OBJECT GEORIËNTEERD PROGRAMMEREN IN PHP

Joren de Wit
Martijn Cornips

OBJECT GEORIËNTEERD PROGRAMMEREN IN PHP

Origineel artikel: OOP Beginnershandleiding (PHP5), http://www.phptuts.nl/view/45/4/
Originele auteur: Joren de Wit, http://www.jorendewit.nl, 2 februari 2010
E-book bewerking: Martijn Cornips, http://www.cornips.nl, 7 oktober 2013

Inhoudsopgave

Inleiding **4**
Deze handleiding 4
Object georiënteerd denken **5**
Besef wat objecten zijn 5
Objecten herkennen 5
Voorbeeld: Datum? Of ook Dag, Maand, Jaar? 6
Foute denkwijze **7**
Elk zelfstandig naamwoord is een object 7
Object georiënteerd programmeren **8**
Aan de slag 8
$this 9
Een bruikbare class, en nu? 9
Meerdere instanties van een class 10
Visibility **11**
Public 11
Private 11
Protected 12
Welke visibility moet je nu gebruiken? 13
Naamgeving **14**
Classes 14
Variabelen 14
Methods 14
Constanten 15
Bestandsnamen 15
Constructor __construct() **16**
Definiëren van een constructor 16
Vereiste properties afdwingen 17
Voorbeeld: HTML tabel **18**
Stap 1: doel bepalen en objecten herkennen 18
Stap 2: herkennen van eigenschappen van objecten 18
Stap 3: bepalen wat de verschillende objecten moeten kunnen 19
Stap 4: de draw() method 20
Stap 5: de procedurele code 21
Conclusie 21

Inheritance 22
Wanneer is extenden toegestaan? 22
De code 22
Methods overschrijven 23
Methods uit de parent class uitvoeren 24
Voorbeeld: HTML tabel 2 (inheritance) 26
Static methods en properties 29
Abstract classes en Interfaces 30
Abstract classes 30
Interfaces 32
Verschil tussen abstract classes en interfaces 34
Magic methods 36
__set() en __get() 36
__isset() en __unset() 38
__call() 39
__toString() 39
Slotwoord en referenties 42
Overige literatuur 43

Inleiding

Welkom in dit e-book over object georiënteerd programmeren in PHP. Hierin wordt aan de hand van voorbeelden een zo duidelijk mogelijke uitleg gegeven van de basis van dit onderwerp.

Wat is object geörienteerd programmeren?

Er zijn op internet vele omschrijvingen van de definitie object georiënteerd programmeren (OOP) te vinden, stuk voor stuk heel uitgebreid en de meesten redelijk vaag. Echter, er is een zeer korte definitie die luidt als volgt:

"Object Oriented Programming is programming which is oriented around objects, thus taking advantage of encapsulation, polymorphism, and inheritance to increase code reuse and decrease code maintenance."

Schrik niet van de waarschijnlijk onbekende termen die nu op je worden afgevuurd, na het lezen van deze handleiding begrijp je deze definitie hopelijk beter. Er zijn echter een aantal belangrijke aspecten aan deze definitie die je even op je moet laten inwerken voordat je met deze handleiding verder gaat. Vrij vertaald zijn dat:

"Object georiënteerd programmeren is een methode die zich richt op objecten (...)" en "(...) met als doel herbruikbaarheid van code te vergroten en benodigd onderhoud aan de code te verkleinen".

Ik zal hier niet verder ingaan op deze twee uitspraken, de betekenis wordt gedurende deze handleiding vanzelf duidelijk. Houd ze echter wel in gedachte, dit is immers hetgeen waar OOP om draait.

Voorkennis

Een gedegen kennis van programmeren in PHP is vereist waarbij de volgende begrippen je zeker niet onbekend in de oren moeten klinken:

- Variabelen
- Loops
- Statements
- Functies

Zo niet, neem dan de literatuur uit het hoofdstuk Overige literatuur door.

Deze handleiding

In deze handleiding is zoveel mogelijk geprobeerd alles in het Nederlands te houden. De programmeertaal is echter volledig in het Engels, dus is besloten om de stukken voorbeeldcode in het Engels te schrijven. Daarnaast zul je ook regelmatig onvertaalde woorden tegenkomen in de teksten omdat de vertaling het verhaal er niet duidelijker op zou maken.

Wat heb je nodig

- Een webserver met PHP5

Object georiënteerd denken

Object georiënteerd programmeren begint bij de denkwijze waarmee je een applicatie benadert. Als je nieuw bent in de wereld van OOP zul je merken dat deze denkwijze (en dus de manier van programmeren) wezenlijk verschilt van hetgeen je tot nu toe gewend was. Dit gedeelte van de handleiding heeft als doel om een besef te creëren van die achterliggende denkwijze en zal dan ook nog weinig regels code bevatten. Desalniettemin is het waarschijnlijk een van de belangrijkste onderdelen, die je als beginner zeker niet over moet slaan.

Dit gedeelte van de handleiding is grotendeels gebaseerd op de bestaande handleiding 'Object georiënteerd denken' geschreven door Erik Duindam. Teksten en voorbeelden uit die handleiding, zul je hier letterlijk terugvinden. Het origineel is te vinden in het hoofdstuk Slotwoord en referenties.

Besef wat objecten zijn

De definitie van OOP geeft het al aan: OOP is een methode die zich richt op objecten. Maar wat zijn objecten nu precies? Wat doen ze? En misschien wel het belangrijkste op dit moment, hoe kan je ze herkennen?

Kijk eens naar het leven. Alles bestaat uit 'dingen', zoals mensen, computers, bomen, water, de lucht, dieren, etc. Die 'dingen' zou je ook objecten kunnen noemen. Jij als mens bent een onafhankelijk object, maar je kunt je wel binnen andere objecten begeven (kantoor) en andere objecten manipuleren (toetsenbord, computer).

Objecten herkennen

Objecten zijn dus losstaande dingen. Objecten hebben bepaalde eigenschappen. Zo zou iemand blauwe ogen kunnen hebben, een lengte van 1.83m en houden van bier. Als die persoon zijn oogkleur zou willen veranderen dan zal dat lichaam dat moeten doen. Een object van buitenaf zou hooguit zijn ogen kunnen maskeren, maar niet daadwerkelijk de kleur veranderen. Misschien dat hij met een bepaalde injectie (=object) wel zijn kleur ogen kan veranderen; dan wordt er dus een object van buitenaf in hem geplaatst waardoor zijn lichaam zelf de kleuren van zijn ogen aanpast. Het spul uit de injectie komt in zijn lichaam dus behoort dan tot zijn lichaam.

Kort gezegd: objecten hebben bepaalde eigenschappen en die eigenschappen kunnen alleen door het object zelf worden gemanipuleerd.

- Objecten definieer je in een class
- Eigenschappen definieer je als variabelen (properties) bovenaan je class
- Gedrag/veranderingen/manipulatie van eigenschappen definieer je als functies (methods) in je class

Als je objecten uit het verhaaltje hierboven gaat vissen dan kom je tot iets van: persoon, oog, bier, lichaam, kleur, injectie, injectiespul. Wat je hier ziet is dat "blauw" een eigenschap van "kleur" is, maar dat "kleur" weer een eigenschap van "oog" is en "oog" weer een eigenschap van "lichaam" en "lichaam" een eigenschap van "persoon". Dus een eigenschap van een object kan op zichzelf ook weer een object zijn. Dat is zelfs heel gebruikelijk, zoals je ziet.

Klinkt ingewikkeld, maar op deze manier werkt het leven ook. Hopelijk zie je nu een beetje in wat een object is. Want het is allemaal veel simpeler dan je denkt. Het is niets technisch.

