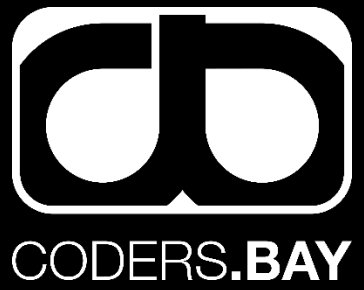

Objektorientierung
Teil 2

CODERS.BAY

Formal erkennst du magische Methoden an den beiden Unterstrichen (__) am Anfang. Zwei magische Methoden hast du bereits kennengelernt - nämlich die Konstruktor- und die Dekonstruktor-Methode die automatisch aufgerufen werden, wenn ein neues Objekt erstellt bzw. ein Objekt zerstört wird.

__SET() UND __GET()

Die beiden magischen Methoden `__set()` und `__get()` werden automatisch aufgerufen, wenn ein Zugriff auf eine nicht definierte Eigenschaft erfolgt: `__set()` bei einem schreibenden, `__get()` bei einem lesenden Zugriff. Wenn du `__set()` und `__get()` in deiner Klasse implementierst, kannst du festlegen, was in diesem Fall geschehen soll.

__SET() UND __GET()

CODERS.BAY

In der Klasse Beispiel wird die Eigenschaft **$vorname** und die Methode **halloSagen()** implementiert. Dann steht die magische Methode **__set()**, die bestimmt, was passieren soll, wenn eine Eigenschaft, die nicht definiert ist, gesetzt wird. **__set()** erwartet zwei Parameter - zuerst die Eigenschaft, die gesetzt wird, und dann den Wert, auf den die Eigenschaft gesetzt wird. Beide kannst du innerhalb der Methode auslesen. Im Beispiel wird gezeigt, dass man je nach gesetzter Eigenschaft auch unterschiedliche Dinge tun kann: Dafür überprüfst du, welchen Wert die Eigenschaft hat. Im Beispiel wird geprüft, ob es sich dabei um die Eigenschaft **$alter** handelt, und falls dem so ist, eine entsprechende Meldung ausgegeben

Wenn du hier mehr Fälle prüfen willst, solltest du die switch-Konstruktion benutzen.

Dann wird die magische **__get()**-Methode implementiert. Sie erwartet nur einen Parameter. Die Eigenschaft, die ausgelesen wird. Auch hier wird eine Meldung ausgegeben.

Dann wird ein neues Objekt erstellt. Das Zuweisen des Vornamens funktioniert wie gewohnt. Anschließend wird versucht, die Eigenschaft **$alter** zu setzen. Jetzt wird die **__set()**-Methode aufgerufen. Es wird der Versuch unternommen, die nicht existierende Eigenschaft **$name** auszugeben. In diesem Fall wird die **__get()**-Methode aufgerufen.

```php
class Beispiel {
   public $vorname;
   public function halloSagen() {
      echo "Hallo {$this->vorname}";
   }
   public function __set($eigenschaft, $wert) {
      if($eigenschaft == "alter") {
         echo "Das Alter ist hier nicht relevant";
      } else {
         echo "Du willst die Eigenschaft $eigenschaft setzen, die es
           nicht gibt.";
      }
   }
   public function __get($eigenschaft) {
      echo "Die Eigenschaft $eigenschaft gibt es nicht.";
   }
}
$bsp = new Beispiel();
$bsp->vorname = "Amina";
$bsp->halloSagen();
echo "<br>\n";
$bsp->alter = 2;
echo "<br>\n";
echo $bsp->name;
```

__SET() UND __GET()

Wann ist der Einsatz von `__get()` und `__set()` sinnvoll? Beispielsweise könnten seltener benötigte Eigenschaften einer Klasse in eine Datei ausgelagert werden. Diese wird erst bei Bedarf geladen - über eine `__get()`-Methode, wenn auf eine unbekannte Eigenschaft zugegriffen wird.

__call() und callStatic()
Magie für Methoden

CODERS.BAY

Das, was `__get()` und `__set()` bei Eigenschaften machen, macht `__call()` für Methoden. `__call()` wird - sofern in der Klasse implementiert - aufgerufen, wenn eine nicht definierte Methode eingesetzt wird:

Im Beispiel siehst du wieder die Klasse `Beispiel`. Implementiert ist dieses Mal die Methode `__call()`. Sie erwartet zwei Parameter: zuerst den Methodennamen und danach die Parameterliste. Auf beide kannst du innerhalb der magischen Methode `__call()` zugreifen. Die Parameterliste (der zweite übergebene Parameter) ist wie zu erwarten ein Array. Ob Parameter übergeben wurden, wird mit `!empty()` überprüft, und gegebenenfalls werden die übergebenen Parameter ausgegeben.

```php
class Beispiel {
   public $vorname;
   public function halloSagen() {
      echo "Hallo {$this->vorname}";
   }
   public function __call($methode, $argumente) {
      echo "Sie haben $methode aufgerufen. Die gibt's nicht.";
      if(!empty($argumente)) {
         echo "Die übergebenen Argumente: ";
         foreach ($argumente as $el) {
            echo "$el";
         }
      }
   }
}
$bsp = new Beispiel();
echo "<br>\n";
$bsp->verabschieden("tschüss", "morgen");
```

