Westfälische Wilhelms-Universität Münster

Erstellen dynamischer Webseiten mit PHP

WS 2016/2017

wissen.leben
WWU Münster

Dozenten: Patrick Förster, Michael Hasseler

Übersicht der heutigen Inhalte

- Type-Hinting
- Abstrakte Klassen
- Interfaces

Wiederholung: Vererbung

- Eine Klasse „erbt“ von einer anderen Klasse durch das Schlüsselwort **`extends`**

```
class Person {…}
```

```
class Student extends Person {…}
```

- Eine erweiternde Klasse erbt alle Attribute sowie Methoden#
- Methoden und Attribute können überschrieben werden
- **Vererbungsregel**: ist eine Methode/Eigenschaft in der aufgerufenen Klasse nicht vorhanden wird in der Elternklasse danach gesucht und dann in deren usw. bis zum ersten Auffinden der Methode/Eigenschaft

- Zugriff auf Attribute/Methoden der Oberklasse erfolgt über den Gültigkeitsbereich **`parent`**

```
parent::__construct($vorname, $nachname, $geburtsdatum);        // Konstruktor in Student
```

- Mit dem Sichtbarkeitsmodifikator **`protected`** können Methoden/Attribute vor Zugriff von außen geschützt aber für erbende Klasse zugänglich gemacht werden

Nachtrag Vorlesung 11: Methoden überschreiben

- Der Konstruktor einer Klasse kann in erbenden Klassen überschrieben und erweitert werden:

```
public function __construct($vorname, $nachname, $geburtsdatum) {…}
```

```
public function __construct($vorname, $nachname, $geburtsdatum, $matrikelnr, $studienfach)
{ … }
```

- Dies gilt nicht für andere Methoden

```
public function setze($vorname, $nachname, $geburtsdatum) {…}
```

```
public function setze($vorname, $nachname, $geburtsdatum, $matrikelnr, $studienfach) {…}
```

```
Warning: Declaration of Student::setze($vorname, $nachname, $geburtsdatum, $matrikelnr,
$studienfach) should be compatible with Person::setze($vorname, $nachname, $geburtsdatum)
```

- Neue Parameter müssen einen Default-Wert zugewiesen bekommen (Vgl. Vorlesung 5.10)

```
public function setze($vorname, $nachname, $geburtsdatum,
  $matrikelnr = 0, $studienfach = "") {…}
```

Klassen und Typen

- In PHP sind Variablen immer implizit typisiert

```
$var1 = "Das ist ein String";  ➔ gettype($var1);    ➔ string
$var2 = 10;                    ➔ gettype($var2);    ➔ integer
```

- Variablentypen sind nicht festgelegt

```
$var1 = $var2;
echo gettype($var1);                    // ➔ integer
```

- Der Typ einer Instanz-Variablen ist nach **`gettype(…)`** „*object*“
- Die Klasse der Instanz, auf die eine Variable verweist, erhält man mit **`get_class(…)`**
- Die „ist ein“ Beziehung zwischen Elternklasse und Subklasse, lässt sich mit dem *Operator* **`instanceof`** überprüfen:

```
$student = new Student(…);
$person = new Person(…);
```

```
$student instanceof Person    ➔ true
$person  instanceof Student   ➔ false
```

Type-Hinting (b01)

- Die implizite Typisierung kann schnell zu Typ-Fehler führen

```
function istEingeschrieben($student, $fach) {
    return $student->getStudienfach() === $fach;
}
```

```
$object = new Student(…);
istEingeschrieben($object, "Informatik");

// … irgendwann irgendwo anders im Code wird $object neuzugewiesen
$object = new Person(…);

// … und später
istEingeschrieben($object, "Informatik");
```

- Obiges ist valider Code führt aber unweigerlich zu

```
Fatal error: Call to undefined method Person::getStudienfach()
```

Type-Hinting: Klassen (b02)

- Bei der Einführung von Setter-Methoden in Vorlesung 11 wurde „Typsicherheit“ in der Methode durch Überprüfung der Eingabe garantiert:

```
public function setVorname($name) {
  if (!is_string($name)) throw new Exception(…);
  $this->vorname = $name;
}
```

- Mit PHP 5 wurde „*Type Hinting*“ eingeführt, durch das Parameter auf bestimmte Klassen eingeschränkt werden können:

```
function istEingeschrieben(Student $student, $fach) {…}
```

- Mit der Einschränkung **`Student`** vor **`$student`** wird garantiert, dass der Parameter im Funktionsrumpf definitiv eine Instanz der Klasse **`Student`** ist
- Wird **`istEingeschrieben`** mit einer Instanz aufgerufen, die nicht von **`Student`** erbt, so wird das Programm noch vor Ausführung der Methode abgebrochen

```
Catchable fatal error: Argument 1 passed to istEingeschrieben must be an
instance of Student, [ÜBERGEBENE KLASSE] given…
```