Voorbeeld: Datum? Of ook Dag, Maand, Jaar?
Het is logisch dat je voor een datum een apart object maakt binnen je systeem, omdat een datum een vrij specifiek 'iets' is. Maar moet je nu ook een Dag, Maand en Jaar object maken?

Je kunt bedenken: heeft een dag, maand of jaar zelf nog specifieke eigenschappen? Op zich wel, want een dag valt binnen de range(1,31), een maand binnen de range(1,12) en elke maand heeft een unieke naam (januari, februari, etc). Maar het zijn wel constante waarden waar verder niet veel spannends mee gebeurt. En de `date()`-functie van PHP kan al erg veel.

Probeer gewoon eens het Datum-object te maken en dan te kijken of je code flexibeler wordt van aparte dag/maand/jaar-objecten of dat het juist een rotzooitje wordt.

Je moet kijken of het nut heeft een laag dieper te gaan, of het nut heeft om nog meer objecten te maken voor sub-onderdelen. Als je in je Datum-object allemaal variabelen moet gaan setten die specifiek voor een Dag gelden, of voor een Maand, dan ben je dus te veel binnen één object bezig.

Dit soort dingen is gewoon een beetje logisch nadenken en kijken of je niet meerdere objecten aan het definiëren bent binnen één object. Als je meerdere functies krijgt als `formatDay()`, `forwardDay($days)`, `resetDay()`, etc, dan zal je wel een apart object moeten gebruiken. Want dan zijn de methods niet meer het Datum-object aan het manipuleren, maar het (niet-bestaande & te creëren) Dag-object.

Strikt gezien moet je wel aparte Dag-, Maand en Jaarobjecten maken omdat het op zichzelf staande 'dingen' zijn.

Misschien weet je al dat PHP zelf een DateTime class kent en je afvragen waarom je die niet gewoon kan gebruiken. In deze situatie zou je deze class zeker moeten gebruiken, maar als voorbeeld is dit een stuk duidelijker! Later in deze handleiding komen we nog terug op het gebruik van de DateTime class in PHP5.

Foute denkwijze

Veel programmeurs denken vaak: class == OOP. Voor de volledigheid, een class is een blauwdruk van een object. Dus het beschrijft hoe een object gemaakt gaat worden, maar het is op zichzelf geen object. Daarom maak je dus instanties van een class, die instanties noem je objecten.

Laten we eens kijken naar een abstract voorbeeld van een class die niets met OOP te maken heeft:

Code

```
+ Gastenboek
  - getReacties()
  - insertReactie()
  - printReacties()
  - editReactie(...)
  - getUsers()
  - createUser(...)
  - nextPage()
```

Hoe kun je zeggen dat dit geen OOP is zonder te zien wat het doet?
Simpel. Je ziet aan de namen van de methods al direct dat ze:

- reacties ophalen en invoegen
- één reactie kunnen wijzigen
- users kunnen ophalen en aanmaken
- iets met een volgende pagina kunnen doen

Je zou hier dus sowieso al te maken krijgen met de objecten `Reactie` en `User` aangezien er methods tussen zitten die specifiek de eigenschappen van een van deze objecten beïnvloeden. Je zou zeggen dat je ook het object `Page` krijgt, maar de pagina is slechts een eigenschap van het Gastenboek of zelfs gewoon een tijdelijk iets.

Aan de functienamen kun je dus meestal herkennen dat het geen goed object is. Daarnaast kun je het vaak herkennen aan het aantal regels code. Veel objecten bestaan echt maar uit maximaal 100 regels, meestal een stuk minder. Dit komt omdat elk object slechts z'n eigen functionaliteit definieert en verder alles aan de andere objecten overlaat. Je zult ook zien dat je dan heel weinig if/else statements krijgt en weinig loops en dergelijke. Uiteraard zijn er ook classes te vinden die meer dan 100 regels code bevatten, maar over het algemeen moet je de opbouw van je class nog eens overwegen als die uit honderden regels code bestaat.

Elk zelfstandig naamwoord is een object
Fiets, Plant, Gastenboek, Reactie, Formulier, Gebruiker, SMS, Connection, Filter, Validator, etc. Meestal zie je aan de naam van een class al of het kans van slagen heeft. Als je een class Reacties hebt dan hoor je al aan de naam dat dit niet één object is. Dus geen object. Nu worden dit soort constructies in de praktijk wel gebruikt om meerdere Reactie-objecten in te bewaren (die bijv. uit de database komen), maar dan komt dat terug in de naamgeving van die classes. In dit geval zou je dan bijvoorbeeld een `Reactie_Store`, of iedere andere variant die aangeeft dat het object meerdere Reactie objecten kan bevatten.

Object georiënteerd programmeren

In de voorgaande hoofdstukken heb je alles kunnen lezen over de denkwijze die schuil gaat achter OOP. Nu is het tijd om die denkwijze uitdrukking te geven in PHP code. Voordat we daadwerkelijk beginnen, kijken we nog even naar drie eerder genoemde conventies van OOP:

- Objecten definieer je in een class
- Eigenschappen definieer je als variabelen (properties) bovenaan je class
- Gedrag/veranderingen/manipulatie van eigenschappen definieer je als functies (methods) in je class

Aan de slag

Laten we snel bekijken hoe dat eruit ziet als we deze regels stap voor stap implementeren. In dit voorbeeld zullen we de basis van een eenvoudige `User` class opzetten.

Code

```php
<?php
class User {

}
?>
```

De volgende stap is het definiëren van de properties van de `User` class. Een eigenschap die iedere `User` heeft, is een gebruikersnaam.

Code

```php
<?php
class User {
  public $username;

}
?>
```

Het enige dat nu nog ontbreekt is een method om de gebruikersnaam te wijzigen en een method om die gebruikersnaam op te vragen.

In onderstaand voorbeeld wordt gebruik gemaakt van een set-method, iets dat later in deze handleiding weer naar de prullenbak zal verwijzen omdat er een veel mooiere manier voor is. Echter gebruiken we hem in dit voorbeeld toch om e.e.a. duidelijk te maken.

Code

```php
<?php
class User {
  public $username;

  public function setUsername($name) {
    $this->username = $name;
  }

  public function getUsername() {
    return $this->username;
  }
}
?>
```

We zien twee nieuwe methods in de class en aan de namen valt direct af te leiden wat ze precies doen. Met deze laatste toevoeging is onze eerste class al bruikbaar!

$this

In het laatste voorbeeld komen we twee keer de variabele `$this` tegen. Dit is een speciale variabele die altijd verwijst naar het huidige object of in andere woorden, `$this` is een speciale 'self-referencing' variabele. We kunnen `$this` gebruiken om properties en methods van de huidige klasse te benaderen.

De `setUsername()` method kent de waarde van `$name` toe aan de property `$username`. En in de method `getUsername()` wordt de waarde van property `$username` teruggegeven.

In het begin kan de werking van `$this` soms verwarrend zijn, beschouw het in dat geval maar als een speciale OO PHP variabele waarmee PHP weet wat er moet gebeuren. In de loop van deze handleiding raak je vanzelf vertrouwd met de werking en weet ook jij precies wat er gebeurt.

Een bruikbare class, en nu?

Ik vertelde al dat we nu een bruikbare `User` class hebben. Het volgende voorbeeld laat zien hoe je van deze `User` class een `User` object maakt en hoe je vervolgens met het `User` object kunt werken.

Code

```php
<?php
/* De User class */
class User {
  public $username;

  public function setUsername($name) {
    $this->username = $name;
  }

  public function getUsername() {
    return $this->username;
  }
}

/* Reguliere procedurele code */
$user = new User();
$user->setUsername('jan');

echo $user->getUsername();
?>
```

De output van dit scriptje ziet er als volgt uit:

Code: **output**

```
jan
```

Het eerste deel is de `User` class die we eerder gemaakt hebben. Over de procedurele code daaronder, moet nog het een en ander gezegd worden.