__call() und callStatic()
Magie für Methoden

CODERS.BAY

Außerdem kannst du auch den Aufruf von unbekannten statischen Methoden abfangen. Hierfür gibt es die magische Methode `__callStatic()`

```php
class Beispiel {
   static function __callStatic($methode, $argumente) {
      echo "Sie haben die statische Methode '$methode' aufgerufen. ";
   }
}
Beispiel::etwas();
```

Ausgabe steuern über __toString()

CODERS.BAY

Möchtest du schnell Informationen zu einem Objekt erhalten, kannst du `var_dump()` benutzen. Dieses liefert dir - neben der Information, dass es sich um ein Objekt handelt - die aktuell gesetzten Eigenschaften. Nehmen wir als Beispiel noch einmal die Klasse `Kunde`.

```php
require_once "kunde.php";
$neuerKunde = new Kunde("Anja");
$neuerKunde->speichern(20);
echo "<pre>";
echo "Infos über var_dum()\n";
var_dump($neuerKunde);
/** Auch print_r() liefert brauchbare Ergebnisse */
echo "Infos über print_r()\n";
print_r($neuerKunde);
echo "</pre>";
```

Schließlich kannst du Objekte wie Arrays in einer `foreach`-Schleife durchlaufen und erhältst alle öffentliche Eigenschaften und die aktuellen Werte

```php
echo "Objekte lassen sich wie ein Array durchlaufen<br>\n";
foreach($neuerKunde as $k => $v) {
   echo "$k: $v<br>\n";
}
```

Ausgabe steuern über __toString()

CODERS.BAY

Wenn du jedoch direkt ein Objekt per `print` oder `echo` ausgeben lässt, liefert dies ein unbefriedigendes Ergebnis. Dies lässt sich mit der magischen Methode `__toString()` ändern. Wenn du diese in einer Klasse definierst, kannst du dort bestimmen, was ausgegeben werden soll, wenn `print` oder `echo` bei einem Objekt benutzt wird.

Die magische Methode `__toString()` muss einen String zurück liefern

Ausgabe steuern über __toString()

CODERS.BAY

Im Beispiel wird in der Klasse Premiumkunde die `__toString()` -Methode implementiert. In dieser Methode wird ein String mit Informationen zum jeweiligen Objekt zusammengestellt. Die Eigenschaften sollten in Form einer Liste ausgegeben werden. Die `__toString()` -Methode gibt diesen String mit `return` zurück.

Das Objekt wird direkt mit `echo` ausgegeben. Jetzt wird die `__toString()` -Methode aufgerufen und gibt das gewünschte Ergebnis aus.

```php
require_once "kunde.php";
class Premiumkunde extends Kunde {
   public $speicherGesamt = 100;
   public $farbSchema;
   public function __construct(
           $name,
           $speicherVerbrauch = 0,
           $farbSchema = "Sonnenaufgang") {
      $this->name = $name;
      $this->speicherVerbrauch = $speicherVerbrauch;
      $this->farbSchema = $farbSchema;
   }
   public function __toString() {
      $string = "Instanz von Premiumkunde<br>\n"
      . "Folgende Eigenschaften sind definiert<br>\n"
      . "<ul>\n"
      . "<li>Name: " . $this->name . "</li> \n"
      . "<li>Name: " . $this->speicherVerbrauch . "</li> \n"
      . "<li>Name: " . $this->farbSchema . "</li> \n"
      ."</ul>\n";
      return $string;
   }
}
$pk = new Premiumkunde ("Hans-Heinrich");
echo $pk;
```

Wichtige Funktionen

Informationen über Objekte bzw. Klassen erhältst du auch über verschiedene von PHP vordefinierte Funktionen:

- `class_exists()`
  - erwartet als Parameter den Namen einer Klasse und gibt true zurück, wenn die Klasse existiert, ansonsten false
- `interface_exists()`
  - überprüft entsprechend die Existenz eines Interfaces
- `get_class_methods()`
  - erwartet als Parameter den Namen einer Klasse und gibt ein Array mit den definierten Methoden zurück
- `get_class_vars()`
  - erwartet als Parameter den Namen einer Klasse und gibt ein Array mit den vordefinierten Eigenschaften zurück
- `get_object_vars()`
  - liefert die öffentlichen Eigenschaften eines Objekts
- `get_class()`
  - zur Ermittlung der Klassennamen eines Objekts
- `get_parent_class()`
  - liefert entsprechend zu einem Objekt den Namen der Elternklasse
- `is_subclass_of()`
  - erwartet zwei Strings als Parameter und gibt an, ob das zuerst genannte Objekt eine Instanz einer Unterklasse der Klasse ist, die im zweiten Parameter genannt wird
- `instanceof`
  - über diesen Operator kannst du feststellen, ob ein gegebenes Objekt zu einer bestimmten Klasse gehört

AUFGABE

CODERS.BAY

- Nehmt die Klasse CAR und implementiert
  - __get & __set
  - Eine Sinnvolle __toString ausgabe
  - Probiert die Funktionen auf Seite 10 aus

Probiert alles aus -> 45 Minuten Zeit

Referenzen, Klone und Vergleiche

CODERS.BAY

Eine Besonderheit zeigt sich bei der Zuweisung von Objekten zu Variablen.

Angenommen, du speicherst in eine Variable einen Wert

```
$a = 4;
```

und weist dann die Variable **$a** einer anderen Variable zu

```
$b = $a;
```

Dann hat **$b** ebenfalls den Wert 4. Änderst du jetzt den Wert von **$a**, so ist **$b** davon nicht betroffen

Referenzen und Klone

Wenn du ein Objekt einer anderen Variablen zuweist, kopierst du damit nicht das gesamte Objekt, sondern die Variable erhält *nur eine Referenz* auf dieses Objekt.

