneigene Konstanten in einem Skript ohne Instanz zugreifen. Ansonsten unterscheiden sich klasseneigene Konstanten nicht von normalen Konstanten, die mit `define()` festgelegt wurden.

const.php

```
<?php
class MeineKlasse
{
  const PI = 3.14;
  public function __construct()
  {
    echo 'Ausgabe aus Konstruktor:' . self::PI . '<br>';
  }
}
echo 'Globaler Kontext: ' . MeineKlasse::PI . '<br>';
$objekt = new MeineKlasse();
?>
```

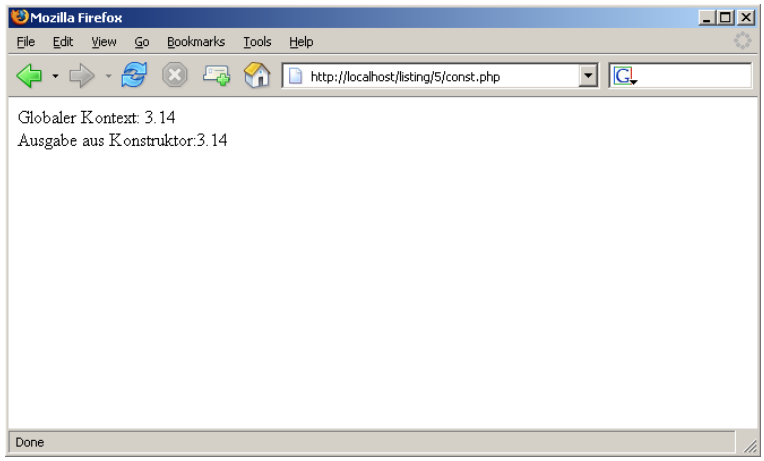

Abb. 4.3: Zugriff auf klasseneigene Konstanten

Klasseneigene Konstanten sind ebenso wie normale Konstanten global verfügbar. Allerdings können Sie klasseneigene Konstanten nicht einfach mit ihrem Namen ansprechen, Sie müssen im globalen Kontext immer die Schreibweise `NameDerKlasse::NameDerKonstante` verwenden.

4.1.4 Vererbung mit extends

Eine komfortable Möglichkeit die Funktionalität einer Klasse zu erweitern ist die Ableitung einer Klasse von einer vorhandenen Klasse. Im Jargon der OOP nennt man diesen Vorgang Vererbung. Die abgeleitete Klasse erbt die vorhandenen Eigenschaften, Methoden und Konstanten von der Klasse, von der Sie ableiten, der Elternklasse. Vererbung erlaubt Ihnen zusätzliche Eigenschaften und Methoden in der abgeleiteten Klasse zu definieren. Die Ableitung einer Klasse von einer Elternklasse notieren Sie mit dem Schlüsselwort `extends` bei der Definition einer Klasse: `class NeueKlasse` **`extends`** `ElternKlasse`. Die abgeleitete Klasse `NeueKlasse` wird auch als Kindklasse bezeichnet. Sie bemerken, dass die Begrifflichkeiten bei der Vererbung an der menschlichen Vererbung angelehnt sind.

Anhand der Klasse *Fahrzeug* und der abgeleiteten Klasse *Oldtimer* können wir uns die Vererbung verdeutlichen:

fahrzeug.php

```
<?php
class Fahrzeug
{
  public $marke;
  public $farbe;
  public function __construct($typ, $farbe)
  {
   $this->marke = $typ;
   $this->farbe = $farbe;
  }
}
class Oldtimer extends Fahrzeug {
  public $baujahr;
  public function __construct ($typ, $farbe, $jahr)
  {
    $this->baujahr = $jahr;
    parent::__construct($typ, $farbe);
    echo 'Merkmale von ' . get_class($this) . '<br>';
    echo 'Automarke: ' . $this->marke . '<br>';
    echo 'Farbe: ' . $this->farbe . '<br>';
    echo 'Baujahr: ' . $this->baujahr . '<br>';
  }
}
$auto1 = new Fahrzeug('vw','rot');
$auto2 = new Oldtimer('Mercedes', 'silber', 1924);
?>
```

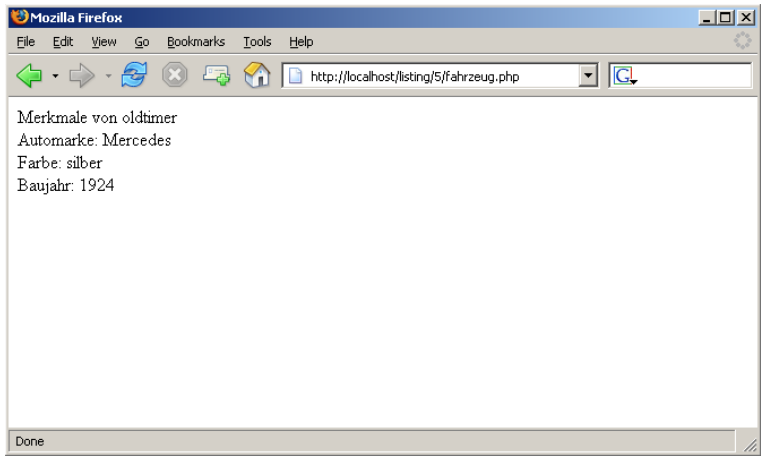

Abb. 4.4: Vererbung und die Ausgabe von fahrzeug.php

Vererbung eröffnet Ihnen die Möglichkeit, eine generische Elternklasse mit grundlegenden Funktionalitäten zu modellieren. Für spezielle Anwendungsfälle können Sie von dieser generischen Klasse eine spezifischere Klasse ableiten, um zusätzliche, speziell benötigte Funktionalitäten zu implementieren. Den Code, den Ihre generische Klasse implementiert, können Sie durch Vererbung weiterverwenden, ohne immer wieder die genau gleichen, grundlegenden Funktionalitäten zu programmieren. Wir empfehlen Ihnen, das Prinzip vom Allgemeinen zum Speziellen bei Ihren eigenen Projekten zu verinnerlichen. Damit stellen Sie sicher, dass Sie einmal geschriebenen Code für speziellere Anwendungsfälle immer wieder verwenden können.

Ein Beispiel für einen konkreten Anwendungsfall dieser Vorgehensweise ist eine generische Klasse, die eine XML-Datei parst. Der grundlegende Ablauf dieses Vorgangs wird unabhängig von Ihren Daten immer gleich sein. Anstatt immer wieder den gleichen Code zu schreiben, entwerfen Sie eine allgemeine Klasse mit Grundfunktionalitäten zum Parsen von XML-Daten. Wollen Sie einen Newsfeed im RSS-Format[1] parsen, können Sie eine neue Klasse von Ihrer generischen Klasse ableiten und die speziell benötigten Funktionalitäten für das Parsen von RSS implementieren. Wenn Sie XML-Daten in einem beliebig anderen Format parsen wollen, benutzen Sie einfach Ihre generische Parserklasse und leiten eine neue Klasse mit den Funktionalitäten, die das neue Format benötigt, ab.

1. RSS, *Rich Site Summary* oder auch *Really Simple Syndication*, ist ein XML-Standard, über den Websites untereinander Informationen austauschen können – und zwar in Echtzeit. So könnten Sie z.B. über den RSS-Feed eines Newstickers dessen Inhalt auf Ihrer eigenen Site wiedergeben – jede Aktualisierung des Tickers würde auch auf Ihren Seiten angezeigt.

PHP unterstützt im Gegensatz zu anderen objektorientierten Sprachen wie C++ nur einfache Vererbung, keine mehrfache Vererbung. Das bedeutet, dass eine Klasse nur jeweils von einer Klasse erben kann. Eine abgeleitete Klasse kann daher nur genau eine einzige Elternklasse besitzen. Beachten Sie, dass es bei einfacher Vererbung möglich ist, beliebig viele Klassen von einer Elternklasse abzuleiten. Bei mehrfacher Vererbung wäre es möglich, dass eine Klasse von mehreren Klassen gleichzeitig erbt und damit mehrere Elternklassen besitzen könnte. Das folgende Schaubild kann Ihnen helfen, den Unterschied zwischen einfacher und mehrfacher Vererbung besser zu verstehen.