Op de eerste regel wordt het `User` object aangemaakt. Beter gezegd: er wordt een instantie van de `User` class aangemaakt of de `User` class wordt geïnstantieerd. Die instantie is vervolgens het `User` object waar je verder mee werkt. Het `User` object is

toegekend aan de variabele $user, hetgeen betekent dat we vanaf nu met $user deze instantie van de User class kunnen benaderen.

Op de regels daarna wordt achtereenvolgens de gebruikersnaam 'jan' aan het User object toegekend en de gebruikersnaam opgevraagd en geëchoed.

Meerdere instanties van een class
Zoals je wellicht al vermoedde is het mogelijk om meerdere instanties van een bepaalde class aan te maken. Sterker nog, het is zelfs vrij normaal dat dit gebeurt. Denk bijvoorbeeld aan een gastenboek waarin allemaal berichten staan. Elk bericht is dan een instantie van de Message class.
In het geval van onze User class werkt het precies hetzelfde. Er zijn genoeg situaties waarbij je twee of meer User objecten nodig hebt binnen een script. Dat ziet er als volgt uit.

Merk op dat hier de code van de User class zelf is weggelaten, dit is enkel de procedurele code. Uiteraard heb je de code van de User class wel nodig voor een werkend script.

Code

```php
<?php
$jan = new User();
$jan->setUsername('jan');

$inge = new User();
$inge->setUsername('inge');

echo 'Dit script kent twee gebruikers: '.$jan->getUsername().' en '.$inge->getUsername().'.';
?>
```

Code: **output**

```
Dit script kent twee gebruikers: jan en inge.
```

We zien nu twee verschillende variabelen ($jan en $inge) die verwijzen naar twee verschillende User objecten. Ook zien we dat aan verschillende objecten verschillende waarden toegekend kunnen worden, terwijl deze verschillende objecten gebasseerd zijn op dezelfde class.

Tot zover dit eerste hoofdstuk over het echte programmeer werk. Tot nu toe hebben we het volgende gedaan:

- Een User class ontworpen
- Verschillende objecten gebasseerd op die class aangemaakt
- Waarden toegekend op opgevraagd van die objecten
- Meerdere objecten binnen een script gebruikt

Voor een eerste hoofdstuk is dit voldoende stof. In de volgende hoofdstukken wordt verder ingegaan op de opbouw van je PHP classes en de mogelijkheden die je hebt.

Visibility

Visibility is de toegang die de buitenwereld (lees: andere objecten, globale functies of procedurele code) heeft tot properties en methods binnen je class. Tot nu toe zijn we een aantal keer het keyword 'public' tegengekomen en ben daar is - expres - nog niet verder op ingegaan. Maar nu is dan het moment om uit te leggen waar dit precies om gaat.

Binnen OOP kennen we drie keywords die te maken hebben met visibility: `public`, `protected` en `private`. Deze keywords vervullen een cruciale rol binnen het principe van Encapsulation en Data Hiding (denk nog even terug aan de definitie van OOP). Met behulp van deze keywords bepaal je welke onderdelen van je class wel of niet voor de buitenwereld toegankelijk moeten zijn.

Public

Het `public` keyword biedt de minste bescherming voor de properties en methods binnen je class. De buitenwereld heeft de mogelijkheid om inhoud van properties te wijzigen of methods uit te voeren.

Code

```php
<?php
class User {
  public $username;

  public function setUsername($name) {
    $this->username = $name;
  }

  public function getUsername() {
    return $this->username;
  }
}

$user = new User();
$user->setUsername('jan');

echo $user->username;
$user->username = 'inge';
?>
```

Aangezien de property `$username` `public` gedeclareerd is, zullen de laatste twee regels geen foutmeldingen opleveren. Het is in dit geval mogelijk om op die manier de property te echoën en aan te passen.

Private

Met het `private` keyword scherm je properties en methods af van de buitenwereld. Dit betekent dat je alleen vanuit de klasse zelf de properties kunt wijzigen of methods kunt aanroepen.

Code

```php
<?php
class User {
  private $username;

  public function setUsername($name) {
    $this->username = $name;
```

```
  }

  public function getUsername() {
    return $this->username;
  }
}

$user = new User();
$user->setUsername('jan');

echo $user->username;
echo $user->getUsername();
?>
```

Aangezien de property nu `private` gedeclareerd is, levert de op een na laatste regel een foutmelding op. Het is niet mogelijk om de property direct te benaderen. De laatste regel is nu nog de enige manier om de username weer te geven.

Protected

Het `protected` keyword speelt een belangrijke rol bij het onderwerp Inheritance (overerving) waar we later in deze handleiding op terug komen. Properties en methods die als protected gedeclareerd zijn, zijn alleen beschikbaar binnen de huidige class en alle child classes (classes die een uitbreiding zijn van de huidige class).

In het voorbeeld hieronder wordt gebruik gemaakt van inheritance. Dit onderwerp is nog niet behandeld en het kan ook geen kwaad om dit voor nu over te slaan. Echter, het spreekt voor zich, vandaar dat het voorbeeld hier wel opgenomen is.

Code

```
<?php
class User {
  protected $username;

  public function setUsername($name) {
    $this->username = $name;
  }

  public function getUsername() {
    return $this->username;
  }
}

class Premium_User extends User {
  private $premiumId;

  public function setData($username, $id) {
    $this->username = $username;
    $this->premiumId = $id;
  }
}

$user = new Premium_User();
$user->setData('jan', 100);

echo $user->username;
?>
```

Omdat de `$username` property nu `protected` gedeclareerd is, zal ook hier de laatste regel een foutmelding geven. De property is immers enkel te gebruiken vanuit de `User` en `Premium_User` class.

Welke visibility moet je nu gebruiken?
Ervaring is eigenlijk het enige dat je in staat stelt om op het eerste gezicht te bepalen niveau van visibility goed is voor je properties en methods. In het begin is het aan te raden om al je properties als `private` te declareren en al je methods als `public`. Op deze manier dwing je jezelf om de properties van een class vanuit de class zelf aan te passen, precies waar het bij OOP om draait. Een object kan alleen zijn eigen eigenschapen bepalen en niet die van een ander.

Visibility is een feature die sinds PHP5 beschikbaar is. Oudere versies van PHP kennen de keywords `public`, `protected` en `private` niet en gebruik ervan zal resulteren in een foutmelding.

Naamgeving

Ik dit hoofdstuk gaan we kort in op de naamgeving conventies binnen OOP. Uiteindelijk blijft het een persoonlijke keuze hoe je je classes, properties en methods noemt, echter is het aan te raden om je aan een van de bestaande conventies te kiezen en je daaraan te houden.

In deze handleiding maken we gebruik van de conventie die door Zend opgesteld is[1]. Deze conventie wordt veelvuldig gebruikt en zorgt er dus voor dat anderen jouw scripts ook eenvoudig kunnen lezen.

Classes

De naam van een class is altijd een zelfstandig naamwoord in het enkelvoud. De eerste letter van een class naam is altijd een hoofdletter, de rest van de letters is altijd lowercase. Mocht de naam uit meerdere woorden bestaan dan worden deze gescheiden door een underscore. De eerste letter van opeenvolgende woorden is ook een hoofdletter.

Voorbeelden van correcte class namen:

- `User`
- `Message`
- `Html_Table`
- `Message_Store`

Voorbeelden van incorrecte class namen:

- `user` (eerste letter geen hoofdletter)
- `Document_PDF` (te veel hoofdletters, DF van PDF moet lowercase)

Variabelen

Namen van properties/variabelen bestaan uit alfanumerieke tekens, alhoewel het gebruik van getallen in een naam afgeraden wordt. Ze beschrijven duidelijk de inhoud van de variabele. Mocht een variabele uit meerdere woorden bestaan, dan wordt de camelCase notatie gehanteerd, underscores horen daar niet thuis.