Im Beispiel wird eine einfache Klasse namens Beispiel erzeugt. Sie hat die Eigenschaft `$farbe` und eine Methode `info()`, die die aktuelle Farbe ausgibt.

Es wird eine Instanz der Klasse `Beispiel` erzeugt. Anschließend wird dieses Objekt der Variablen `$obj2` zugewiesen. Schließlich wird die Eigenschaft `$farbe` des Objekts `$obj1` auf `blau` gesetzt. Für beide Objekte wird die Methode `info()` aufgerufen. Da `$obj2` nur eine Referenz auf `$obj1` enthält, das das Ergebnis dasselbe. Beide Male wird ausgegeben: "Farbe ist blau".

Das zeigen auch nochmal die letzten beiden Zeilen wo die Eigenschaft `$farbe` von `$obj1` auf den Wert orange gesetzt wird. Liest man danach diese Eigenschaft über die `info()`-Methode aus, zeigt sich, dass auch bei `$obj2` die Eigenschaft `$farbe` den Wertorange hat. Da `$obj2` nur eine Referenz auf `$obj1` enthält, betreffen alle Änderungen, die du an `$obj1` durchführst, genauso `$obj2`.

Wenn du Objekte an Funktionen übergibst, passiert genau dasselbe - sie werden als Referenzen übergeben:

```php
class Beispiel {
   public $farbe;
   public function info() {
      echo "Farbe ist {$this->farbe}<br>\n";
   }
}
$obj1 = new Beispiel();
$obj2 = $obj1;
$obj1->farbe = "blau";
$obj1->info();
$obj2->info();
$obj1->farbe = "orange";
$obj2->info();
```

Referenzen und Klone

Falls du jedoch nicht eine Referenz auf ein Objekt erstellen möchtest, sondern ein Objekt mit allen Eigenschaften kopieren möchtest, sodass du danach zwei Objekte hast, die sich unabhängig voneinander verändern lassen, dann musst du ein Objekt über `clone` klonen.

`$obj3` wird als Klon von `$obj1` mit allen Eigenschaften erstellt. Wird jedoch nach dem Klonvorgang `$obj1` verändert, so betrifft dies `$obj3` nicht mehr.

Soll beim Klonen mehr geschehen, musst du die magische Methode `__clone()` in deiner Klasse implementieren. Sie wird automatisch aufgerufen, wenn ein Objekt geklont wird. Hier kannst du beispielsweise einzelne Eigenschaften wieder zurücksetzen, bei denen ein anderer Anfangswert wichtig ist.

```php
class Beispiel {
   public $farbe;
   public function info() {
      echo "Farbe ist {$this->farbe}<br>\n";
   }
}
$obj1 = new Beispiel();
$obj2 = $obj1;
$obj1->farbe = "blau";
$obj1->info();
$obj2->info();

$obj3 = clone $obj1;
$obj1->farbe = "orange";
$obj3->info();
$obj2->info();
```

Objekte vergleichen

Zum Vergleichen von Objekten kannst du `==` und `===` verwenden. Diese verhalten sich jedoch bei Objekten anders als bei normalen Variablen.

`==` überprüft, ob die Eigenschaften identisch sind, `===` überprüft hingegen, ob der Objekt-Handler identisch ist, also auf dasselbe Objekt verweist.

Im Beispiel wird wieder eine einfache Beispielklasse mit der Eigenschaft `$farbe` benutzt. Dann wird ein neues Objekt instanziiert und die Eigenschaft `$farbe` auf `Blau` gesetzt.

Weiters wird eine Referenz auf `$obj1` erstellt und in `$obj2` gespeichert. `$obj1` und `$obj2` verweisen so auf dasselbe Objekt.

Mit `clone` wird eine Kopie von `$obj1` erstellt und in der Variable `$obj3` gespeichert. `$obj3` hat damit dieselben Eigenschaften wie `$obj1`, ist aber ein eigenständiges Objekt.

Mit `var_dump()` werden die Ergebnisse der Vergleiche direkt ausgegeben.

```php
class Beispiel {
    public $farbe;
    public function info() {
        echo "Farbe ist { $this->farbe } <br>\n";
    }
}
$obj1 = new Beispiel();
$obj1->farbe = "Blau";
$obj2 = $obj1;
$obj3 = clone $obj1;

echo "<pre>";
var_dump($obj1 == $obj2);  // bool (true)
var_dump($obj2 === $obj2); // bool (true)

var_dump($obj1 == $obj3);  // bool (true)
var_dump($obj1 === $obj3); // bool (false)
echo "</pre>";
```

NAMENSRÄUME

CODERS.BAY

Es kann bei größeren Projekten leicht zu Namenskonflikten kommen, wenn es mehrere Funktionen mit einem vorgegebenen Namen gibt, bspw. zwei Funktionen mit dem Namen `start()`.

Dies ist bei der objektorientierten Programmierung weniger wahrscheinlich, weil die Funktionen - dann als Methoden - an die jeweiligen Klassen gebunden sind. Zwei Methoden mit dem Namen `start()`, die bei unterschiedlichen Klassen definiert sind, stören sich nicht.

Das PHP-Manual vergleicht Namensräume mit Ordnern im Dateisystem. Innerhalb von zwei verschiedenen Ordnern können Dateien desselben Namens vorkommen, ohne dass sich diese in die Quere kommen.