Type-Hinting

- Type-Hinting ist eingeschränkt auf
  - Klassen
  - Arrays
  - „Callable“

- Damit lässt sich die Typsicherheit für **`setVorname`** *nicht* auf Type-Hinting reduzieren

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

```
$person->setVorname("Moritz");
```

```
Catchable fatal error: Argument 1 passed to Person::setVorname() must be an instance of string, string given…
```

- Es wurde erwartet, dass die Angabe vor **`$name`** eine Klasse ist, allerdings ist **`string`** keine Klasse (ähnliches gilt für alle „primitiven“ Datentypen, wie **`integer`**, ect.)!

Type-Hinting: Array (b02)

- Mit **array** kann definiert werden, dass ein Parameterwert ein Array sein muss
- Achtung: Der Inhalt des Arrays muss ggf. auf Typsicherheit überprüft werden!

```
class Vorlesung {
  private $dozenten = array();
  …

  public function setDozenten(array $dozenten) {
    $this->dozenten = [];

    // Dozent über die fuegeDozentHinzu-Methode hinzufügen
    foreach ($dozenten as $dozent) {
      $this->fuegeDozentHinzu($dozent);
    }
  }

  public function fuegeDozentHinzu(Dozent $dozent) {…}

}
```

Type-Hinting: Callable (b02)

- Als **`Callable`** werden Funktionen oder Methoden bezeichnet
- Vgl. Vorlesung 5: „Funktionen als Werte“

```
function test($errechnet, $erwartet, callable $callback) {
  if ($callback($errechnet, $erwartet))
    ok(“($errechnet) $callback $erwartet");
  else
    fehler("nicht (($errechnet) $callback $erwartet)");
}

function ist_ein($a, $b) {
  return $a instanceof $b;
}

test($dozent1, "Person", "ist_ein");
test($dozent1, "Student", "ist_ein");
```

```
(Person … => Dozent: Informatik) ist_ein Person
nicht ((Person …=> Dozent: Informatik) ist_ein Student)
```

Abstrakte Klassen

- Das Verhalten einer Klasse wird durch ihre Methoden definiert
- In PHP können Klassen definiert werden, ohne deren Verhalten gänzlich zu spezifizieren
- Konkret heißt dies, dass Methoden zwar vorgegeben aber nicht implementiert werden:
  - Eine solche Methode wird mit dem Modifiktator **`abstract`** definiert
  - Existiert auch nur eine abstrakte Methode, so muss auch die Klasse **`abstract`** sein

```
abstract class Prozess {

  protected abstract function verarbeiten($daten);

}
```

- Die Methode **`verarbeiten`** wird nicht implementiert, sondern überlässt dies allen erbenden Klassen
- Abstrakte Klassen können weiterhin auch „normale“ Methoden definieren
- Ebenso können Attribute definiert werden

Abstrakte Klassen II (b03)

- Eine abstrakte Klasse wird immer dann eingesetzt, wenn
  - Ein Teilverhalten allgemein genug ist, um es für alle erbenden zu implementieren
  - Ein anderer Teil zu wage, um auch nur ein Basisverhalten zu spezifizieren

```
abstract class Prozess {
  private $daten;

  protected abstract function initialiseren();    // Diese Methoden
  protected abstract function verarbeiten();      // können nicht allgemein-
  protected abstract function aufrauemen();       // gültig implementiert

  public final function ausfuehren($daten) {      // Diese Methode legt
    $this->$daten = $daten;                       // konkret und für alle
    $this->initialiseren();                       // Prozesse fest, dass diese
    $this->verarbeiten();                         // in drei Phasen
    $this->aufrauemen();                          // aufgeteilt ausgeführt
    $this->$daten = null;                         // werden
  }
}
```

Abstrakte Klassen III

- Da das Verhalten von Prozess noch nicht gänzlich implementiert ist, kann keine Instanz der Klasse erstellt werden:

```
$process = new Prozess();
```