Namen van `private` of protected properties worden altijd vooraf gegaan door een underscore. Dit is de enige situatie waarin underscores in een variabelenaam voorkomen.

Voorbeelden van correcte property/variabele namen:

- `public $username`
- `public $homeAddress`
- `protected $_databaseConnection`
- `private $_resultSet`

Methods

De naam van een method beschrijft de handeling die de method uitvoert. Ze zijn volgens de camelCase notatie opgebouwd bestaan tevens enkel uit alfanumerieke tekens. Ook hier geldt dat private en protected methods voorafgegaan worden door een underscore.

Voorbeelden van correcte method namen:

- `public function filterInput()`
- `public function draw()`
- `protected function _calculateTotal()`
- `private function _countMessages()`

[1] Zie http://corni.ps/8m0Ng (http://framework.zend.com/manual/en/coding-standard.naming-conventions.html)

Constanten

Namen van constante properties worden volledig in hoofdletters geschreven. Meerdere woorden worden gescheiden door een underscore en de namen worden binnen de class vooraf gegaan door het keyword 'const'.

Voorbeelden van correcte constante namen:

- `const EMPTY`
- `const INVALID_ARGUMENT_COUNT`

Bestandsnamen

De naamgeving van je PHP bestanden staat los van de naamgeving in je PHP code, maar we gaan er toch even doorheen. Het is gebruikelijk om voor iedere class een apart PHP bestand aan te maken. Met oog op een functionaliteit die in een later hoofdstuk zal worden behandeld, is het verstandig om de volgende conventie aan te houden: `class_naam.class.php`. We zorgen dus dat de naam van de class terug komt in de bestandsnaam van het bestand waarin die class staat. De class naam wordt daarbij geheel lowercase geschreven.

Voorbeelden van bestandsnamen:

- `user.class.php`
- `message.class.php`
- `html_table.class.php`
- `message_store.class.php`

Zoals al eerder is gezegd is naamgeving vooral een kwestie van persoonlijke smaak en bewust programmeren. Ook zijn er meer conventies dan alleen die van Zend, dus kijk eens verder als deze je niet bevalt. Uiteindelijk komt het erop neer dat je in ieder geval voor jezelf een consistente naamgeving in je scripts moet gebruiken, anders zie je op een gegeven moment door de bomen het bos niet meer.

Het wordt tijd om weer eens naar wat code te gaan kijken. In het volgende hoofdstuk bespreken we het gebruik van de constructor van een class.

Constructor __construct()

Een constructor is een speciale functie binnen een class die automatisch uitgevoerd wordt zodra de class geïnstantieerd wordt. Op deze manier kunnen we het initialiseren van een class, dus het toekennen van waarden aan properties en bijvoorbeeld het opzetten van de benodigde database verbinding, automatiseren. Met andere woorden, we hoeven geen extra method(s) meer aan te roepen om dit te doen.

In PHP5 wordt de constructor gedefinieerd door de `__construct()` method. Deze method is een zogenaamde 'magic method' waarover later in deze handleiding meer verteld wordt. Voor nu kijken we eerst naar de werking van de constructor.

Definiëren van een constructor

De constructor wordt als volgt binnen een class gedefinieerd:

Code

```php
<?php
class User {
  public function __construct() {
    // Hier de code die uitgevoerd moet worden.
  }
}
?>
```

Zoals je ziet is het net een normale method, alleen dan een met een speciale functie. Laten we eens naar een realistischer voorbeeld kijken:

Code

```php
<?php
class User {
  private $_username;

  public function __construct($name) {
    $this->_username = $name;
  }

  public function getUsername() {
    return $this->_username;
  }
}

$user = new User('jan');
echo $user->getUsername();
?>
```

Code: **output**

```
jan
```

We zien dat de `setUsername()` method vervangen is door de constructor, die dezelfde parameter accepteert. In de procedurele code zien we dat het toekennen van de gebruikersnaam nu direct tijdens het instantiëren van de class gedaan wordt. Het is niet meer nodig om apart `setUsername()` aan te roepen. Sterker nog, in de veel gevallen kun je hierdoor de vele `set*()` methods de deur uit gooien, dat scheelt weer behoorlijk wat regels code!

Vereiste properties afdwingen

Naast het feit dat een constructor in veel gevallen regels code kan besparen, is er nog een andere belangrijke situatie waarin de constructor een uitkomst biedt. Stel dat we een class hebben waarbinnen een databaseverbinding nodig is, dan ligt het voor de hand om deze verbinding beschikbaar te hebben zodra de class geïnstantieerd wordt. Zonder deze verbinding zou de class anders niets waard zijn.

Als voorbeeld kijken we naar de class `Message_Controller` die kan zorgen dat `Message` objecten van en naar de database gestuurd worden.

Code

```php
<?php
class Message_Controller {
    private $_db;

    public function __construct($db) {
        $this->_db = $db;
    }

    public function storeMessage($message) {
        // Hier wordt $_db gebruikt om het $message object
        // naar de database te sturen.
    }
}
?>
```

De parameter $db waar de constructor om vraagt, zal een instantie van een of andere class moeten zijn die de communicatie naar de database kan afhandelen, bijvoorbeeld PDO.

Een constructor hoeft niet per se een parameter te accepteren, binnen de method kan net zo goed een actie uitgevoerd worden waarvoor geen parameters nodig zijn of er kan zelfs helemaal niets gebeuren. In PHP is het niet verplicht om je class van een constructor te voorzien, in tegenstelling tot andere programmeertalen zoals bijvoorbeeld Java. Maar aangezien wij in PHP programmeren, zullen we hem gebruiken als we hem nodig hebben (en dat zal bijna altijd zijn).

Voorbeeld: HTML tabel

Tot nu toe hebben we een redelijke basis om mee uit de voeten te kunnen. Er is nog zo veel meer over OOP te vertellen en een deel komt verderop in deze handleiding ook zeker aan bod, maar nu is het tijd voor een voorbeeld.

Ervaring is de sleutel tot succes, voordat je OOP een beetje onder de knie hebt ben je behoorlijk wat classes en scripts verder. We beginnen daarom met een voorbeeld die niet direct wat te maken heeft met OOP, maar het maakt de informatie uit de voorgaande hoofdstukken wel goed duidelijk. Laten we naar het voorbeeld kijken: de tabel in HTML.

Denk bij het definiëren van namen voor classes, methods en properties nog even terug aan het hoofdstuk over naamgeving conventies!

Stap 1: doel bepalen en objecten herkennen
In dit voorbeeld is het doel om, op OOP wijze, een script te schrijven dat de weergave van een HTML tabel verzorgt. Nu weten we allemaal hoe een tabel in HTML eruit ziet, maar voor het herkennen van objecten hier nog eens de meest eenvoudige structuur:

Code

```
<table>
  <tr>
    <td>
    </td>
  </tr>
</table>
```

Een tabel bestaat in alle gevallen uit rijen die op hun beurt weer uit cellen bestaan. Hier herkennen we drie objecten met allen hun specifieke eigenschappen: `Tabel`, `Rij` en `Cel`. Kortom, we zullen (minimaal) drie classes krijgen om deze objecten te beschrijven. De eerste opzet is geboren:

Code

```
<?php
class Table {

}

class Row {

}

class Cell {

}
?>
```

Stap 2: herkennen van eigenschappen van objecten
De volgende stap is het herkennen van de eigenschappen van de verschillende objecten. Pas als we dat weten, kunnen we de benodigde properties aan onze classes toevoegen. Als we opmaak van de tabel buiten beschouwing laten, zijn rijen een eigenschap van de tabel. Cellen zijn op hun beurt een eigenschap van een rij en tenslotte is de inhoud van een cel de eigenschap van een cel.