CODERS.BAY

- Einen Namensraum gibst du hinter dem Schlüsselwort `namespace` an
- Danach kannst du die zu dem Namensraum gehörigen Klassen, Funktionen und Konstanten definieren
- Die namespace-Anweisung muss sich im Code zualleroberst befinden. Es dürfen sich keine PHP-Befehle (mit Ausnahme von `declare()` zur Angabe einer Codierung) darüber befinden, auch darf kein HTML-Code davor ausgegeben werden. Ansonsten gibt es eine Fehlermeldung!
- Innerhalb der Datei selbst, in der du den Namensraum deklariert hast, kannst du dann direkt die Klassen, Funktionen oder Konstanten direkt benutzen, weil sie sich innerhalb dieses Namensraums befinden
- **Über die vordefinierte Konstante __NAMESPACE__ hast du Zugriff auf den aktuellen benutzten Namensraum**
- Wenn du hingegen die Namensraum-Datei in eine andere Datei einbindest und hierin au feine Klasse, Funktion oder Konstante zugreifen möchtest, musst du den Namenraum davor angeben. Dafür gibst du zuerst den Namenraum, dann einen Backslash als Separator und schließlich die Klasse, Funktion oder Konstante an:

```php
namespace meinProjekt;
class Benutzer {}
function wastun() {
    echo "getan";
}
const zahl = 42;

wastun();
$a = new Benutzer();
echo " " . zahl;
```

```php
require "namensraum.php";
/* das geht nicht */
// wastun();

/* das hingegen geht */
meinProjekt\wastun();
```

CODERS.BAY

- Die Namen von Namensräumen können selbst auch den Backslash beinhalten, sodass sich diese noch besser organisieren lassen. Üblicherweise baut man Namensräume so auf, dass zuerst der Name des Herstellers und danach der Name des Pakets angegeben wird. Dadurch ist bei der Verwendung immer klar, worum es sich handelt, und gleichnamige Pakete kommen sich nicht in die Quere

```
namespace pear2\text_diff;
namespace zend\controller;
```

Absolut und relativ

Selbstverständlich können mehrere Dateien denselben Namensraum definieren. Aber auch das Umgekehrte ist möglich: die Definition von mehreren Namensräumen innerhalb einer Datei.

```
namespace Eins;
const zahl = 1;
namespace Zwei;
```

Nehmen wir jetzt an, dass innerhalb des Namensraums `Zwei` die Konstante `zahl` aufgerufen wird.

Die erste Möglichkeit, die fehlschlägt, ist, den Namen der Konstante direkt anzugeben:

```
namespace Zwei;
echo zahl;
```

Das kann nicht funktionieren, da `zahl` ja im aktuellen Namensraum nicht definiert ist.

Ein zweiter Ansatz könnte so aussehen, dass man den Namensraum folgendermaßen angibt:

```
namespace Zwei;
echo Eins\zahl;
```

Aber auch das funktioniert nicht, denn `Eins\zahl` ist eine relative Angabe und wird damit ***im Verhältnis zum aktuellen Namensraum Zwei*** aufgelöst.

Absolut und relativ

Um innerhalb des Namensraums `Zwei` auf ein Element des Namensraums `Eins` zuzugreifen muss eine ***absolute Angabe*** gewählt werden, indem vor der Namensraumangabe ein Backslash geschrieben wird

```
namespace Zwei;
echo \Eins\zahl;
```

Diese Art von absoluten und relativen Angaben erinnert deutlich an die Pfadangaben im Dateisystem, und man kann sie ganz ähnlich betrachten. Es gibt eine alternative Syntax für die Verwendung von mehreren Namensräumen innerhalb einer Datei: Es können den zum jeweiligen Namensraum gehörenden Code auch in geschweifte Klammern eingefasst werden:

```
namespace Eins {
const zahl = 1;
}
namespace Zwei {
echo \Eins \zahl;
}
```

Wichtig ist aber dann, dass kein Code außerhalb der Namensraum-Angaben, also außerhalb der geschweiften Klammern notieret wird, sonst ergibt es eine Fehlermeldung.

Abkürzungen: use benutzen

Wenn die Namensräume lang werden, kann das viel Schreibarbeit bedeuten. Verkürzungen sind über `use` möglich.

Nehmen wir einmal folgende "Bibliothek" an, die einen Namensraum definiert und hier der Einfachheit halber nur eine Konstante:

```php
namespace anwendung\bibliothek;
const zahl = 42;
```

Wenn jetzt außerhalb dieser Datei auf die Konstante zugegriffen wird, muss der Namensraum angegeben werden, also bspw. `anwendung\bibliothek\zahl`. Kürzer geht es über `use`:

```php
require_once "bibliothek.php"
use anwendung\bibliothek as a;
echo a\zahl;
```

Mit `use` wird `a` als Alias für den Namensraum definiert, sodass über `a\zahl` auf die Konstante zugegriffen werden kann

Abkürzungen: use benutzen

Außerdem kann mit `use` auch ein Alias ***für Klassen*** definiert werden.

```php
namespace meinprojekt;
class Benutzer {
   public function __construct() {
      echo "Benutzer erstellt";
   }
}
```

Jetzt soll die Klasse in einer anderen Datei verwendet werden, allerdings soll nicht immer der vollständige Name benutzt werden. Dafür ist `use` da:

Nach der Angabe von `use meinprojekt\Benutzer` kann die Klasse direkt als `Bentuzer()` verwendet werden.

```php
require "namensraum_beispiel.php";

use meinprojekt\Benutzer;
$a = new Benutzer();
```

Das geht auch bei Funktionen:

```php
use function My\Full\functionName as func;
```

oder bei Konstanten:

```php
use const My\Full\CONSTANT;
```

Mit `as` können beliebige Aliasnamen für Klassen, Funktionen, Konstanten vergeben werden.