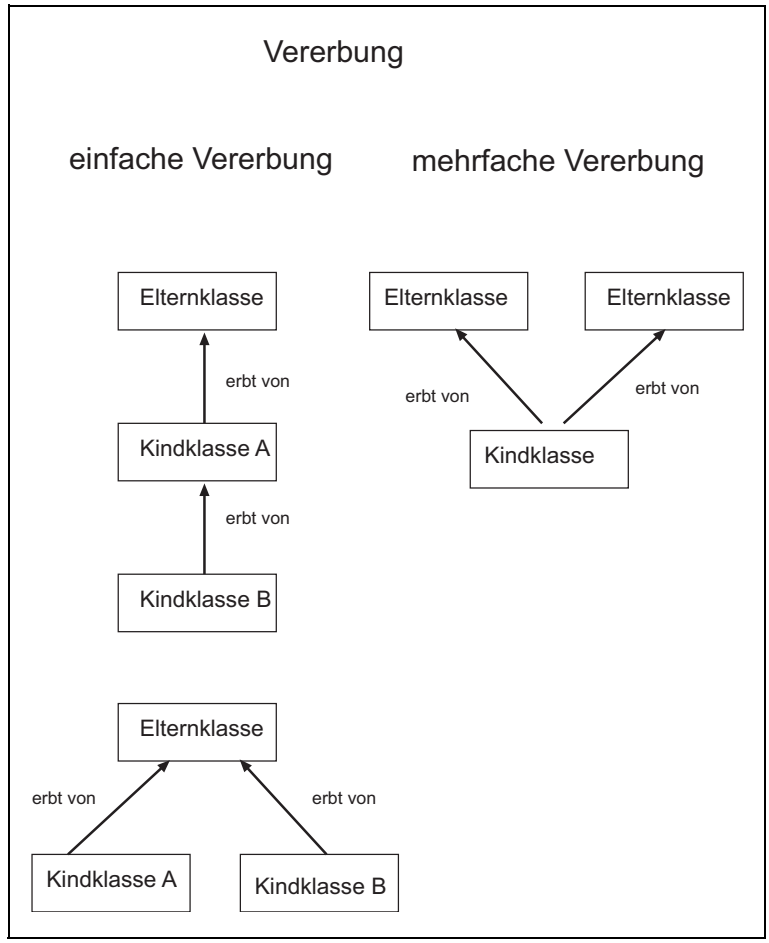

Abb. 4.5: Unterschied zwischen einfacher und mehrfacher Vererbung

Überschreiben von Methoden

Sie können in einer abgeleiteten Klasse eine Methode der Elternklasse überschreiben. Dazu definieren Sie eine Methode mit dem gleichen Namen wie in der Elternklasse. Wenn Sie auf eine solche Methode zugreifen, wird nicht die Methode der Elternklasse aufgerufen, sondern die von Ihnen überschriebene Methode. Die Originalmethode der Elternklasse können Sie mit `parent::MethodeDerElternklasse()` explizit aufrufen.

4.1.5 Sichtbarkeit und Zugriffsbeschränkung mit PHP

Die Sichtbarkeit und Zugriffsmöglichkeiten auf Methoden und Eigenschaften können Sie mit den Schlüsselwörtern `public`, `private` und `protected` bestimmen. Damit ist es möglich, das Konzept der Kapselung von Informationen bei der objektorientierten Programmierung in PHP umzusetzen: Sie kontrollieren, welche Informationen in welchem Kontext sichtbar sein sollen. Damit können Sie unkontrollierten Zugriff auf Eigenschaften und Methoden verhindern.

Sichtbarkeit von Eigenschaften

Auf Eigenschaften, die als `public` gekennzeichnet sind, ist der Zugriff in keiner Weise beschränkt. Diese sind außerhalb einer Klasse sichtbar und können von einer Instanz gelesen und verändert werden. Eigenschaften, die Sie als `protected` festlegen, sind nur innerhalb der definierenden Klasse selbst und in abgeleiteten Klassen sichtbar. Solche Eigenschaften werden zwar vererbt, aber der Zugriff ist außerhalb von Klassen nicht möglich. Das bedeutet, dass ein Objekt diese Eigenschaften nicht lesen oder verändern kann. Ein Objekt hat keinerlei Information über diese Eigenschaften. Die restriktivste Einschränkung legen Sie mit `private` fest. Diese Eigenschaften sind nur innerhalb der definierenden Klasse selbst sichtbar und werden nicht vererbt.

Sie *müssen* Eigenschaften mit `public`, `private` oder `protected` definieren, sonst liefert Ihnen PHP einen Fehler!

Sichtbarkeit von Methoden

Den Zugriff auf Methoden können Sie ebenfalls durch die Verwendung von `public`, `private` oder `protected` steuern. Das Verhalten entspricht dem der entsprechend definierten Eigenschaften. Auf Methoden, die Sie als `public` deklarieren, haben Sie uneingeschränkten Zugriff, `protected` erlaubt den Zugriff nur in der definierenden Klasse oder in einer abgeleiteten Klasse. Auf private Methoden hat nur die definierende Klasse selbst Zugriff, diese Methoden werden nicht vererbt.

Tabelle 4.1: Sichtbarkeit und Zugriffsbeschränkung für Eigenschaften und Methoden

Sichtbarkeit/ Zugriffsbeschränkung	Beschreibung
`public` (öffentlich)	Zugriff ist nicht eingeschränkt, wird vererbt, kann überschrieben werden
`protected` (geschützt)	Zugriff in der Klasse selbst und in abgeleiteten Klassen. Objekte haben keinen Zugriff, kann überschrieben werden.
`private` (privat)	Zugriff nur in der Klasse selbst, keine Vererbung. Objekte haben keinen Zugriff.

Den Zugriff auf Methoden können Sie mit `protected` oder `private` einschränken, müssen es aber nicht. Schränken Sie den Zugriff auf Methoden nicht ein, werden diese implizit als `public` definiert. Obwohl nicht zwingend erforderlich, ist es guter Stil, auch Methoden mit Zugriffsbeschränkungen zu definieren. Ihr Code wird für Sie und andere Leute dadurch leichter verständlich.

4.1.6 Zugriff auf Eigenschaften und Methoden

Innerhalb einer Klasse

Für den Zugriff auf Eigenschaften und Methoden ist die Pseudovariable `$this` zuständig. Da beim Entwurf einer Klasse nicht feststeht, unter welchem Namen Sie eine Instanz später in Ihrem Skript ansprechen wollen, brauchen Sie innerhalb einer Klassendefinition einen Mechanismus, der das für Sie erledigt. `$this` repräsentiert das aktuelle Objekt innerhalb einer Klasse, kann also als Platzhalter für den unbekannten Namen des Objekts verstanden werden.

Methoden rufen Sie mit `$this->NameDerMethode($parameter1, $parameter2, ...)` auf, während Sie Variablen mit `$this-> variablenname` ansprechen. Sie haben auch die Möglichkeit, Methoden mit `self::NameDerMethode()` oder `NameDerKlasse::NameDerMethode` aufzurufen. Für Eigenschaften gibt es diese Möglichkeit allerdings nicht.

Innerhalb einer abgeleiteten Klasse

Auf vererbte Methoden können Sie mit `$this->methode()` zugreifen oder mit `parent::NameDerMethode()` die Methode der Elternklasse explizit aufrufen. Nützlich ist das, wenn Sie in einer abgeleiteten Klasse eine Methode überschreiben, aber trotzdem die Methode der Elternklasse aufrufen wollen.