In de `Table` en `Row` classes moeten we dus een property hebben voor respectievelijk de rijen en de cellen. De `Cell` class heeft een property nodig voor de content.

Code

```php
<?php
class Table {
    private $_rows;

}

class Row {
    private $_cells;

}

class Cell {
    private $_content;

}
?>
```

Stap 3: bepalen wat de verschillende objecten moeten kunnen

Binnen de `Cell` class hebben we op dit moment alleen een method nodig waarmee we de content kunnen aanmaken en een waarmee we de content kunnen opvragen. In het eerste geval kunnen we mooi de constructor gebruiken en voor het tweede ligt een method `getContent()` voor de hand.

De `Row` class heeft een method nodig om cellen aan de rij toe te voegen en een om alle cellen op te vragen. De methods `append()` en `getCells()` zouden hier logische keuzes zijn.

Als laatste heeft de `Table` class een method nodig om rijen aan de tabel toe te voegen en een om de tabel weer te geven. Hier zouden de methods `append()` en `draw()` voor kunnen zorgen.

Code

```php
<?php
class Table {
    private $_rows;

    public function __construct() {
        $this->_rows = array();
    }

    public function append($row) {
        $this->_rows[] = $row;
    }

    public function draw() {

    }
}

class Row {
    private $_cells;

    public function __construct() {
        $this->_cells = array();
    }

    public function append($cell) {
```

```
26          $this->_cells[] = $cell;
27      }
28
29      public function getCells() {
30          return $this->_cells;
31      }
32  }
33
34  class Cell {
35      private $_content;
36
37      public function __construct($content) {
38          $this->_content = $content;
39      }
40
41      public function getContent() {
42          return $this->_content;
43      }
44  }
45  ?>
```

Je ziet dat de classes `Table` en `Row` ook een constructor gekregen hebben. Dit heeft ermee te maken dat de properties `$_rows` en `$_cells` als array gedeclareerd moeten worden om later een `Notice` te voorkomen als we waarden aan de arrays toevoegen. Dit is een stukje vooruit kijken en zou je eventueel ook met een if-statement in de `append()` methods op kunnen vangen, maar deze manier is netter. Zorg altijd dat je properties op een juiste manier geïnitialiseerd zijn.

Stap 4: de `draw()` method

Het enige dat nu nog moet gebeuren voordat we het geheel kunnen gebruiken is het schrijven van de `draw()` method. Dit is bewust nog niet gedaan omdat hier een paar kanttekeningen bij geplaatst moeten worden. Normaal gesproken is het namelijk niet gewenst om vanuit methods te echoën. Het is veel vanzelfsprekender om waarden te retourneren om die vervolgens in je procedurele code te echoën. Voor nu negeren we dat even, de reden daarvoor zal later in deze handleiding duidelijk worden.

De `draw()` method zou er als volgt uit kunnen zien:

Code

```
1   <?php
2   public function draw() {
3       echo '<table border="1">'.PHP_EOL; //Begin van tabel, border voor duidelijkheid
4
5       foreach($this->_rows as $row) {
6           echo '<tr>'.PHP_EOL;
7
8           foreach($row->getCells() as $cell) {
9               echo '<td>'.$cell->getContent().'</td>'.PHP_EOL;
10          }
11
12          echo '</tr>'.PHP_EOL;
13      }
14
15      echo '</table>'.PHP_EOL;
16  }
17  ?>
```

Stap 5: de procedurele code

Zoals we nu inmiddels weten vormen de classes enkel de blauwdrukken van de objecten die we kunnen gebruiken. We zullen dus nog een klein stukje procedurele code moeten schrijven waarin we de classes (een of meerdere keren) instantiëren om uiteindelijk een tabel met inhoud weer te kunnen geven.

Code

```php
<?php
/* Procedurele code */
$cellA1 = new Cell('Dit is cel A1');
$cellA2 = new Cell('Dit is cel A2');

$rowA = new Row();
$rowA->append($cellA1);
$rowA->append($cellA2);
$rowA->append(new Cell('Dit is cel A3')); // Zo kan het ook!

$table = new Table();
$table->append($rowA);
$table->draw();
?>
```

Dit is een klein voorbeeldje van hoe je met de drie gemaakte classes kunt werken. Allereerst definiëren we twee `Cell` objecten met een bepaalde inhoud. Vervolgens creëren we een instantie van de `Row` class om daar vervolgens de twee `Cell` objecten aan toe te voegen. Een derde cel wordt aan de rij toegevoegd om te illustreren dat je niet per se een variabele hoeft te declareren voordat je het object kunt gebruiken. Tenslotte wordt het `Table` object geïnstantieerd, de rij toegevoegd en wordt de tabel weergegeven.

De broncode van de resulterende pagina ziet er als volgt uit:

Code

```html
<table border="1">
<tr>
<td>Dit is cel A1</td>
<td>Dit is cel A2</td>
<td>Dit is cel A3</td>
</tr>
</table>
```

Conclusie

Een HTML tabel is opgebouwd uit verschillende onderdelen die wij als objecten gedfinieerd hebben. Met behulp van een beetje OOP kennis is het ons gelukt om de weergave van een HTML tabel te verzorgen met behulp van 3 classes en een klein beetje procedurele code. Dit voorbeeld laat zien hoe je iets opbreekt in objecten en deze objecten vervolgens samen laat komen tot een geheel.

In de rest van deze handleiding wordt nog een aantal keer terug gekomen op dit voorbeeld om te laten zien hoe een en ander eenvoudiger of anders kan.

Inheritance

Een van de onderwerpen waar de grote kracht van OOP naar voren komt, is inheritance (overerving). Inheritance is het principe waarbij classes uitgebreid (extend) kunnen worden door een nieuwe class (child) te schrijven die de eigenschappen van de oorspronkelijke class (parent) erft. Het resulterende object van een child class heeft alle eigenschappen van de parent class plus de nieuw gedefinieerde eigenschappen van de child class.

Wanneer is extenden toegestaan?

Voordat we verder gaan wil even je aandacht op de belangrijkste regel als het gaat om extenden van classes. Je kunt namelijk alleen een child class 'X' van de parent class 'Y' aanmaken als je kunt zeggen 'elke X is een Y'. Zo zou `Premium_User` een child van de parent `User` kunnen zijn omdat je kunt zeggen 'elke `Premium_User` is een `User`'. Andersom daarentegen kan nooit, want je kunt niet zeggen 'elke `User` is een `Premium_User`'.

Correcte voorbeelden:

- `MySQL extends Database` -> kan, MySQL is een type database
- `Mens extends Dier` -> kan, ieder mens is een dier
- `Dier extends Organisme` -> kan, ieder dier is een organisme
- `Mens extends Organisme` -> kan ook, ieder mens is immers een organisme

Incorrecte voorbeelden:

- `Query extends MySQL` -> kan niet, een query is geen database!
- `Dier extends Konijn` -> kan niet, niet ieder dier is een konijn!

De code

Zoals je misschien al vermoedde (of in het hoofdstuk over Visibility gezien hebt), extenden doe je met het '`extends`' keyword. Laten we onze `User` class er weer eens bij pakken:

Code

```php
<?php
class User {
    protected $_username;

      public function __construct($name) {
        $this->_username = $name;
    }

    public function getUsername() {
        return $this->_username;
    }
}
?>
```

Merk op dat de visibility van de property `$_username` veranderd is naar '`protected`'. We gaan deze class uitbreiden en we willen deze variabele vanuit de child class kunnen benaderen en manipuleren.

Stel nu dat we voor een webshop de class `Customer` nodig hebben. Iedere klant is een gebruiker van de webshop, we kunnen de `User` class dus extenden en zo de `Customer` class maken. Iedere klant heeft een eigen klantnummer, iets dat een gebruiker niet per

se hoeft te hebben. Dat is dus een specifieke eigenschap van de klant en dus een property van de `Customer` class.