Abkürzungen: use benutzen

CODERS.BAY

Seit PHP 7 können **use**-Angaben auch gruppiert werden

```
use some\namespace\ClassA;
use some\namespace\ClassB;
use some\namespace\ClassC as C;
```

```
use some\namespace\{ClassA, ClassB, CassC as C};
```

Globaler Namensraum

CODERS.BAY

Der globale Namensraum ist der namenlose Namensraum, in dem sich beispielsweise alle von PHP vordefinierten Funktionen wie `htmlspecialchars()`, `date()` usw. befinden.

Die magische Konstante `__NAMESPACE__` ist beim globalen Namensraum leer

Wenn innerhalb eines Namensraums ein Element aus dem globalen Kontext benutzt wird, reicht ein Backslash am Anfang

Diesbezüglich gibt es aber einen Unterschied zwischen Klassen auf der einen Seite und Funktionen und Konstanten auf der anderen Seite. Dieser zeigt sich wenn ein Element im aktuellen Namensraum ***nicht gefunden wird***: Dann wird nämlich bei Funktionen und Konstanten im globalen Namensraum gesucht, ob eine entsprechende Funktion bzw. Konstante dort existiert. Bei Klassen passiert das nicht. Wenn eine Klasse aus dem globalen Namensraum benutzt wird, muss, wenn es sich innerhalb eines anderen Namensraum befindet, vor dem Klassennamen immer ein Backslash stehen.

Das heißt, es kann, sofern es im Namensraum `Eins` keine Funktion mit Namen `date()` gibt, ebenfalls folgender Code genutzt werden kann, um `date()` aus dem globalen Kontext aufzurufen:

```php
namespace Eins;
/* ruft date() aus dem globalen
Kontext auf */
$b = \date(...);
```

```php
namespace Eins;
/* ruft date() aus dem globalen
Kontext auf, wenn date() im
aktuellen Namensraum nicht definiert
ist */
$b = date(...);
```

Vollständigen Klassennamen ermitteln

```php
namespace Namens\Raum\Verschachtelt;
class BeispielKlasse { };

echo BeispielKlasse::class;
echo "<br>\n";
```

Über `::class` kann der vollständige Klassennamen ermittelt werden, was besonders praktisch in Kombination mit (komplexen) Namensräumen ist.

`BeispielKlasse::class` liefert dann den vollständigen Klassennamen inklusive Namensraumangabe, das heißt hier "Namens\Raum\Verschachtelt\BeispielKlasse".

Traits - Code wiederverwenden

CODERS.BAY

In PHP. gibt es keine Mehrfachvererbung. Einige der damit verbundenen Beschränkungen lassen sich mit ***Traits*** umgehen. Traits erlauben es, ausgewählte Methoden und Eigenschaften in mehreren Klassen zu verwenden, die sich auf unterschiedlichen hierarchieebenen befinden.

Einen Trait definiert man über das Schlüsselwort `trait` und kann dann mit gewünschte Methode angegeben werden

```
trait beispieltrait {
   public function eins() {
      echo "eins<br>\n";
   }
   public function zwei () {
      echo "zwei<br>\n";
   }
}
```

Den Trait benutzt man durch das `use`-Schlüsselwort in einer Klasse:

```
class Beispiel {
use beispieltrait;
}
```

und danach kann auf die Methode zugegriffen werden, die eigentlich im Trait definiert ist, wenn ein neues Objekt der Klasse erstellt wird.

```
$b = new Beispiel();
$b->eins();
$b->zwei();
```

KONFLIKTLÖSUNGEN

Es gibt bestimmte Regeln, die definieren was sich durchsetzt, wenn es widersprechende Angaben innerhalb einer Klasse und in einem Trait gibt:

- Angenommen, eine Klasse benutzt einen Trait mit einer Methode **a**. Gleichzeitig gibt es aber in der Elternklasse der Klasse ebenfalls eine Methode **a**. Dann setzt sich die Methode aus dem Trait durch
- Gibt es gleichnamige Methoden im Trait und in der Klasse, setzt sich die Methode in der Klasse selbst durch

Mehrere Traits nutzen

Es können auch mehrere Traits innerhalb von einer Klasse benutzt werden. Diese werden durch Komma getrennt hinter `use`:

```
trait a { }
trait b { }
class Beispiel {
   use a, b;
}
```

Auf diese Art können auch Traits aus anderen Traits zusammen gesetzt werden.

```
trait a { }
trait b { }
trait c {
   use a, b;
}
```

Wenn jetzt beide Traits eine gleichnamige Methode besitzen, erzeugt das eine Fehlermeldung. Die Lösung für dieses Problem besteht darin, dass man innerhalb von geschweiften Klammern bei `use` angibt, was bei Konfliktfällt gewählt werden soll.

