# Allgemeine Konzepte der OOP

Die objektorientierte Programmierung (OOP) ist eine seit vielen Jahren eingeführte Art und Weise des Programmierens, die sich wesentlich von der sog. strukturierten Programmierung unterscheidet.

C:\Users\C\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\4DDEELXO\MC900351929[1].wmfDie strukt. Programmierung (auch prozedurale Programmierung genannt) trennt zwischen den Daten und den Methoden, die diese Daten manipulieren. Die objektorientierte Programmierung führt diese beiden Aspekte wieder in Form von Klassen zusammen. Die Klasse ist der Container für Daten, die logisch gesehen zusammengehören und einen realen/irrealen Gegenstand unseres Systems abbilden. Sie beinhaltet auch sämtliche Methoden, die diese Daten manipulieren sollen.

**Wichtige Konzepte**

|  |  |
| --- | --- |
| * Klasse / Objekt / Instanziierung * Attribute / Methoden * Konstruktor/Destruktor * Statische Attribute und Methoden * Geheimnisprinzip | * Einfach-, Mehrfachvererbung * Abstrakte Klasse / Interface * Überschreiben von Methoden/Polymorphie * Assoziation/Aggregation/Komposition |

### Klasse / Objekt /Instanziierung

Die Klasse ist der grundlegende Begriff in der OOP.

Ausgehend von der jeweiligen Problemstellung versucht man, die zusammengehörenden Informationen in einem gemeinsamen Container, der Klasse eben, zu speichern.

Die Klasse erhält einen Namen, die zu speichernden Informationen bezeichnet man als Attribute der Klasse.

Die Manipulation, d.h. das Schreiben, Lesen und Ändern der Daten, obliegt ebenfalls dem Verantwortungsbereich der Klasse. Neben diesen Aufgaben kann eine Klasse noch weitere Fähigkeiten besitzen. Diese Fähigkeiten werden in der Klasse durch Methoden definiert.

Eine Klasse besteht deshalb zumindest aus 3 Bereichen.

1. Dem Klassennamen
2. Der Liste der Attribute
3. Der Liste der Methoden

Ein typisches Beispiel für eine Klasse könnte wie folgt aussehen:

|  |  |  |
| --- | --- | --- |
| Quellcode | Klassendiagramm | |
| class Person{  private $Alter  private $Name  private $Vorname  public function getName():string  {  return $this->Name  }  public function setName(string $VorName)  {  $this->Name = $VorName  }  Public function sprechen(string $Text)  {  *// HAs to be done#>*  }  } | |  |

Neben den wirklichen Fähigkeiten, im obigen Beispiel sprechen, muss eine Klasse häufig über Verwaltungsmethoden verfügen, die eine Manipulation der internen Attribute ermöglichen. Man nennt diese Methode häufig getter/setter-Methoden. Da ein Methodenname lediglich einmal genutzt werden darf, muss man deshalb häufig 2 Methodennamen benutzen.

Innerhalb der Klasse gibt es häufig eine spezielle Methode, den sog. Konstruktor. Sie hat in vielen Programmiersprachen den gleichen Namen wie die Klasse selbst und sie kann in verschiedenen Variationen vorliegen. PHP benutzt dafür die Funktion **\_\_construct()**. Beim Erzeugen eines Objektes (s.u.) wird diese Methode als Erstes aufgerufen. Wird diese Methode nicht innerhalb der Klasse definiert, so wird ein sog. Standardkonstruktor benutzt. Er besteht lediglich aus dem Methodennamen ohne irgendwelche Parameter. Werden eigene Konstrukoren geschrieben, so muss der Standardkonstruktor explizit definiert werden, sonst ist er nicht mehr vorhanden.

Hinweis: PHP kennt keine Mehrfachkonstruktoren. Man kann die damit verbundenen Absichten aber mimiken. Siehe dazu <https://stackoverflow.com/questions/1699796/best-way-to-do-multiple-constructors-in-php> bzw. <https://www.kerstner.at/2015/03/overloading-constructors-and-functions-in-php/> bzw. <http://verraes.net/2014/06/named-constructors-in-php/>

|  |  |
| --- | --- |
| Quellcode | Klassendiagramm |
| class Person{  private $Alter  private $Name  private $Vorname  public function \_\_construct (){  }  public function \_\_construct (string $FamName, string $Vorname, $Alter)  {  $this->Alter = $Alter  $this->Vorname = $Vorname  $this->Name = $FamName  }  Public function getName():string  {  return $this->Name  }  public function setName(string $VorName){  $this->Name = $VorName  }  public function sprechen(string $Text){  *<# HAs to be done#>*  }  } |  |

### Benutzung von Objekten / Konstruktor

Die Klasse definiert als Bauplan lediglich die Art der gehaltenen Informationen bzw. die Methoden, die zur Manipulation der Daten bzw. zur Funktionalität der Klasse notwendig sind. Doch wie kann man nun diese Klasse nutzen? Da wir ja Informationen zu einem realen Gegenstand des Systems erheben wollen, müssen wir den Bauplan der Klasse einem realen Objekt zuordnen. Nur ein reales Objekt kann Daten speichern. Dieser Vorgang wird Instanziierung genannt und läuft in folgenden Schritten ab:

* Deklaration einer Variablen
* Instanziierung des Objektes mit Hilfe des new-Operators in Verbindung mit dem Klassennamen.