Code

```php
<?php
class User {
    protected $_username;

      public function __construct($name) {
        $this->_username = $name;
    }

    public function getUsername() {
        return $this->_username;
    }
}

class Customer extends User {
    private $_customerId;

    public function __construct($username, $customerId) {
        $this->_username = $username;
        $this->_customerId = $customerId;
    }
}

$customer = new Customer('jan', 1);
echo $customer->getUsername();
?>
```

Code: **output**

```
jan
```

De nieuwe class `Customer` heeft een property `$customerId` en de constructor van deze class initialiseert zowel de property van de child class als die van de parent `User` class. Tevens zien we dat we het `Customer` object `$customer` kunnen gebruiken om methods uit de parent class aan te roepen. Dit laat zien dat het child object de eigen eigenschappen heeft maar ook de eigenschappen van de parent class zonder dat we daar extra regels code voor hoeven toe te voegen aan de child class.

Dit is een klassiek voorbeeld van hoe OOP het benodigd aantal regels code kan reduceren, dezelfde methods hoeven niet telkens opnieuw geschreven te worden.

Methods overschrijven

Soms komt het voor dat je bestaande methods uit de parent class wilt overschrijven in de child class. Met andere woorden, je wilt een andere werking toekennen aan een bepaalde method. Dit doe je heel eenvoudig door een method met dezelfde naam als die in de parent class, te definiëren in de child class:

Code

```php
<?php
class User {
    protected $_username;

      public function __construct($name) {
        $this->_username = $name;
    }

```

```
9       public function getUsername() {
10          return $this->_username;
11      }
12  }
13
14  class Customer extends User {
15      private $_customerId;
16
17      public function __construct($username, $customerId) {
18          $this->_username = $username;
19          $this->_customerId = $customerId;
20      }
21
22      public function getUsername() {
23          return 'De gebruikersnaam is: '.$this->_username;
24      }
25  }
26
27  $customer = new Customer('jan', 1);
28  echo $customer->getUsername();
29  ?>
```

Code: **output**

```
1   De gebruikersnaam is: jan
```

We zien dat hier de method uit de Customer class uitgevoerd wordt en dat er een uitgebreidere string geretourneerd wordt.

Methods uit de parent class uitvoeren

Bij het overschrijven van methods kan het in sommige gevallen voorkomen dat je toch de oorspronkelijke method uit de parent class uit wilt voeren. Dat kun je doen door het 'parent' keyword te gebruiken.

```
1   <?php
2   class User {
3       protected $_username;
4
5         public function __construct($name) {
6           $this->_username = $name;
7       }
8
9       public function getUsername() {
10          return $this->_username;
11      }
12  }
13
14  class Customer extends User {
15      private $_customerId;
16
17      public function __construct($username, $customerId) {
18          $this->_username = $username;
19          $this->_customerId = $customerId;
20      }
21
22      public function getUsername() {
23          if($this->_username == 'jan') {
24              return parent::getUsername();
25          }
26          else {
```

```
27              return 'De gebruikersnaam is: '.$this->_username;
28          }
29      }
30  }
31
32  $jan = new Customer('jan', 1);
33  $inge = new Customer('inge', 2);
34
35  echo $jan->getUsername(). '<br />' .$inge->getUsername();
36  ?>
```

Code: **output**

```
1  De gebruikersnaam is: jan
```

In de method `getUsername()` in de `Customer` class hebben we als voorwaarde gesteld dat de `getUsername()` method uit de `User` class aangeroepen moet worden als de gebruikersnaam gelijk is aan 'jan'. In de output zien we dat gebeuren.

Tot zover dit hoofdstuk over inheritance. We kunnen concluderen dat ook dit onderwerp vooral terug grijpt op de denkwijze die achter OOP schuil gaat. De bijbehorende PHP code is niet bijzonder lastig, je zult alleen moeten zorgen dat de logica van je applicatie ook in orde is. Mocht je tijdens het programmeren op dit soort punten vastlopen, dan is het verstandig om al je (denk)stappen nog eens na te lopen en te controlen of de opzet die je gebruikt eigenlijk wel klopt. Vaak ligt daar namelijk het probleem.

Voorbeeld: HTML tabel 2 (inheritance)

In het eerdere voorbeeld van de HTML tabel hebben we gekeken naar de manier waarop dit OO geprogrammeerd wordt. In dit hoofdstuk gaan we een stapje verder en kijken we naar specifieke eigenschappen van de verschillende onderdelen van de tabel. We gaan op zoek naar een manier om cellen (of althans de inhoud daarvan) efficiënt op verschillende manieren te tonen.

Voor dit voorbeeld zijn een aantal kleine wijzigingen doorgevoerd in de eerdere voorbeeld code:

Code

```php
<?php
class Table {
    private $_rows;

    public function __construct() {
        $this->_rows = array();
    }

    public function append($row) {
        $this->_rows[] = $row;
    }

    public function draw() {
        echo '<table border="1">'.PHP_EOL;

        foreach($this->_rows as $row) {
            $row->draw();
        }

        echo '</table>'.PHP_EOL;
    }
}

class Row {
    private $_cells;

    public function __construct() {
        $this->_cells = array();
    }

    public function append($cell) {
        $this->_cells[] = $cell;
    }

    public function draw() {
        echo '<tr>'.PHP_EOL;

        foreach($this->_cells as $cell) {
            $cell->draw();
        }

        echo '</tr>'.PHP_EOL;
    }
}

class Cell {
    protected $_content;

```

```
49      public function __construct($content) {
50          $this->_content = $content;
51      }
52
53      public function draw() {
54          echo '<td>'.$this->_content.'</td>'.PHP_EOL;
55      }
56  }
57  ?>
```

We zien dat elke class zijn eigen `draw()` method heeft gekregen. In het kader van encapsulation en eigen verantwoordelijkheid van objecten (denk even terug aan de definitie van OOP) is deze opzet eigenlijk nog iets logischer. Een `Cell` bepaalt immers hoe hij in HTML code weergegeven moet worden, een `Table` hoort daar niets vanaf te weten. Ook zien we dat de property `$_content` de visibility protected heeft gekregen. Deze class gaan we zo extenden en deze property moeten we vanuit de child kunnen benaderen.

```
1   <?php
2   class Strong_Cell extends Cell {
3       public function __construct($content) {
4           parent::__construct($content);
5       }
6
7       public function draw() {
8           echo '<td><strong>'.$this->_content.'</strong></td>'.PHP_EOL;
9       }
10  }
11  ?>
```

Hier zien we de `Strong_Cell` class, een child van de `Cell` class. Omdat de constructor van de child precies dezelfde functionaliteit heeft als die van de parent, is het eenvoudiger om simpel de parent constructor aan te roepen. De `draw()` method daarentegen heeft wel een andere functionaliteit en dus overschrijven we die in de child class. In plaats van de content in een normale cel te plaatsen, worden nu ook de `<strong></strong>` tags geëchoed om te zorgen dat de content dikgedrukt wordt.

De procedurele code zou er nu als volgt uit kunnen zien:

```
1   <?php
2   /* Procedurele code */
3   $cellA1 = new Cell('Dit is cel A1');
4   $cellA2 = new Strong_Cell('Dit is cel A2');
5
6   $rowA = new Row();
7   $rowA->append($cellA1);
8   $rowA->append($cellA2);
9   $rowA->append(new Cell('Dit is cel A3')); // Zo kan het ook!
10
11  $table = new Table();
12  $table->append($rowA);
13  $table->draw();
14  ?>
```

De broncode van de resulterende pagina ziet er als volgt uit:

Code: **output**

```
<table border="1">
<tr>
<td>Dit is cel A1</td>
<td><strong>Dit is cel A2</strong></td>
<td>Dit is cel A3</td>
</tr>
</table>
```

Dit voorbeeld laat zien dat we heel eenvoudig de weergave van een cel kunnen aanpassen door bijvoorbeeld het `Strong_Cell` object te gebruiken. Uiteraard zou je dit veel verder kunnen uitbreiden met bijvoorbeeld een `Underlined_Cell` en `Italic_Cell` voor cellen met respectievelijk onderstreepte en cursieve inhoud.