An einem Beispiel lässt sich das veranschaulichen. Wir definieren einen Konstruktor in der abgeleiteten Klasse und überschreiben damit den Konstruktor der Elternklasse, rufen diesen aber explizit mit `parent::` auf.

parent.php

```
<?php
class ElternKlasse
{
  public function __construct()
  {
    echo get_class($this) . '<br>';
    echo 'Ich bin der Konstruktor der Elternklasse' . '<br>';
  }
}
class KindKlasse extends ElternKlasse
{
  public function __construct()
  {
    echo get_class($this) . '<br>';
    echo 'Konstruktor der abgeleiteten Klasse aufgerufen' . '<br>';
    echo 'Konstruktor der Elternklasse wurde aufgerufen:' . '<br>';
    parent::__construct();
  }
}
$objekt = new KindKlasse();
?>
```

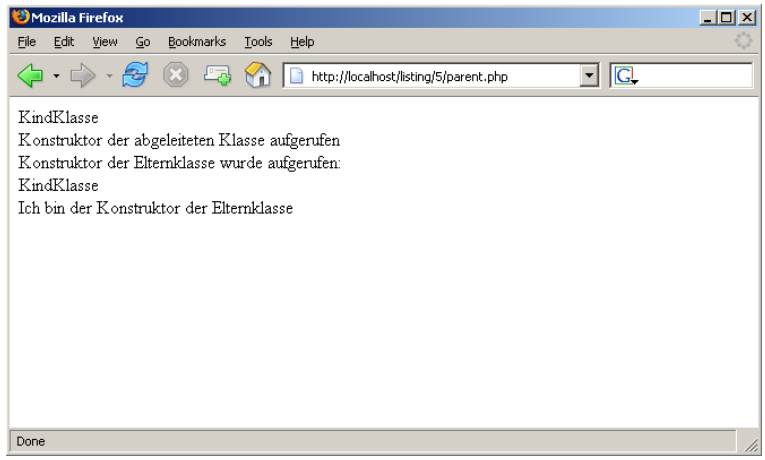

Abb. 4.6: Ausgabe von parent.php

Verfügt eine abgeleitete Klasse über einen eigenen Konstruktor, wird beim Erzeugen einer Instanz der Konstruktor der Elternklasse *nicht* automatisch aufgerufen.

Verfügt eine abgeleitete Klasse über keinen eigenen Konstruktor, wird beim Erzeugen einer Instanz automatisch der Konstruktor der Elternklasse aufgerufen.

4.1.7 Final, static und abstract

Für Eigenschaften, Methoden und Klassen gibt es weitere Definitionsmöglichkeiten mit `final`, `static` und `abstract`.

Überschreiben von Methoden mit final verhindern

Sie können neben Zugriffsbeschränkungen Methoden zusätzlich als `final` deklarieren. Im ganz wörtlichen Sinne sind solche Methoden als final, endgültig festgelegt. Diese Methoden werden zwar vererbt, lassen sich aber nicht überschreiben. Das Schlüsselwort `final` müssen Sie bei der Definition einer Methode als erste Option notieren. Sie können diese Eigenschaft einzelnen Methoden zuweisen, wenn Sie sicherstellen wollen, dass bestimmte Methoden in einer abgeleiteten Klasse nicht überschrieben werden. Damit ist garantiert, dass immer die Methode der Elternklasse zur Anwendung kommt.

Innerhalb von `MeineKlasse` definieren wir die Methode `MeineMethode` als `final`. Den Versuch, diese Methode in einer abgeleiteten Klasse zu überschreiben, wird PHP mit der Meldung eines fatalen Fehlers vereiteln.

FinaleMethode.php

```
<?php
class MeineKlasse
{
  final public function MeineMethode()
  {
  }
}
class AbgeleiteteKlasse extends MeineKlasse
{
  public function MeineMethode()
  {
  }
}
?>
```

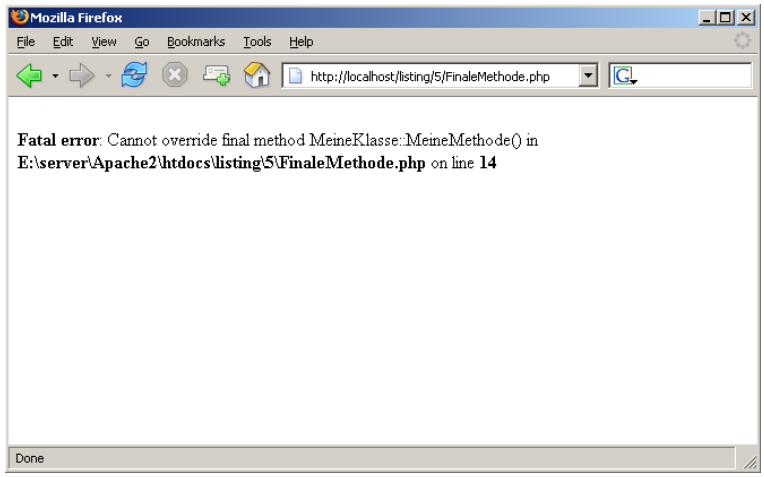

Abb. 4.7: Fehlermeldung beim Versuch eine finale Methode zu überschreiben

Ableitung von Klassen mit final verhindern

Ebenso wie Methoden können auch Klassen als final gekennzeichnet werden. Ableitungen von finalen Klassen sind nicht möglich und Vererbung wird dadurch verhindert. Definieren Sie eine Klasse als `final`, sind damit auch alle Methoden implizit als final definiert. Da keine andere Klasse von einer finalen Klasse erben kann, können auch die Methoden nie überschrieben werden.

Wir definieren die komplette Klasse `MeineKlasse` als `final`. Beim Versuch von dieser Klasse abzuleiten, meldet uns PHP einen Fehler.

FinaleKlasse.php

```
<?php
final class MeineKlasse
{
}
class AbgeleiteteKlasse extends MeineKlasse
{
}
?>
```

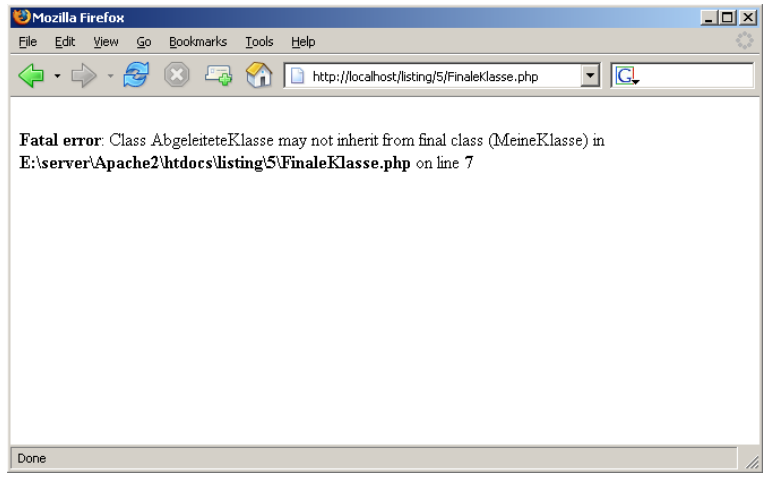

Abb. 4.8: Fehlermeldung beim Versuch von einer finalen Klasse abzuleiten

Statische Eigenschaften und Methoden

Mit dem Schlüsselwort `static` können Sie Eigenschaften und Methoden als statisch definieren. Auf statische Eigenschaften und Methoden einer Klasse können Sie ohne Instanz zugreifen. In Abhängigkeit der Zugriffsbeschränkung werden diese Klassenelemente vererbt. Das Schlüsselwort `static` notieren Sie nach den Zugriffsbeschränkungen. Der Zugriff auf statische Eigenschaften erfolgt innerhalb einer Klasse mit `self::$StatischeEigenschaft` oder mit `parent::$StatischeEigenschaft` innerhalb einer abgeleiteten Klasse. In allen anderen Fällen, entweder im globalen Kontext oder innerhalb unabhängiger Klassen, haben Sie mit `NameDerKlasse:: $StatischeEigenschaft` Zugriff auf eine statische Eigenschaft. Genauso greifen Sie auf statische Methoden zu. Sie müssen beim Zugriff nur die Parameterliste oder ein leeres Klammerpaar anhängen. Statische Eigenschaften und Methoden können nicht überschrieben werden, sie gehören zu der sie definierenden Klasse und existieren damit genau einmal.