$karl = new Person()

$gerd = new Person("Gerd", "Sych", 76)

echo($gerd->Alter);

echo($karl->Vorname);

Der new-Operator erzeugt ein sog. Objekt der Klasse. Dieses Objekt ist einzigartig und ist nun in der Lage, die Informationen aufzunehmen und zu verarbeiten. Bei allen Instanziierungen wird immer ein Konstruktor aufgerufen. Er lautet wie der Name der Klasse und kann überladen sein, d.h. in verschiedenen Versionen existieren. Durch ihn ist das Objekt in der Lage, Zustände seiner Variablen bei der Erzeugung zu kontrollieren. Definiert man eigene Konstruktoren, so muss der parameterlose Standardkonstruktor ebenfalls angegeben werden, wenn man ihn zur Verfügung stellen will.

Erst nach dem Erzeugen kann man nun die Fähigkeiten des Objektes benutzen, d.h. man kann die Methoden der Klasse benutzen. Merke: Methoden werden auf Klassenebene definiert, aber auf Objektebene genutzt ! (Es gibt aber eine Ausnahme ! Welche ?)

Parallel zum Konstruktor gibt es den sog. Destruktor. Er wird durch den folgenden Aufruf beschrieben:

~Klassenname() PHP: function \_\_destruct()

Das Zerstören eines Objektes wird durch das NULL-Setzen der Objektreferenz bewirkt. Der tatsächliche Zeitpunkt des Zerstören eines Objektes hängt von mehreren Faktoren ab

* Anzahl der noch gültigen Referenzen
* Tatsächliches Freigeben des Speichers durch die Garbage Collection

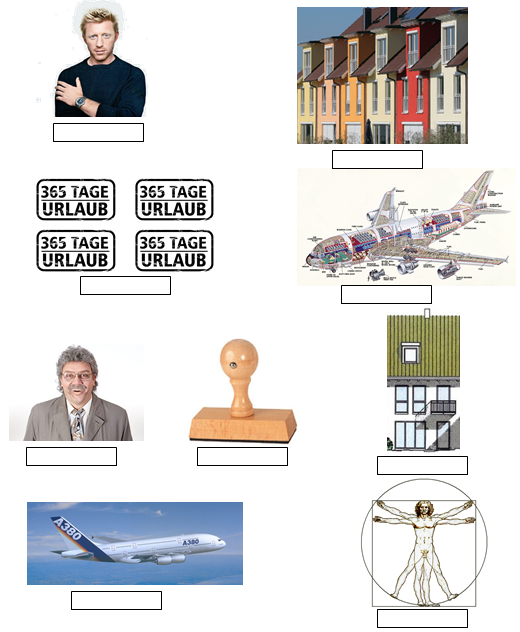
Es kann deshalb nicht genau vordefiniert werden, ob und wann durch das Löschen einer Objektreferenz die Destruktor-Methode aufgerufen wird.

Siehe dazu auch: <https://stackoverflow.com/questions/2777942/php-destructor-behaviour>

Beispiel:

### Aufgabe:

Weisen Sie den untenstehenden Bildern jeweils die Begriffe Klasse bzw. Objekt zu



**C:\Users\C\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.IE5\4DDEELXO\MC900351929[1].wmfAufgabe:** Überlegen Sie sich, welche Klassen es in Ihrer beliebten Berufsschule gibt.

Finden Sie auch einige Attribute und Methoden

### Statische Attribute/Methoden

Um in OOP arbeiten zu können, sind offenbar immer zunächst Objektinstanziierungen notwendig. Dies ist aber manchmal lästig, weil wir eigentlich nur eine Funktionalität brauchen bzw. redundante Informationen speichern wollen, die für alle gleich sind.

So könnte es z.B. sein, dass Schüler einer Klasse die Anzahl der Mitschüler in ihrer Klasse kennen sollen. Bei 32 Schülern würde dies bedeuten, dass 32 Objekte die gleiche Information in einer lokalen Instanzvariable halten müssten! Noch schlimmer, bei Änderungen der Schülerzahl durch Hinzukommen / Weggehen neuer Schüler müsste in allen Objekten dieser Wert geändert werden !

Für solche Fälle kennt die OOP die Möglichkeit sog. statischer Attribute bzw. Methoden. Diese werden bei der Klasse gehalten und durch das Schlüsselwort static deklariert.