Hier zijn we ingegaan op de weergave van bepaalde objecten, en dat terwijl we eerder zeiden dat je zoveel mogelijk echo's in je classes moet proberen te voorkomen. Dat is nog steeds het standpunt en dit is dan ook enkel een voorbeeld. En visualisatie is nu eenmaal erg waardevol bij voorbeelden. Normaal gesproken zou je veel liever een andere oplossing kiezen voor de opmaak, zoals bijvoorbeeld het gebruik van templates.

Static methods en properties

Gebruik van het 'static' keyword zorgt ervoor dat betreffende methods of properties gebruikt kunnen worden zonder dat er een instantie van de class aangemaakt hoeft te worden. Omdat de class niet geïnstantieerd wordt, is het ook niet mogelijk om gebruik te maken van $this, aangezien deze variabele naar het huidige object verwijst. Static properties behoren dan ook tot de class zelf en niet tot een object van die class. Om properties of methods van binnenuit de class te benaderen, gebruiken we het 'self' keyword.

Een voorbeeld waarin het gebruik duidelijk wordt is deze Counter class:

Code

```php
<?php
class Counter {
    private static $_count = 0;

    public function __construct() {
        self::$_count++;
    }

    public static function getCount() {
        return self::$_count;
    }
}

echo Counter::getCount().'<br />';

$x = new Counter;
echo Counter::getCount().'<br />';

$y = new Counter;
echo Counter::getCount().'<br />';

$z = new Counter;
echo Counter::getCount().'<br />';
?>
```

Code: **output**

```
0
1
2
3
```

We zien een static property $_count die zoals gezegd dus tot de class zelf behoort en niet tot een object van die class. Verder zien we een constructor die met behulp van het 'self' keyword telkens de waarde van $_count met 1 ophoogt zodra de class geïnstantieerd wordt. Tenslote zien we nog een static method getCount() die de huidige waarde van $_count terug geeft.

In de procedurele code wordt de Counter class tot driemaal toe geïnstantieerd, resulterende in een verhoging van $_count zoals in de output te zien is. Verder is te zien dat public static gedeclareerde members of properties ook van buiten de class benaderbaar zijn, ook zonder dat de class ooit geïnstantieerd is geweest.

Abstract classes en Interfaces

In dit hoofdstuk bekijken we een ander krachtig onderdeel van OOP, namelijk het gebruik van abstract classes en interfaces. Beide middelen zijn bedoeld om de programmeur (jijzelf, of iemand anders) te dwingen bepaalde methods of properties te gebruiken. Op die manier kun je vooraf bepalen hoe bepaalde classes gebruikt dienen te worden of in een applicatie opgenomen dienen te worden.

Abstract classes

Een abstract class is een class met of zonder eigen properties en een aantal methods die gedeeltelijk de functionaliteit van de class bepalen maar tegelijkertijd een deel van de functionliteit onbepaald laat. Het onbepaalde gedeelte zijn de abstract methods en deze dienen uitgewerkt te worden in de child class die deze abstract class extend.

Deze lastige definitie is eigenlijk alleen maar goed uit te leggen met een voorbeeld, dus laten we de User class er weer eens bijpakken. Stel je nu de situatie voor dat je een webshop aan het bouwen bent waarbij je twee verschillende typen gebruikers kent: klanten en werknemers. In het hoofdstuk over inheritance hebben we gezien hoe de `User` class te extenden is tot een `Customer` class, maar het grote nadeel is dat er in dat geval nog steeds een `User` object aangemaakt kan worden waar je eigenlijk niets mee kan, we hebben immers alleen klanten en werknemers geen gebruikers zonder functie. Om dat te voorkomen definiëren we de `User` class nu als abstract, hetgeen ondermeer betekent dat hij niet geïnstantieerd kan worden.

Code

```php
<?php
abstract class User {
    private $_username;

      public function __construct($name) {
        $this->_username = $name;
    }

    public function getUsername() {
        return $this->_username;
    }

    public abstract function getUserStatus();
}
?>
```

Deze class komt ons inmiddels bekend voor, maar de abstract method `getUserStatus()` is nieuw. Dat deze method als abstract gedeclareerd is, betekent dat het de verantwoordelijkheid is van de child class om voor de functionaliteit van `getUserStatus()` te zorgen. Deze method moet wel abstract zijn omdat de functionaliteit verschillend is bij de child classes.

De twee classes die we nu nog missen, `Customer` en `Employee`, zijn beide een child van de `User` class. Beide classes worden dus gedwongen om minimaal de `getUserStatus()` method te definiëren.

Code

```php
<?php
class Customer extends User {
```

```
    private $_customerId;

    public function __construct($username, $id) {
        $this->_username = $username;
        $this->_customerId = $id;
    }

    public function getUserStatus() {
        return 'customer';
    }
}

class Employee extends User {
    private $_employeeId;

    public function __construct($username, $id) {
        $this->_username = $username;
        $this->_employeeId = $id;
    }

    public function getUserStatus() {
        return 'employee';
    }
}
?>
```

De twee classes lijken (nog) erg veel op elkaar, het belangrijke verschil zit hem echter in de `getUserStatus()` method. Als we het geheel samenvoegen met de `User` class en nog een paar regels procedurele code toevoegen, is dit het resultaat:

Code

```
<?php
abstract class User {
    private $_username;

      public function __construct($name) {
        $this->_username = $name;
    }

    public function getUsername() {
        return $this->_username;
    }

    public abstract function getUserStatus();
}

class Customer extends User {
    private $_customerId;

    public function __construct($username, $id) {
        $this->_username = $username;
        $this->_customerId = $id;
    }

    public function getUserStatus() {
        return 'customer';
    }
}

class Employee extends User {
```

```
30      private $_employeeId;
31
32      public function __construct($username, $id) {
33          $this->_username = $username;
34          $this->_employeeId = $id;
35      }
36
37      public function getUserStatus() {
38          return 'employee';
39      }
40  }
41
42  $jan = new Customer('jan', 1);
43  $inge = new Employee('inge', 1);
44
45  echo 'Jan is een '.$jan->getUserStatus().'. <br />';
46  echo 'Inge is een '.$inge->getUserStatus().'.';
47  ?>
```

Code: **output**

```
1  Jan is een customer.
2  Inge is een employee.
```

Zoals aan de output te zien is, doet de `getUserMethod()` wat van hem gevraagd wordt.

Dit voorbeeld geeft zeer eenvoudig de werking van abstract classes weer. Op deze manier kun je vooraf een gemeenschappelijk gedeelte van meerdere classes programmeren om deze abstract class later te extenden met de classes die je daadwerkelijk gaat gebruiken. Het grote voordeel: de gemeenschappelijke functionaliteit hoef je maar een keer te programmeren.

Interfaces

Een interface is een overeenkomst tussen ongerelateerde objecten voor het uitvoeren van dezelfde functionaliteit. Een interface stelt je in staat om aan te geven dat een object een bepaalde functionaliteit moet bezitten, maar het bepaalt niet hoe het object dat moet doen. De child class is dus vrij om de hele implementatie te doen, zolang hij maar voldoet aan de functionaliteit die de interface afdwingt.

In het geval van een interface extend de child class de parent niet, maar implementeert hij hem. Daartoe maken we gebruik van het keyword '`implements`'.