Innerhalb einer statischen Methode können Sie die Pseudovariable `$this` nicht benutzen. Eine Instanz hat keine Möglichkeit, auf statische Eigenschaften und Methoden mit `->` zuzugreifen

Abstrakte Klassen und Methoden

Mit dem Schlüsselwort `abstract` definieren Sie abstrakte Methoden und Klassen, `abstract` notieren Sie dabei als erste Option bei der Definition. Wenn Sie innerhalb einer Klasse eine abstrakte Methode implementieren, muss die Klasse insgesamt abstrakt sein. Definieren Sie eine abstrakte Klasse, kann diese Klasse abstrakte und nicht-abstrakte Methoden enthalten. Von abstrakten

Klassen können Sie nur ableiten, die Erzeugung einer Instanz ist nicht möglich. Indem Sie eine Klasse als abstrakt definieren, erzwingen Sie, dass diese Klasse nicht direkt, sondern nur implizit durch eine abgeleitete Klasse genutzt werden kann. Damit haben Sie die Möglichkeit, Vorlagen für Ableitungen zu erstellen. Das ist dann interessant, wenn Sie den Grundsatz vom Allgemeinen zum Speziellen bei der Modellierung Ihrer Anwendung beherzigen. Ihre Ausgangsklassen können dabei so allgemein gehalten sein, dass eine Instanz einer solchen Klasse keinen Sinn ergeben würde.

Aus dem vorherigen Abschnitt ergibt sich, dass abstrakte Methoden nur in abstrakten Klassen vorkommen dürfen. Bei abstrakten Methoden definieren Sie nur den Namen und die Parameterliste und erzwingen dadurch, dass eine ableitende Klasse diese Methode und ihre Funktionalität implementiert. Sinn machen abstrakte Methoden, wenn Sie zwar wissen, dass Sie eine solche Methode benötigen, aber die konkrete Funktionalität der Methode von der ableitenden Klasse abhängt.

Stellen Sie sich vor, Sie modellieren eine abstrakte Klasse `FortbewegungsMittel` mit der abstrakten Methode `fortbewegen()`. Die tatsächliche Implementierung von `fortbewegen()` möchten Sie aber für die abgeleiteten Klassen `Kfz` und `Fahrrad` bestimmt unterschiedlich implementieren.

abstract.php

```
<?php
abstract class FortbewegungsMittel
{
  abstract public function fortbewegen();
}

class Kfz extends FortbewegungsMittel
{
  public function fortbewegen()
  {
    Kfz::fahren();
  }
  private static function fahren()
  {
    echo 'Einsteigen, Motor anlassen, Gang rein, gasgeben.<br>';
  }
}

class Fahrrad extends FortbewegungsMittel
{
  public function fortbewegen()
  {
   Fahrrad::fahren();
  }
```

```
  protected static function fahren()
  {
      echo 'Aufs Rad schwingen, Pedale treten. <br>';
  }
}
$vw = new Kfz();
$vw->fortbewegen();
$fischer = new Fahrrad();
$fischer->fortbewegen();
?>
```

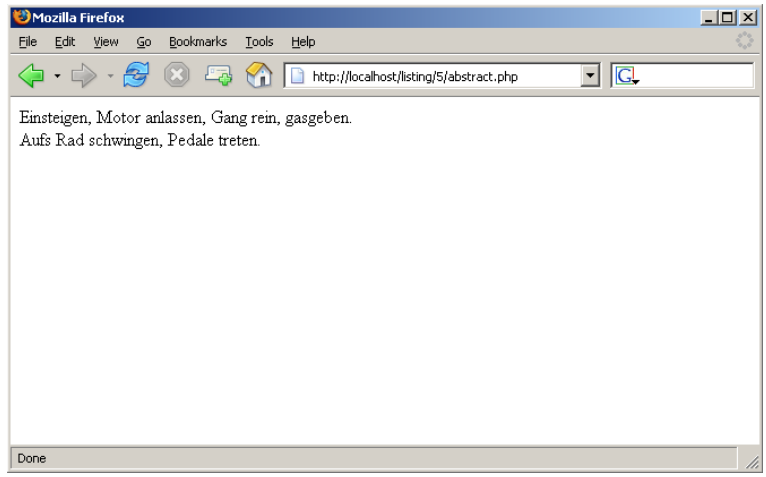

Abb. 4.9: Fortbewegung mit einem VW funktioniert anders als mit dem Rad

4.2 Weiterführende Möglichkeiten

4.2.1 Magische Funktionen

PHP bietet Ihnen magische Funktionen, die Sie daran erkennen, dass sie mit zwei Unterstrichen beginnen. Zwei dieser magischen Funktionen haben Sie mit Konstruktoren und Destruktoren im Abschnitt 4.1.2 kennen gelernt. In den folgenden Abschnitten stellen wir Ihnen weitere magische Funktionen vor.

PHP reserviert intern alle Funktionsnamen, die mit __ beginnen, für magische Funktionen. Deshalb sollten Sie bei der Namensgebung für eigene Funktionen darauf verzichten, diese mit zwei Unterstrichen zu beginnen.

Klassen automatisch laden: __autoload

Stellen Sie sich vor, Sie haben eine ganze Reihe von eigenen Klassen programmiert, die Sie innerhalb eines Skripts verwenden wollen. Alle Klassen liegen in separaten Dateien vor und bevor Sie auf diese zugreifen können, müssen diese Ihrem Skript bekannt sein. Erreichen können Sie das dadurch, dass Sie am Anfang Ihres Skripts alle benötigten Dateien mit `include_once()` oder `require_once()` einbinden. Allerdings sind Sie sich nicht sicher, ob Sie wirklich alle Klassen brauchen. Optimal wäre daher ein Mechanismus, der eine Klasse genau dann dynamisch einbindet, wenn Sie auf diese Klasse zugreifen. Diese Funktionalität bietet Ihnen die magische Funktion `__autoload('klassenname')`die Sie in Ihrem Skript definieren können. Als einzigen Parameter erwartet diese Funktion den Namen einer Klasse. An einem Beispiel können wir uns diesen Mechanismus verdeutlichen.

Wir haben eine Klasse *MeineKlasse*, die Sie in der Datei `MeineKlasse.php` abgelegt haben.

Inhalt von MeineKlasse.php

```
<?php
class MeineKlasse
{
  public function __construct()
  {
    echo 'Ich wurde mit __autoload dynamisch zur Laufzeit geladen
<br>';
  }
}
?>
```

In einem anderen Skript wollen Sie diese Klasse verwenden, ohne diese vor der Benutzung einzubinden.

autoload.php

```
<?php
function __autoload($klassenname)
{
  include_once($klassenname . '.php');
}
$objekt = new MeineKlasse();
?>
```

Sobald das Skript `autoload.php` auf die Anweisung `new MeineKlasse();` trifft, stellt PHP fest, dass die benötigte Klasse nicht zur Verfügung steht. Daraufhin sucht PHP in Ihrem Skript nach der magischen Funktion `__autoload()`, führt diese aus und die benötigte Klasse wird geladen.