Mehrere Traits nutzen

Es kann `as` ebenfalls genutzt werden, um die Sichtbarkeit von in Traits definierten Methoden zu modifizieren

```
class MyClass! {
use HelloWorld { sayHello as protected; }
}
```

Hinweise zu Traits:

- In Traits kann man als `abstract` definieren. Genauso wie bei Klassen bedeutet das dann, dass die Klasse, die diesen Trait nutzt, die abstrakte Methode implementieren muss
- Methoden in Traits können auf statische Variablen verweisen; auch können in Traits statische Methoden definiert werden
- Meistens wird man in Traits Methoden definieren, du kannst in Traits aber auch Eigenschaften festlegen.

Fehlerbehandlung mit der Exception

Exceptions ("Ausnahmen") ermöglichen eine gezielte Fehlerbehandlung. Bei einem Fehler wird das Programm nicht sofort beendet, sondern der Fehler wird der aufrufenden Funktion mitgeteilt, die dann den Fehler behandeln kann. Zum Einsatz von Exceptions erstellst du neue Instanzen der von PHP vorgegebenen Klasse **`Exception`**.

Der Teil des Programms, der Probleme machen kann, wird dabei in einem **`try`**-Block notiert. Wenn ein Fehler auftritt, wird dieser durch den nachfolgenden **`catch`**-Block abgefangen.

Im Beispiel gibt es eine Funktion namens **`teilen()`**, die das Ergebnis der Division von 1 durch den übergebenen Parameter liefert.

```php
function teilen($x) {
    return 1/$x;
}
```

Ist jedoch der übergebene Wert 0, so würde es zu einem Fehler kommen. In diesem Fall wird in der verbesserten Version des Beispiels über **`throw new Exception()`** eine Ausnahme ausgelöst. Damit wird ein neues Objekt der **`Exception`**-Klasse erstellt.

```php
function teilen($x) {
    if($x==0) {
        throw new Exception("Teilen durch 0!");
    } else {
        return 1/$x;
    }
}
```

Fehlerbehandlung mit der Exception

Im zweiten Beispiel wir die Funktion `teilen()` mit unterschiedlichen Parametern aufgerufen. Diese Funktionsaufrufe sind in einem `try`-Block gekapselt. Darauf folgt ein `catch`-Block mit der Fehlerbehandlung: Im Beispiel wird eine Meldung ausgegeben

```php
try {
    echo teilen(4) . "<br>\n";
    echo teilen(0) . "<br>\n";
    echo teilen(5) . "<br>\n";
} catch(Exception $e) {
    echo "Exception abgefangen: " . $e->getMessage() . "<br>\n";
}
```

Die Funktion `teilen()` wird dreimal aufgerufen. Der erste Aufruf verläuft normal. Beim zweiten Aufruf wird hingegen 0 als Parameter übergeben und damit die `Exception` ausgelöst. Wenn eine `Exception` ausgelöst wird, springt der PHP-Interpreter zum folgenden `catch`-Block. Der Code nach der `Exception` und vor dem `catch`-Block wird hingegen nicht mehr ausgeführt.

Bei der Instanziierung des neuen `Exception`-Objekts wird ein Text übergeben. Dieser beinhaltet die eigentliche Fehlermeldung, die dann über die Methode `getMessage()` ausgegeben wird.

```php
function teilen($x) {
    if($x==0) {
        throw new Exception("Teilen durch 0!");
    } else {
        return 1/$x;
    }
}
```

Fehlerbehandlung mit der Exception

CODERS.BAY

Wenn ein bestimmter Code ausgeführt werden soll, unabhängig davon, ob der `try`- oder der `catch`-Block ausgeführt wird, braucht man `finally`.

Den `finally`-Block notiert man nach `try-catch` - und er beinhaltet den Code, der immer ausgeführt wird. Das ist beispielsweise sinnvoll für irgendwelche Aufräumarbeiten, die auf jeden Fall durchgeführt werden sollen. Im Beispiel wird der `finally`-Block ergänzt

```
try {
    echo teilen(5) . "<br>\n";
} catch(Exception $e) {
    echo "Exception abgefangen: " . $e->getMessage() . "<br>\n";
} finally {
    echo "auf jeden Fall ausgeführt<br>\n";
}
```

Differenzierte Meldungen

Die Fehlerbehandlung kann differenzierter gestaltet werden, indem je nach Situation unterschiedliche Exceptions geworfen (ausgelöst) werden und dabei einen Fehlercode übergeben. Beim Fangen der `Exception` kann je nach Fehlercode unterschiedlich reagiert werden. Auf den Fehlercode kann man über die Methode `getCode()` der `Exception` zugreifen.

Im Beispiel wird unterschieden: Ist der übergebene Parameter nicht numerisch, wird eine `Exception` mit dem Fehlercode 1 ausgelöst; ist er 0, erfolgt eine `Exception` mit dem Fehlercode 2.