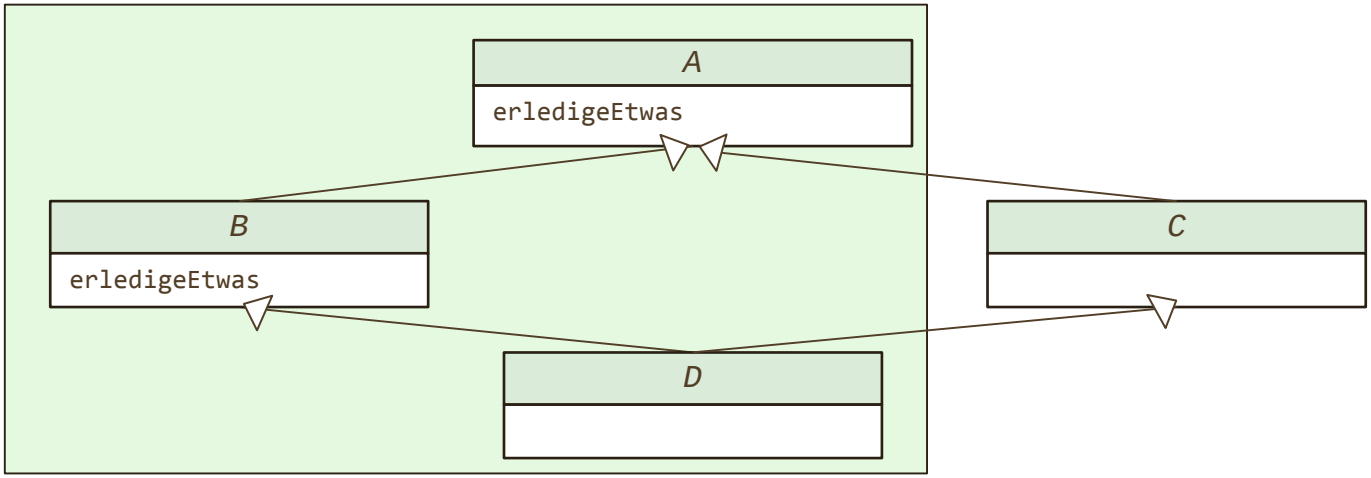
```
➔ Fatal error: Cannot instantiate abstract class Prozess
```

- Abstrakte Klassen werden wie gewohnt vererbt:

```
class EinleseProzess extends Prozess {…}
```

- Einen erbende Klasse muss entweder:
  - Alle abstrakten Methoden implementieren
  - Oder ist selbst wieder **`abstract`**
- Eine abstrakte Methode kann nicht **`private`** sein
- Eine als **`protected`** definierte abstrakte Methode kann als **`public`** implementiert werden

Mehrfachvererbung




```
$d = new D();
$d->erledigeEtwas()
```

- Welche Methode wird aufgerufen **`B::erledigeEtwas`** oder **`A::erledigeEtwas`**?
- Das Problem ist nicht trivial lösbar (Diamantproblem), PHP unterstützt daher keine konkrete Mehrfachvererbung
- Es gibt aber die Möglichkeit eine Art „Pseudo-Vererbung“ zu definieren

Interfaces

- Ein Interface ist eine abstrakte Klasse, in der alle Methoden
  - **`public`** sind (per default, die Sichtbarkeit muss nicht angegeben werden)
  - **`abstract`** sind (per default, **`abstract`** wird nicht angegeben)
- In einem Interface können zudem keine Instanz-Attribute festgelegt werden

- Ein Interface wird mit dem Schlüsselwort **`interface`** (anstelle von **`class`**) definiert

```
interface Lautstark {
    function gibLaut();
}
```

- Eine Klasse „implementiert“ ein Interface mit Hilfe des Schlüsselwortes **`implements`**

```
class Hund implements Lautstark {
  public function gibLaut() {echo "WUFF";}
}
```

- Die implementierende Klasse muss entweder alle im Interface definierten Methoden implementieren oder **`abstract`** sein

Interfaces

- Ein Interface kann ein anderes Interface mittels **`extends`** erweitern:

```php
interface Regelbar extends Lautstark {
  function lauter($vol=1);
  function leiser($vol=1);
}
```

```php
class Box implements Regelbar {
  public function lauter($vol=1) {…}
  public function leiser($vol=1) {…}
  …
  public function gibLaut() {…}
}
```

- Eine Klasse kann mehrere Interfaces implementieren

```php
interface Flugfaehig {
  function fliege();
  function lande();
}
```

```php
class Vogel implements Lautstark, Flugfaehig {
  private $fliegt = false;
  // Interface Lautstark:
  public function gibLaut() {echo "PIEP";}
  // Interface Flugfaehig:
  public function fliege() {$this->fliegt = true;}
  public function lande() {$this->fliegt = false;}
}
```

Interfaces

- Auch das Diamant-Problem stellt kein Problem mehr dar:

```php
interface Lautstark {
    function gibLaut();
}
```

```php
interface Musikalisch {
    function gibLaut();
}
```

```php
class Saenger implements Lautstark, Musikalisch {…}
```

- Beide Interfaces geben die Methode **`gibLaut`** vor
- Allerdings bestimmt keines das Verhalten der Methode

- Die Klasse **`Saenger`** muss für beide Interface nur eine Methode definieren
- Das Verhalten dieser Methode bestimmt die Klasse selbst

- Es stellt sich allerdings die Frage, wie die Klasse beiden Interfaces gerecht werden kann
- Dies ist allerdings ein Problem des Programmierers
- Im Zweifel muss die Doppeldeutigkeit durch Weglassen eines der Interfaces gelöst werden