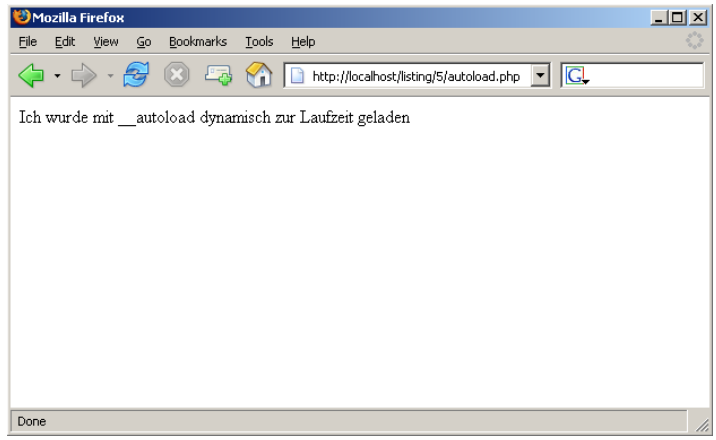

Abb. 4.10: Ausgabe von autoload.php

__set und __get

```
__set($NameDerEigenschaft, $wert) ohne Rückgabewert
__get($NameDerEigenschaft) Rückgabewert: Wert der Eigenschaft
```

Der Zugriff auf nicht vorhandene Eigenschaften führt normalerweise zu einer Fehlermeldung. Mit `__set` und `__get` können Sie diese Zugriffe abfangen und damit zur Laufzeit eigentlich nicht vorhandenen Eigenschaften einen Werte zuweisen oder abfragen. Weisen Sie einer nicht vorhandenen Eigenschaft einen Wert zu, wird automatisch `__set()` aufgerufen, fragen Sie eine nicht vorhandene Eigenschaft ab, wird automatisch `__get()` aufgerufen. Diese Funktionalität ermöglichen es Ihnen, praktisch virtuelle Eigenschaften zu definieren.

set_get.php

```
<?php
class personen
{
  protected $person = array(
    'Franz' => 'Ludwigshafen',
    'Kai'   => 'Mannheim',
    'Katja' => 'Köln'
  );
  function __get($bezeichner)
  {
    return $this->person[$bezeichner];
  }
```

```
  function __set($bezeichner, $wert)
  {
    $this->person [$bezeichner] = $wert;
    return;
  }
}

$objekt = new personen();
$objekt->Herbert = 'Karlruhe';
echo $objekt->Herbert;
?>
```

Ausgabe:

Karlsruhe

Mit der Anweisung `$objekt->Herbert = 'Karlruhe';` wird automatisch `__set()` aufgerufen und im Array `$person` der Schlüssel *Herbert* mit dem Wert *Karlsruhe* belegt. Die Anweisung `echo $objekt->Herbert` ruft automatisch `__get()` auf und liefert folgerichtig *Karlsruhe*.

Zerbrechen Sie sich nicht den Kopf über diese Art von Magie. Nur in seltenen speziellen Fällen werden Sie von dieser Möglichkeit Gebrauch machen wollen.

__call

`__call($funktionsname, $parameter) Rückgabewert: beliebige Werte`

Was `__set()` und `__get()` für nicht vorhandene Eigenschaften sind, ist `__call()` für nicht vorhandene Methoden. Beim Aufruf nicht vorhandener Methoden wird automatisch `__call()` aufgerufen, falls Sie diese Funktion implementiert haben. Die Funktion erwartet zwei Parameter, der erste wird als Methodenname interpretiert, der zweite als numerisches Array von Parametern für die Methode.

__call.php

Wir implementieren die virtuelle Methode `GetWohnort` mit Hilfe von `__call()`, die uns den Wohnort einer Person liefert.

```
<?php
class personen
{
  protected $person = array(
    'Franz' => 'Ludwigshafen',
    'Kai'   => 'Mannheim',
  );
```

```
  public function __call($methodenname, $parameter)
  {
    switch ($methodenname) {
       case 'GetWohnort':
         if ( isset($parameter[0])
              && isset($this->person[$parameter[0]]) ) {
           return $this->person[$parameter[0]];
         } else {
           return FALSE;
                      }
        break;
    }
    return FALSE;
  }
}

$objekt = new personen();
echo $objekt->GetWohnort('Franz') . '<br>';
echo $objekt->GetWohnort('Kai');
?>
```

Ausgabe:

Ludwigshafen
Mannheim

Beim Aufruf von `$objekt->GetWohnort('Franz');` wird automatisch `__call()` aufgerufen und der Wohnort von Franz ausgegeben. Dabei wird `GetWohnort` als Methode und Franz als der Wert von `parameter[0]` interpretiert.

__toString

`__toString() Rückgabewert: string`

Geben Sie ein Objekt mit `echo` oder `print` aus, erhalten Sie eine Angabe in der Form *Object id #1*. Das liegt daran, dass Sie ein Objekt als String, Zeichenkette, ausgegeben haben. Mit Hilfe der Methode `__toString()` können Sie beeinflussen, wie ein Objekt als Zeichenkette ausgegeben wird. Sie sollten darauf achten, dass Ihre `__toString()` Methode tatsächlich eine Zeichenkette als Rückgabewert liefert. Sonst wird Ihnen PHP bei der Ausführung einen hässlichen Fehler melden.

Ein einfaches Beispiel sehen Sie hier. Bei der Ausgabe mit `echo` wird automatisch `__toString()` aufgerufen und ein einfacher Text angezeigt.

__toString.php

```
<?php
class MeineKlasse
{
  public function __toString()
  {
    return 'Repräsentation des Objekts bei Aufruf mit echo oder
print';
  }
}
$objekt = new MeineKlasse();
echo $objekt;
?>
```

Ausgabe:

Repräsentation des Objekts bei Aufruf mit `echo` oder `print`

clone und __clone

Beim Erzeugen eines Objekts weisen Sie das Objekt einer Variablen zu. Unter diesem Namen können Sie das Objekt ansprechen. Diese Variable enthält eine Referenz auf das Objekt und keine Kopie. Wenn Sie eine echte 1-zu-1 Kopie eines Objektes benötigen, erhalten Sie diese Kopie nicht durch eine einfache Zuweisung mit dem Zuweisungsoperator =, wie Sie es von Variablen gewohnt sind. Bei dieser einfachen Zuweisung würde die Variable nur eine weitere Referenz auf das gleiche Objekt enthalten. Eine echte Kopie eines Objekts erzeugen Sie mit dem Schlüsselwort `clone`. Dieses Objekt können Sie dann unabhängig vom Ausgangsobjekt manipulieren.

An einem Beispiel machen wir uns den Unterschied zwischen einer Zuweisung und der Verwendung von `clone` zum Kopieren eines Objekts klar.

clone.php

```
<?php
class MeineKlasse
{
  private $zahl = 15;
  public function SetZahl($var)
  {
    $this->zahl = $var;
  }
  public function GetZahl()
  {
    return $this->zahl;
  }
} /* Ende von MeineKlasse */
```

```
function zeige($obj)
{
  echo $obj->GetZahl() . '<br>';
}

$objekt1 = new MeineKlasse();
$objekt2 = $objekt1;   /* objekt2 ist eine Referenz */
$objekt2->SetZahl(50); /* Eigenschaft zahl wird auf 50 gesetzt */
zeige($objekt1);

unset($objekt1, $objekt2); /* Löschen der beiden Objekte */

$objekt1 = new MeineKlasse();
$objekt2 = clone $objekt1; /* objekt2 ist eine echte Kopie */
$objekt2->SetZahl(50);
zeige($objekt2);
zeige($objekt1);
?>
```

Nach der Zuweisung `$objekt2 = $objekt1;` ist `$objekt2` eine Referenz auf `$objekt1`. Das zeigt sich dadurch, dass wir eine Eigenschaft von `$objekt2` mit der Methode `SetZahl(50)` verändern, was sich auch auf unser Ausgangsobjekt `$objekt1` auswirkt. Die erste Zeile der Ausgabe der Zahl 50 bestätigt es. Erzeugen wir mit `$objekt2 = clone $objekt1` eine echte Kopie von `$objekt1`, können wir diese Kopie unabhängig manipulieren. Die Ausgabe der Zahl 15 in der dritten Zeile beweist, dass `$objekt2` unabhängig von `$objekt1` ist.