Der Zugriff auf diese Werte ist sowohl über jedes Objekt als auch über die Klasse an sich möglich.

Die Aktualisierung solcher Informationen beim Zerstören solcher Objekte kann aber problematisch werden. Folgendes Beispiel verdeutlicht die Situation:

<?php

class Person{

private $Alter;

private $Name;

private $Vorname;

static $AnzahlSchueler=0;

public function \_\_construct1(){

}

public function \_\_construct(string $FamName, $Vorname, int $Alter)

{

$this->Alter = $Alter;

$this->Vorname = $Vorname;

$this->Name = $FamName;

self::$AnzahlSchueler++;

}

public function getName():string

{

return $this->Name;

}

public function sprechen(string $Text)

{

echo( $Text);

}

}

$Karl = new Person("Steinam", "Karl", 20);

echo(Person::$AnzahlSchueler);

echo($Karl::$AnzahlSchueler);

$Gerd = new Person("Sych", "Gerd", 32);

echo(Person::$AnzahlSchueler);

Person::$AnzahlSchueler = 32;

echo($Gerd::$AnzahlSchueler);

$Gerd::$AnzahlSchueler = 4;

echo(Person::$AnzahlSchueler);

echo($Karl::$AnzahlSchueler);

### Geheimnisprinzip

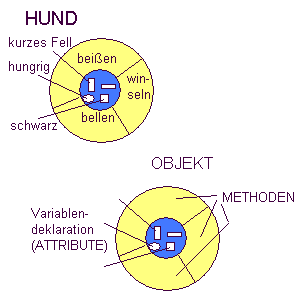
So wie Objekte im realen Leben auch nicht jedes Geheimnis nach draußen preisgeben, so gilt dies auch in der OOP. Das Objekt sollte prinzipiell den Zustand seiner Attribute versteckt halten, d.h. keinen direkten Zugriff auf seine Attribute erlauben.

Durch das Bereitstellen von entsprechenden getter/setter-Methoden bzw. Properties kann das Objekt den Zugriff auf seine Attribute kontrollieren.

Der generelle Zugriffsmöglichkeit auf Attribute und Methoden wird in der OOP über die Sichtbar-keiten definiert. Diese stehen vor dem Attribut bzw. der Methode und definieren den möglichen Zugriff von außerhalb auf die Attribute und Methoden. Falls keine Aussage im Quellcode getroffen wird, sol gelten die default-Sichtbarkeiten der jeweiligen Programmiersprachen, die sich durchaus unterscheiden können. Gängige Definitionen von Sichtbarkeiten sind:

* **public**: Öffentlich sichtbar, von überall aufruf- und damit manipulierbar
* **private**: Nur innerhalb des Objektes selbst benutzbar
* **protected**/internal: Nur innerhalb des gleichen Namespaces bzw. über Vererbungsmechanismen sichtbar.

…..

In den Bildern sind die Variablen des Objektes im Zentrum, umschlossen von den Methoden. Dies deutet grafisch an, daß Variablen gegen Zugriffe von außen geschützt sind, d.h. Programme können nicht direkt auf die Variablen des Objektes zugreifen, son-dern müssen die entsprechenden Methoden auf-rufen. Dieser Schutz wird Kapselung genannt. Mit Kapselung werden oft Implementationsdetails versteckt.

Ein Objekt hat eine nach außen definierte Schnitt-stelle, mit der andere Objekte kommunizieren können. Das Objekt kann die privaten Informa-tionen und Methoden ändern, ohne daß andere Objekte betroffen sind. Zudem kann ein Objekt problemlos herumgereicht werden.

**Arbeitsauftrag**

Erstellen Sie zwei Klassen (Auto, Mitarbeiter) zur Verwaltung eines Fuhrparks in einem Unternehmen  
Speichenr Sie sich für das Auto die Attribute (Hersteller, Modell, Leistung, Verbrauch, Baujahr, KilometerStand)  
Die Klasse Auto verfügt über eine Methode fahren(). Diese prüft, ob die Türen geschlossen sind. Falls ja, wird der Motar gestartet, falls nein erhält der Nutzer einen Piepston. Simulieren Sie den Piepston mit Hilfe von [console]::beep(Frequenz(190 - 8500, Dauer (ms))

Mitarbeiter können sich Fahrzeuge für eine Dienstreise ausleihen. Bei der Rückgabe müssen Sie die gefahrenen Kilometer sowie den Kilometerstand des Autos an den Chef melden.

Erstellen Sie zunächst ein Klassendiagramm

Implementieren Sie anschließend den Sachverhalt mit Hilfe von Powershell  
  
Der Mitarbeiter Brenner leiht sich für eine Dienstreise nach Hamburg (700 km einfach) den BMW mit dem Kennzeichen WÜ-MA-777 aus. Der Kilometerstand ist am Ende der Fahrt im Auto zu setzen.  
.

**HighScoreListe**