Beim Fangen der Exception wird unterschieden: Wenn der Fehlercode, der über `$e->getCode()` ermittelt wird, den Wert 2 hat, wird eine andere Meldung ausgegeben, als wenn er den Wert 1 hat. Selbstverständlich lassen sich hier auch beliebige andere Reaktionen angeben - auch bspw. ein Abbruch des Skripts über `exit`.

```php
function teilen($x) {
    if(!is_numeric($x)) {
        throw new Exception("keine Zahl", 1);
    } elseif ($x == 0) {
        throw new Exception("Teilen durch 0!", 2);
    } else {
        return 1/$x;
    }
}

try {
    echo teilen(4) . "<br>\n";
    echo teilen("hallo") . "<br>\n";
} catch (Exception $e) {
    if($e->getCode() == 2) {
        echo "Falscher Wert: " - $e->getMessage();
    } elseif($e->getCode() == 1) {
        echo "Falscher Datentyp: " . $e->getMessage();
    }
}
```

Differenzierte Meldungen

CODERS.BAY

Die in PHP vordefinierte Exception-Klasse, die bisher zum Einsatz kam, kann auch erweitert werden. Sie hat folgende Eigenschaften und Methoden:

Exceptions implementieren das `Throwable`-Interface, das ist neu in PHP 7 und das verbindet die Exceptions mit der gleich vorgestellten Error-Klassen, die ebenfalls als Throwable-Interface implementieren

- `$message`
  - ❖ die Meldung, die im Fehlerfall ausgegeben wird
- `$code`
  - ❖ die Fehlernummer der Exception
- `$file`
  - ❖ der Name der Datei, in der die Exception aufgetreten ist
- `$line`
  - ❖ die Nummer der Zeile, in der die Exception aufgetreten ist

```
Exception implements Throwable {
    /** Eigenschaften */
    protected string $message;
    protected int $code;
    protected string $file;
    protected int $line;
}
```

Differenzierte Meldungen

CODERS.BAY

Die in PHP vordefinierte Exception-Klasse, die bisher zum Einsatz kam, kann auch erweitert werden. Sie hat folgende Eigenschaften und Methoden:

- Die Konstruktormethode. Der Standardfehlertext ist ein leerer String und die Standardfehlernummer 0.
- `getMessage()` liefert die Fehlermeldung
- `getPrevious()` liefert die vorherige Exception
- `getCode()` liefert die Fehlernummer
- `getFile()` liefert die Datei, in der der Fehler aufgetreten ist
- `getLine()` liefert die Zeile
- `getTrace()` liefert den Stacktrace, das heißt Infos über Datei, Zeile und eventuell Funktion usw. als Array
- `getTraceAsString()` liefert dieselben Infos wie `getTrace()`, aber direkt als String zur Ausgabe

```
Exception implements Throwable {
    /** Methoden */
    public __construct ([
        string $ message = "" [,int $ code = 0 [,Throwable $ pr
evious = NULL ]]
    ])
    final public string getMessage (void)
    final public Throwable getPrevious (void)
    final public mixed getCode (void)
    final public string getFile (void)
    final public int getLine (void)
    final public array getTrace (void)
    final public string getTranceAsString (void)
    public string __toString (void)
    final private void __clone (void)
}
```

Error-Klasse

Bestimmte Fehler kann man seit PHP 7 abfangen. Dazu zählt beispielsweise der Aufruf einer nicht definierten Funktion.

Im Beispiel gibt es eine Klasse mit einer Konstruktor- und einer Destruktor-Methode. Innerhalb eines `try`-Blocks wird ein Objekt der Klasse erstellt. Problematisch ist die angegebene Funktion `problem()`, da sie nicht existiert.

Es folgen zwei `catch`-Blöcke: Im ersten werden Exceptions abgefangen, im zweiten Fehler (Error). Außerdem gibt es einen `finally`-Block.

An der Ausgabe sieht man, dass alles geklapt hat: Der Fehler konnte abgefangen werden, und auch die Destruktor- und die `finally`-Methoden werden ausgeführt - Letzteres wäre in Version vor PHP 7 nicht der Fall.

```php
class Ressource {
    public function __construct() {
        echo "Ressourcen nutzen<br>";
    }
    public function __destruct() {
        echo "Ressourcen freigeben<br>";
    }
}
try {
    $ressource = new Ressource();
    problem();
} catch (Exception $e) {
    echo $e->getMessage() . "<br>\n";
} catch (Error $e) {
    echo $e->getMessage() . "<br>\n";
} finally {
    echo "Finally aufgerufen<br>";
}
```

Error-Klasse

Da Error und Exception beide als `Throwable`-Interface implementieren, kannst du beide Arten von Fehlern/Probleme auch auf die folgende Art abfragen:

```php
try {
    $ressource = new Ressource();
    problem();
} catch (Throwable $e) {
    echo $e->getMessage() . "<br>\n";
} finally {
    echo "Finally aufgerufen<br>";
}
```

Es gibt unter anderem folgende Fehlertypen:

- `ArgumentCountError`
  an eine Funktion oder Methode werden zu wenig Argumente übergeben
- `ArithmeticError`
  Fehler bei mathematischen Operationen
- `ParseError`
  Wenn ein Parse-Fehler auftritt, beispielsweise beim Aufruf von `eval()`
- `TypeError`
  Wenn der von einer Funktion/Methode erwarteter Datentyp nicht korrekt ist

Generatoren

Es gibt viele in PHP vordefinierte Klassen und Schnittstellen, wie beispielsweise das Iterator-Interface, eine Schnittstelle für selbstiterierende Objekte. Wenn du diese nutzen willst, musst du jedoch eine große Anzahl an Methoden implementieren, was relativ aufwendig ist. Einfacher geht das mit Generatoren. Um eine Generator-Funktion zu erstellen, genügt es, das `yield`-Schlüsselwort anzugeben: Jede Funktion, die `yield` beinhaltet, ist automatisch eine Generator-Funktion

Die Funktion `generator_beispiel()` erzeugt einen Generator mit den Zahlen von 1 bis 6

```php
function generator_beispiel() {
    for ($i = 1; $i <= 6; $i++) {
        yield $i;
    }
}
```

Dann wird diese Funktion einmal testweise an `print_r()` übergeben und dann mit `foreach` durchlaufen

```php
print_r (generator_beispiel());
foreach (generator_beispiel() as $wert) {
    echo "$wert ";
}
```

Generatoren

In PHP 7 gibt es zwei Neuerungen bei Generatoren: Sie können innerhalb eines Generators einen Rückgabewert mit `return` angeben, auf den du dann mit `getReturn()` zugreifst. Außerdem kannst du einen Generator über `yield from` in einem zweiten nutzen.