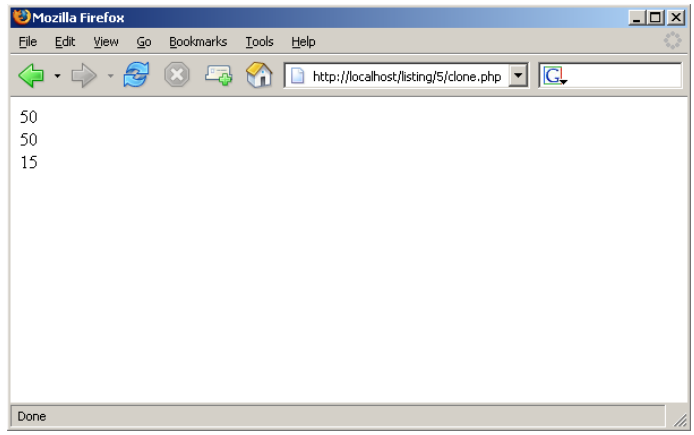

Abb. 4.11: Ausgabe von clone.php

Unter Umständen möchten Sie gerne den Kopiervorgang bei Objekten beeinflussen. Bei zwei Anwendungsfällen kann das von Nutzen sein. Sie könnten den Wunsch haben, bestimmten Eigenschaften einen anderen Wert zuzuweisen. Den Kopiervorgang zu beeinflussen kann auch dann Sinn machen, wenn das zu kopierende Objekt Referenzen auf andere Objekte enthält. Mit `clone` wird nämlich eine so genannte flache Kopie erzeugt. Das bedeutet, dass Referenzen auf andere Objekte innerhalb eines Objekts als Referenzen kopiert werden. Hier kommt uns die magische Funktion `__clone()` zu Hilfe, mit der Sie den Kopiervorgang beeinflussen können. Haben Sie `__clone()` in einer Klasse implementiert und erzeugen mit `clone` eine Kopie dieser Klasse, wird am Ende des Kopiervorgangs automatisch `__clone()` aufgerufen und enthaltene Anweisungen ausgeführt.

Im folgenden Beispiel definiert die Klasse `neuesFahrzeug` eine `__clone()`-Methode. Die Referenz auf ein Objekt vom Typ `Tankstelle` wird bei den Objekten `$vw` und `$ente` als Referenz 1-zu-1 kopiert. An der Ausgabe erkennen wir das daran, dass beide Super tanken. Bei den Objekten `$mercedes` und `$kombi` wird nach der 1-zu-1 Kopie die Referenz auf das Objekt vom Typ `Tankstelle` mit Hilfe der `__clone()` Methode ebenfalls kopiert. Das hat zur Folge, dass beide unterschiedlichen Kraftstoff tanken, nämlich Super oder Diesel.

__clone.php

```
<?php
class Tankstelle
{
    public $sorte = '';
    public function __construct($tanken)
    {
    $this->sorte = $tanken;
    }
    public function setSorte($tanken)
    {
    $this->sorte = $tanken;
    }
} /* Ende class Tankstelle */
class Fahrzeug
{
  public $marke = '';
  public $brennstoff = NULL;
  public function __construct($auto, $sprit)
  {
    $this->marke = $auto;
    $this->brennstoff = new Tankstelle($sprit);
  }
```

```
  public function setMarke($auto)
  {
    $this->marke = $auto;
  }
  public function gibAuskunft()
  {
    echo 'Ich bin ein ' . $this->marke;
    echo ' und tanke ' . $this->brennstoff->sorte . '<br>';
  }
} /* Ende class Fahrzeug */
class neuesFahrzeug extends Fahrzeug
{
  public function __clone()
  {
    $this->brennstoff = clone $this->brennstoff;
  }
} /* Ende class neuesFahrzeug */
$vw = new Fahrzeug('VW', 'NormalBenzin');
$ente = clone $vw;
$ente->setMarke('Citroen');
$ente->brennstoff->setSorte('Super');
$vw->gibAuskunft();
$ente->gibAuskunft();
echo '<br>';
$mercedes = new neuesFahrzeug('Benz', 'Extra Super');
$kombi = clone $mercedes;
$kombi->setMarke('Lada');
$kombi->brennstoff->setSorte('Diesel');
$mercedes->gibAuskunft();
$kombi->gibAuskunft();
```

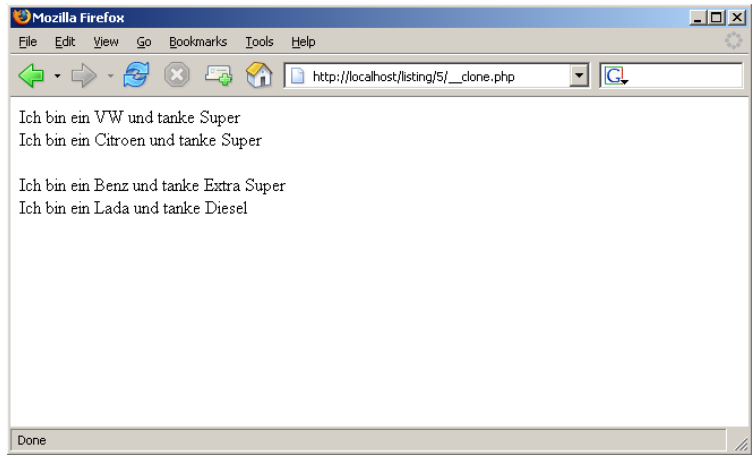

Abb. 4.12: Beeinflussung des Kopiervorgangs mit __clone

4.2.2 Ausnahmebehandlung mit Exception

Von anderen Programmiersprachen wie Java kennen Sie vielleicht die Ausnahmebehandlung mit `try`, `throw` und `catch`. Mit `try` wird versucht, eine Aktion auszuführen, mit `throw` werfen Sie eine Ausnahme (Exception) und mit `catch` können Sie die geworfene Ausnahme einfangen und reagieren.

Dieses Modell sollten Sie wirklich nur zur Behandlung von Ausnahmen einsetzen und nicht für logische Entscheidungen in Programmabläufen missbrauchen. Für die logische Programmablaufsteuerung stehen Ihnen die Möglichkeiten aus Kapitel 2.3 zur Verfügung. Beispiele für echte Ausnahmen, die in Skripten vorkommen könnten und die Sie entsprechend behandeln möchten, sind: eine Datenbankverbindung kommt nicht zustande oder das Öffnen einer Datei schlägt fehl. Als Faustregel sollten Sie sich merken: Eine echte Ausnahme liegt immer dann vor, wenn eine Aktion fehlschlägt, die für den weiteren ordnungsgemäßen Ablauf Ihres Skripts von entscheidender Bedeutung ist und auf die Sie reagieren wollen.

Die interne Klasse Exception

PHP stellt Ihnen die interne Klasse *Exception* bereit, deren Methoden Sie für die Behandlung von Ausnahmen verwenden können. An einem Beispiel können wir sehen, wie die Ausnahmebehandlung funktioniert.