Interfaces (b04)

- Mit Interfaces erhält man eine Art Pseudo-Mehrfachvererbung, denn

```
class Vogel implements Lautstark, Flugfaehig {…}
```

  - **Vogel** ist sowohl eine Instanz von **Lautstark** als auch von **Flugfaehig**

```
class Rekorder {
  private $geraeusche = [];
  public function nehmeauf(Lautstark $l) {
    $this->geraeusche[] = $l->gibLaut();
  }
}
```

```
class Schwarm {
  private $objekte = [];
  public function fuegeHinzu(Flugfaehig $f) {
    $this->objekte[] = $f;
  }
}
```

```
$vogel     = new Vogel();

$rekorder  = new Rekorder();
$scharm    = new Schwarm();



// $vogel ist “Lautstark2
$rekorder->nehmeauf($vogel)

// $vogel ist “Flugfaehig”
$schwarm->fuegeHinzu($vogel)
```

Interface-Einschränkungen

- Bei der Erweiterung eines Interfaces kann eine geerbte Methode *nicht* neudefiniert werden

```php
interface Lautstark {

    function gibLaut();

}
```

```php
interface Regelbar extends Lautstark {

  function gibLaut($lautstaerke);

  function lauter($vol = 1);
  function leiser($vol = 1);
}
```

- Wenn eine Klasse zwei Interfaces implementieren soll, die namensgleiche Methoden haben, so müssen diese auch Parameter gleich sein

```php
interface Lautstark {
    function gibLaut();
}
```

```php
interface Musikalisch {
    function gibLaut($tonart);
}
```

```php
class Saenger implements Lautstark, Musikalisch {…}
```

Polymorhpie (b05)

```
class Orchester {
  private $instrumente = [];              // Inhalt vom Typ “Instrument”
  …
  public function performen($minuten) {
    for ($i = 0; $i < $minuten; $i++) {
      $instrument = $this->instrumente[rand(0, count($this->instrumente) - 1)];

      echo $instrument->gibLaut() . NL;
    }
  }
}
```

- **`$instrument`** wird per Zufall zugewiesen
- Implementierung von **`gibLaut`** kann erst zur Laufzeit festgestellt werden
- Dieses Verhalten nennt man ***spätes Binden*** (auch „*dynamisches Binden*“)
- Die Methode **`gibLaut`** wird ***polymorph*** genannt, da sie abhängig von der implementierenden Klasse unterschiedliches Verhalten definiert
- Die Variable **`$instrument`** wird ebenfalls als ***polymorph*** bezeichnet, da sie zu auf verschiedene Instanzen ableitend von der gleichen *Basisklasse* (**`Lautstark`**) verweist

Duck-Typing (b05)

- Häufig wird Typisierung genutzt, um ein bestimmtes Verhalten zu garantieren
- Im Orchester-Beispiel: Alle Objekte in **`$instrumente`** erweitern die Klasse `Instrument`

- Ein anderer Ansatz ist das sogenannte ***Duck-Typing***
  - Nicht die Klasse, sondern einzig das Vorhandensein einer Methode bzw. eines Attributes qualifiziert eine Instanz
  - *„Wenn ich einen Vogel sehe, der wie eine Ente läuft, wie eine Ente schwimmt und wie eine Ente schnattert, dann nenne ich diesen Vogel eine Ente.“* - James Whitcomb Riley
  - Ob eine Instanz eine Methode anbietet, kann per **`method_exists`** abgefragt werden

```
class Jam {
  private $klangvoll = [];
  …
  public function fuegeHinzu($klang) {
    if (!method_exists($klang, "gibLaut")) throw new Exception(…);
    $this->klangvoll[] = $klang;
  }
}
```

Zusammenfassung

- Wozu dient Type-Hinting?
- Welche Einschränkungen hat Type-Hinting?
- Was unterscheidet eine abstrakte von einer nicht-abstrakten Klasse?
- Was unterscheidet ein Interface von einer abstrakten Klasse?
- Wozu dienen Interfaces?

Aufgaben

1. Gegen sei folgendes Interface

```php
interface Form {
  public function getUmfang();
  public function getFlaeche();
}
```

   Implementieren Sie dieses Interface durch mindestens zwei Klasse, bspw.

   a. Rechteck
   b. Kreis

2. Betrachten Sie die Klassen **`Orchester`** und **`Jam`** aus Bespiel 05. Extrahieren Sie die abstrakte Klasse `Ensemble` durch Analyse der beiden Klassen.

Anhang: ZIVinteraktiv-Abstimmung

- QR-Code zur anonymen Abstimmung:

- https://www.uni-muenster.de/ZIV/anw/zivinteraktiv/abstimmung.php?code=561fd9fa