Warum aber erstellt man nicht einfach ein Array und durchläuft dieses wie gewohnt? Bei großen Mengen an Daten gibt es einen großen Performance-Unterschied zwischen der normalen Array-Erstellung und dem Einsatz von Generatoren: Letztere sind wesentlich performanter.

Fazit: Generatoren solltest du wählen, wenn du eine große dynamisch erstellte Liste durchlaufen willst.

Überblick

Überblick über die bei der objektorientierten Programmierung benutzten Schlüsselwörter

Es gibt viele Schlüsselwörter, Operatoren und Methoden, die bei der objektorientierten Programmierung relevant sind. Die folgende Tabelle listet sie alphabetisch auf - mit kurzer Erklärung.

CODERS.BAY

Schlüsselwort	Funktion
`->`	Zugriff auf Eigenschaften und Methoden
`::`	Doppel-Doppelpunkt-Operator, auch Gültigkeitsbereichsoperator oder Paamayim Nekudotayim genannt
`abstract`	Eine Methode, die als `abstract` deklariert wird, muss in einer abgeleiteten Klasse überschrieben werden
`__call()`	Magische Methode, die automatisch aufgerufen wird, wenn eine in einer Klasse nicht definierten Methode eingesetzt wird
`__callStatic()`	Magische Methode, die automatisch aufgerufen wird, wenn eine in einer Klasse nicht definierte Methode eingesetzt wird
`clone`	Erstellt eine exakte Kopie eines Objekts, die dann unabhängig vom ursprünglichen Objekt verändert werden kann
`__clone()`	Magische Methode, die automatisch aufgerufen wird, wenn ein Objekt geklont wird
`const`	Dient zur Definition von Konstanten in Klassen
`__construct()`	Methode, die automatisch aufgerufen wird, wenn ein neues Objekt erstellt wird

Schlüsselwort	Funktion
`__destruct()`	Methode, die automatisch aufgerufen wird, wenn ein neues Objekt zerstört wird
`extends`	Relevant bei der Vererbung zur Ableitung einer Klasse von der Basisklasse
`final`	Eine als `final` definierte Methode kann in einer abgeleiteten Klasse nicht überschrieben werden
`__get()`	Magische Methode, die automatisch aufgerufen wird, wenn versucht wird, eine nicht definierte Eigenschaft auszulesen
`implements`	Eine Schnittstelle wird über `implements` implementiert, das heißt die Methodendefinitionen mit Inhalten gefüllt
`insteadof`	Kann bei Traits benutzt werden, um anzugeben, welche der gleichnamigen Methoden verwendet werden soll
`interface`	Eine Schnittstelle enthält nur öffentliche Eigenschaften, Konstanten und Methoden, wobei die Methoden noch nicht implementiert sind
`namespace`	Dient zur Definition eines Namensraums
`parent`	Verweis auf die Elternklasse

Überblick

CODERS.BAY

Schlüsselwort	Funktion
`private`	Regelt den Zugriff auf Eigenschaften und Methoden: Auf als `private` gekennzeichnete Methoden/Eigenschaften kann nur innerhalb der Klasse selbst zugegriffen werden, nicht in einer abgeleiteten Klasse oder über ein Objekt
`protected`	Regelt den Zugriff auf Eigenschaften und Methoden: Auf als `protected` gekennzeichnete Methoden/Eigenschaften kann innerhalb der Klasse selbst und in einer abgeleiteten Klasse zugegriffen werden, nicht über ein Objekt
`public`	Regelt den Zugriff auf Eigenschaften und Methoden: Auf als `public` gekennzeichnete Methoden/Eigenschaften kann von überall zugegriffen werden; sie sind öffentlich
`self`	Verweis auf die Klasse selbst
`__set()`	Magische Methode, die automatisch aufgerufen wird, wenn versucht wird, eine nicht definierte Eigenschaft zu setzen

Schlüsselwort	Funktion
`static`	Kann bei Eigenschaften oder Methoden eingesetzt werden. Diese können dann direkt ohne Initialisierung eines neuen Objekts aufgerufen werden
`static`	`static` kann anstelle von `self` zum Zugriff auf statische Methoden innerhalb einer Klassendefinition benutzt werden. Es wird später, das heißt nicht zur Kompilierzeit, sondern zur Laufzeit durch den Namen der aufrufenden Klasse ersetzt (Late Static Binding)
`$this`	Verweis auf das Objekt selbst
`__toString()`	Magische Methode, die automatisch aufgerufen wird, wenn `print` oder `echo` bei einem Objekt benutzt wird. Damit kann man bestimmen, welche Information dann ausgegeben werden soll
`trait`	Erlaubt die Codewiederverwendung
`use`	Dient dem Import von Namensräumen in den aktuellen Kontext

Quelle

Maurice, Florence. PHP 7 und MySQL: Ihr praktischer Einstieg in die Programmierung dynamischer Websites (German Edition). dpunkt.verlag. Kindle-Version.