Eingeschlossen in einen `try`-Block versuchen wir eine Datei zu öffnen und deren Inhalt zu lesen. Falls das Öffnen einer wichtigen Datei fehlschlägt, wird mit `throw new Exception` ein Exception-Objekt erzeugt und geworfen, dem eine Fehlermeldung übergeben wird. Mit einem `catch`-Block werden die möglichen Ausnahmen eingefangen. Mit der Anweisung `catch (Exception $ausnahme)` wird das Objekt `Exception` der Variable `$ausnahme` zugewiesen. In unserem Beispiel verwenden wir die Methoden `getMessage()`, `getFile()` und `getLine()` des Exception-Objekts, um die übergebene Fehlermeldung, die Datei und die Zeile, in der ein Fehler aufgetreten ist, auszugeben.

ausnahme.php

```
<?php
try {
  if ( ! ($fh = @fopen('WichtigeDatei.txt', 'rb')) ) {
    throw new Exception('Die Datei konnte nicht geöffnet werden!');
}
  if ( ($text = fread($fh, 1024)) === FALSE ) {
    throw new Exception('Die Datei konnte nicht gelesen werden.');
  }
  @fclose($fh);
  echo $text . '<br>';
} catch (Exception $ausnahme) {
```

```
  echo $ausnahme->getMessage() . '<br>';
  $file = $ausnahme->getFile();
  $line = $ausnahme->getLine();
  echo "Fehler aufgetreten im Skript $file in Zeile $line <br>";
  }
echo 'weiter im Skript.... <br>';
?>
```

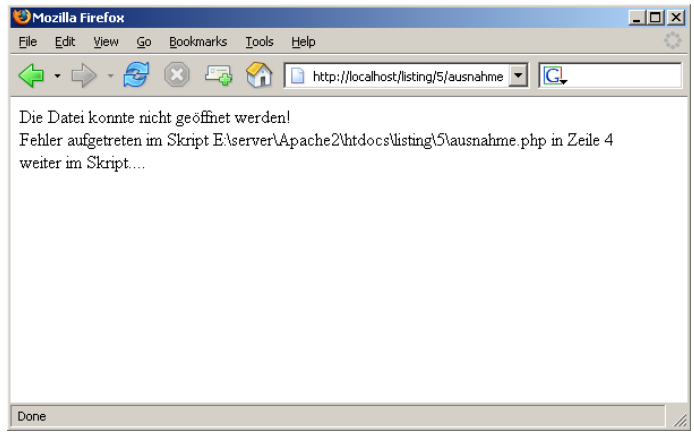

Abb. 4.13: Ausnahmebehandlung mit der internen Klasse Exception

Sobald Sie innerhalb eines `try`-Blocks eine Ausnahme werfen, wird dieser Block verlassen und die Programmausführung mit dem Code innerhalb des `catch`-Blocks fortgesetzt. Sie müssen eine geworfene Ausnahme mit `catch` abfangen, sonst wird Ihnen PHP einen Fehler melden.

An welcher Stelle und in welchem Kontext innerhalb eines `try`-Blocks eine Ausnahme geworfen wird, ist nicht entscheidend. Sie können auch einen Funktionsaufruf mit `try` und `catch` umschließen und eine Ausnahme innerhalb der aufgerufenen Funktion werfen.

ausnahme2.php

Wie das funktioniert, zeigt uns das folgende Beispiel:

```
<?php
function leseDatei($datei)
{
  if ( ! ($fh = @fopen('$datei', 'rb')) ) {
    throw new Exception('Die Datei ' . $datei . ' konnte nicht
                         geöffnet werden! ');
  }
```

```
  if ( ($text = fread($fh, 1024)) === FALSE ) {
    throw new Exception('Die Datei ' . $datei . 'konnte nicht
                         gelesen werden');
  }
  @fclose($fh);
  return $text;
}
try {
  $inhalt = leseDatei('WichtigeDatei.txt');
  echo $inhalt;
} catch (Exception $ausnahme) {
    echo $ausnahme->getMessage() . '<br>';
    $file = $ausnahme->getFile();
    $line = $ausnahme->getLine();
    echo "Aufgetreten im Skript $file in Zeile $line <br>";
  }
echo 'weiter im Skript.... <br>';
?>
```

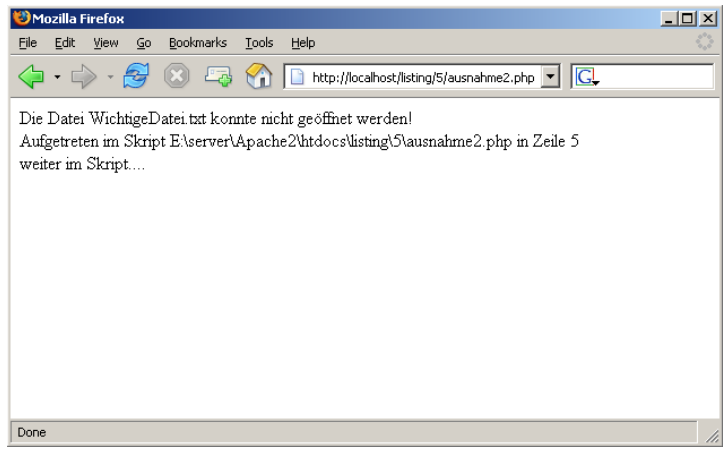

Abb. 4.14: Werfen einer Ausnahme innerhalb einer Funktion

4.2.3 Objektschnittstellen mit interface

PHP bietet Ihnen die Möglichkeit, Objektschnittstellen (Interfaces) zu definieren. Eine Schnittstelle ist ein Spezialfall einer abstrakten Klasse. In einer Schnittstelle definieren Sie ausschließlich Methoden und ihre Parameter, ohne allerdings die Funktionalität der Methoden festzulegen. Sie legen nur die Signatur der zu implementierenden Methoden fest. Zur Signatur gehört der Methodenname und die Parameterliste. Von einer Schnittstelle können Sie nicht ableiten, sie kann nur von einer Klasse mit dem Schlüsselwort `implements` verwendet werden. Eine Klasse, die eine Schnittstelle implementiert,

muss alle Methoden mit den erforderlichen Parametern gemäß der von der Schnittstelle vorgegebenen Signatur definieren. Eine Schnittstelle ist damit eine abstrakte Festlegung von Methoden und ihren Parametern. Eine Klasse, die eine Schnittstelle implementiert, gibt das Versprechen, diese Methoden mit Funktionalität auszustatten.

interface.php

Wir definieren die Schnittstelle `KontoBewegung` mit den Methoden `einzahlen()` und `auszahlen()`.

```
<?php
interface KontoBewegung
{
  public function einzahlen($betrag);
  public function auszahlen($betrag);
}

class GiroKonto implements KontoBewegung
{
  private $KontoStand = 350;
  public function einzahlen($betrag)
  {
    $this->KontoStand += $betrag;
  }
  public function auszahlen($betrag)
  {
     $this->KontoStand -= $betrag;
  }

  public function GetKontostand()
  {
     return $this->KontoStand;
  }
}

$Dresdner = new GiroKonto();
$Dresdner->einzahlen(135);
echo 'Kontostand nach Einzahlung: ' . $Dresdner->GetKontostand() .
     '<br>';
$Dresdner->auszahlen(500);
echo 'Kontostand nach Auszahlung: ' . $Dresdner->GetKontostand() .
     '<br>';
?>
```

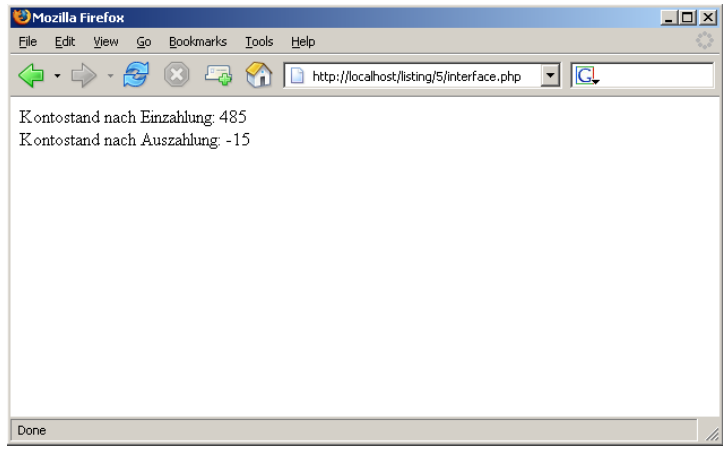

Abb. 4.15: Verwendung von Objektschnittstellen

Eine Schnittstelle darf nur Methoden definieren und sämtliche Methoden müssen als `public` gekennzeichnet sein. Eine Klasse kann mehrere Schnittstellen implementieren, die bei der Deklaration durch Kommata getrennt werden. Damit lässt sich auf Umwegen eine Art Mehrfachvererbung realisieren.

4.3 Funktionen zum Umgang mit Klassen und Objekten

Oft ist es sehr hilfreich, Informationen über Klassen und Objekte zu erhalten, z.B. welche Methoden stellt eine Klasse zur Verfügung, wie ist der Name des aktuellen Objekts, über welche Eigenschaften verfügt ein Objekt oder eine Klasse. Ähnliche und viele weitere Fragen können Sie mit Hilfe der folgenden Funktionen beantworten.

Funktion	Beschreibung
`class_exists($klassenname, [$autoload])`	Prüft, ob eine Klasse Ihrem Skript bekannt ist, optional kann `$autoload` `TRUE` oder `FALSE` sein. Bei `TRUE` (Standard) wird eine vorhandene `__autoload`-Funktion aufgerufen.
`method_exists($objekt, $methodenname)`	Prüft, ob ein Objekt über eine Methode verfügt
`interface_exists($interfacename, [$autoload])`	Prüft, ob eine Schnittstelle Ihrem Skript bekannt ist, optional kann `$autoload` `TRUE` oder `FALSE` sein. Bei `TRUE` (Standard) wird eine vorhandene `__autoload` Funktion aufgerufen.

Tabelle 4.2: Hilfreiche Funktionen für den Umgang mit Objekten und Klassen

Tabelle 4.2: Hilfreiche Funktionen für den Umgang mit Objekten und Klassen (Forts.)

Funktion	Beschreibung
`get_class($objekt)`	Liefert den Namen der Klasse, zu der `$objekt` gehört
`get_declared_classes()`	Liefert ein Array aller Ihrem Skript bekannten Klassen
`get_declared_interfaces()`	Liefert ein Array aller Ihrem Skript bekannten Schnittstellen
`get_class_methods($name)`	Liefert alle Methoden einer Klasse oder eines Objekts. Für `$name` können Sie ein Objekt oder einen Klassennamen einsetzen.
`get_class_vars()`	Liefert ein assoziatives Array mit den Namen und Werten der Eigenschaften einer Klasse
`get_object_vars()`	Liefert ein assoziatives Array mit den Namen und Werten der Eigenschaften eines Objekts
`get_parent_class($name)`	Falls `$name` ein Objekt ist: Liefert den Namen der Elternklasse der Klasse, von der `$name` eine Instanz ist. Falls `$name` ein Klassenname ist: Liefert den Namen der Elternklasse der Klasse mit diesem Namen.
`is_subclass_of($objekt, $elternklasse)`	Prüft, ob `$objekt` zu einer abgeleiteten Klasse von `$elternklasse` gehört
`instanceof`	Prüft, ob ein Objekt eine Instanz einer bestimmten Klasse ist

Der Rückgabewert dieser Funktionen ist meist `TRUE` oder `FALSE`, falls kein abweichender Wert angegeben ist.

Eine Fülle an Informationen über Klassen, Objekte, Methoden, Schnittstellen, ja sogar PHP-Erweiterungen können Sie mit Hilfe der Reflection API (**A**pplication **P**rogramming **I**nterface) gewinnen. Die Reflection API ist eine objektorientierte PHP-Erweiterung und in jeder Standardinstallation von PHP 5 verfügbar. Eine umfassende Beschreibung der Möglichkeiten, die Ihnen die Reflection API bietet, würde mindestens ein ganzes Buchkapitel umfassen, deshalb belassen wir es an dieser Stelle mit einem Hinweis auf dieses mächtige Werkzeug. Unter *http://www.php.net/manual/en/language.oop5.reflection.php* finden Sie eine kurze Erläuterung der Reflection API in englischer Sprache.

Als Anreiz, sich mit den Möglichkeiten der Reflection API auseinander zu setzen, stellen wir Ihnen ein Beispiel vor, wie Sie ganz einfach Informationen zu der internen Klasse *Exception* erhalten.

```
<pre>
<?php
Reflection::export(new ReflectionClass('Exception'));
?>
</pre>
```

reflection.php

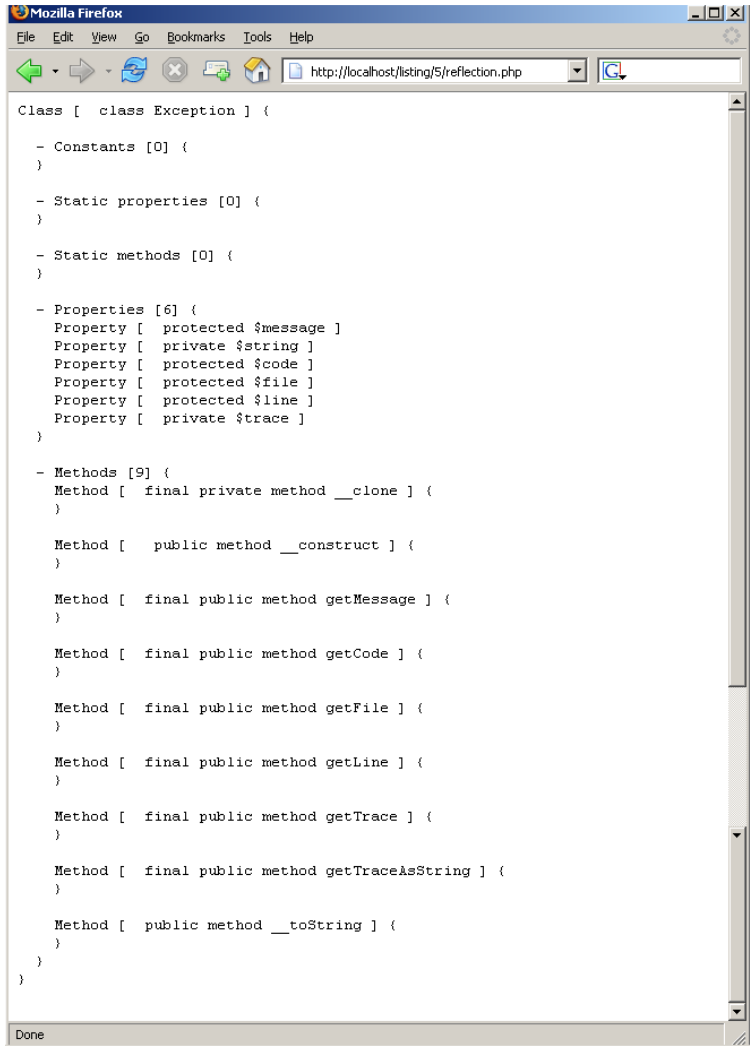

Abb. 4.16: Informationen über die Klasse Exception

4.4 Fragen

1. Welcher Zusammenhang besteht zwischen einer Klasse und einem Objekt?
2. Was versteht man unter dem Begriff *Instanziierung*?
3. Welche Sichtbarkeitsbeschränkungen können Sie mit welcher Wirkung für Methoden verwenden?
4. Was ist der Unterschied zwischen einfacher und mehrfacher Vererbung?
5. Wie können Sie das Überschreiben einer Methode verhindern?
6. Wie können Sie die Ableitung von einer Klasse verhindern?
7. Welche Besonderheiten weisen magische Funktionen auf?
8. Wie erzeugen Sie eine Kopie eines Objekts?
9. Worin besteht der Unterschied zwischen abstrakten und finalen Klassen?
10. Was ist die Signatur einer Methode?