Stel dat we bezig zijn met het ontwikkelen van een applicatie die de communicatie met verschillende type databases moet kunnen afhandelen. Bekend is dat de ene database anders werkt dan de ander en dat vaak verschillende (PHP) functies nodig zijn. Het is onmogelijk om één class te schrijven die met alle type databases werkt, sterker nog voor elke database heb je een aparte class nodig. Maar we kunnen wel vooraf de functionaliteit bepalen die elke database class minimaal moet hebben, ongeacht de database waarmee we werken. Dat zou er als volgt uit kunnen zien:

Code

```
1  <?php
2  interface Database
3  {
4      public function connect();
5      public function error();
6      public function errno();
```

```
7       public function escape($string);
8       public function query($query);
9       public function fetchArray($result);
10      public function fetchRow($result);
11      public function fetchAssoc($result);
12      public function fetchObject($result);
13      public function numRows($result);
14      public function close();
15  }
16  ?>
```

Deze interface dwingt elke class die hem implementeert om minimaal functionaliteit toe te kennen aan deze methods. Bovendien moet de child class bij elke method minimaal de parameters accepteren die in de interface bepaald zijn. Een method mag meer parameters hebben, zolang ze optioneel zijn, maar zeker niet minder.

Een child class die de communicatie met een MySQL database kan afhandelen, zou er als volgt uit kunnen zien:

Code

```
1   <?php
2   class Mysql_Database implements Database {
3       private $_link;
4
5       public function connect($server='', $username='', $password='', $new_link=true,
          $client_flags=0) {
6           $this->_link =
              mysql_connect($server, $username, $password, $new_link, $client_flags);
7       }
8
9       public function error() {
10          return mysql_errno($this->_link);
11      }
12
13      public function errno() {
14          return mysql_error($this->_link);
15      }
16
17      public function escape($string) {
18          return mysql_real_escape_string($string, $this->_link);
19      }
20
21      public function query($query) {
22          return mysql_query($query, $this->_link);
23      }
24
25      public function fetchArray($result, $array_type = MYSQL_BOTH) {
26          return mysql_fetch_array($result, $array_type);
27      }
28
29      public function fetchRow($result) {
30          return mysql_fetch_row($result);
31      }
32
33      public function fetchAssoc($result) {
34          return mysql_fetch_assoc($result);
35      }
36
37      public function fetchObject($result) {
```

```
38          return mysql_fetch_object($result);
39      }
40
41      public function numRows($result) {
42          return mysql_num_rows($result);
43      }
44
45      public function close() {
46          return mysql_close($this->_link);
47      }
48  }
49  ?>
```

Dit is de functionaliteit die door de Database interface afgedwongen wordt en elke database class moet bezitten. Er zijn echter veel meer MySQL functies dus deze class zou verder uitgebreid kunnen worden om hem beter aan te laten sluiten op de MySQL functionaliteit. Deze selectie van methods is echter voor elke database te implementeren, daarom worden ze afgedwongen door de interface.

De procedurele code om te communiceren met de database zou er nu als volgt uit kunnen zien:

Code

```
1  <?php
2  $db = new Mysql_Database();
3  $db->connect('host', 'username', 'password');
4  $db->query('USE webshop');
5
6  $result = $db->query("SELECT username FROM users");
7
8  while($row = $db->fetchAssoc($result)) {
9      echo($row['username']);
10 }
11 ?>
```

Als we de classes voor andere databases geschreven hebben, kunnen we in dit voorbeeldje eenvoudig van database wisselen door enkel de eerste regel te veranderen. Voor een postgreSQL database zou dat bijvoorbeeld zo kunnen zijn:

Code

```
1  <?php
2  $db = new Postgresql_Database();
3  ?>
```

Verschil tussen abstract classes en interfaces

Ze lijken erg op elkaar, maar er zijn een aantal belangrijke verschillen tussen abstract classes en interfaces.

Abstract classes

- Een abstract class kan bepaalde functionaliteit definiëren en de rest overlaten aan de child.
- Een child kan de reeds gedefinieerde methods overschrijven, maar hoeft dat niet.
- De child class moet een logische relatie hebben met de parent.
- Een child kan maximaal een abstract class extenden.

Interfaces

- Een interface kan geen functionalteit bevatten. Het is enkel een definitie van de methods die gebruikt moeten worden.
- De child class moet alle methods uit de interface van functionaliteit voorzien.

- Verschillende niet gerelateerde classes kunnen op een logische manier gegroepeerd worden door een interface.
- Een child kan meerdere interfaces tegelijkertijd implementeren.

Magic methods

Binnen OO PHP kennen we een aantal zogenaamde magic methods. Dit zijn methods die binnen elke class werken en een speciale functionaliteit hebben. Eerder in deze handleiding zijn we al zo'n method tegen gekomen, namelijk `__construct()`, die de constructor van een class definieert.

Magic methods zijn herkenbaar aan de dubbele underscore voorafgaand aan de naam van de method. In dit hoofdstuk gaan we de andere magic methods bespreken en het gebruik ervan toelichten.

`__set()` en `__get()`

PHP is standaard een 'loosely typed language' en als gevolg daarvan is het niet noodzakelijk om variabelen te declareren alvorens ze te gebruiken. Ditzelfde geldt voor de properties van een class:

Code

```php
<?php
class User {
  public $username;
}

$user = new User();
$user->username = 'jan';
$user->email = 'jan@email-example.nl';

echo $user->email;
?>
```

Code: **output**

```
jan@email-example.nl
```

Dit voorbeeld zal geen foutmeldingen opleveren ondanks dat de property `$email` niet bestaat. PHP zal deze property gewoon aanmaken met een `public` visibility en de opgegeven waarde eraan toekennen.

In eerdere hoofdstukken van deze handleiding zou dit gedrag van PHP een reden zijn om nooit deze notatie te gebruiken om properties van een object te manipuleren. Daar zouden we liever gekozen hebben voor een alternatieve oplossing via een set- of get-method. PHP biedt echter een oplossing waarmee we dit gedrag kunnen beïnvloeden, de `__set()` en `__get()` magic methods.

Met behulp van `__set()` kun je beïnvloeden hoe informatie opgeslagen wordt binnen je object. De `__get()` method bepaalt logischerwijs de manier hoe informatie uit je object opgehaald kan worden:

Code

```php
<?php
class User {
    public $username;
    private $_data;

    public function __construct($username) {
        $this->username = $username;
        $this->_data = array();
    }
```

```
10
11      public function __set($var, $value) {
12          $this->_data[$var] = $value;
13      }
14
15      public function __get($var) {
16          if(isset($this->_data[$var])) {
17              return $this->_data[$var];
18          }
19      }
20  }
21
22  $jan = new User('jan');
23  $jan->email = 'jan@email-example.nl';
24
25  echo $jan->email;
26  ?>
```

Code: **output**

```
1  jan@email-example.nl
```

Dit voorbeeld verschilt qua output niets van het voorbeeld daarvoor maar achter de schermen gebeurt er wel degelijk iets anders. PHP herkent dat de `$email` property niet bestaat binnen de class. Er wordt gekeken of er een `__set()` method bestaat die dit af kan handelen voordat de property als `public` gedeclareerd wordt. In ons geval is dat zo en wordt de waarde weggeschreven naar de array `$_data`.

De `__get()` method werkt op eenzelfde manier. Als er een property opgevraagd wordt die niet bestaat, zal in plaats van een foutmelding eerst `__get()` aangeroepen worden. Aangezien wij in `__set()` de waarden weggeschreven hebben naar de `$_data` array, zullen we in `__get()` controleren of de opgevraagde variabele daarin voorkomt. Als dat het geval is, wordt de waarde van bijbehorende variabele geretourneerd.

De `__set()` method is ook te gebruiken om het loosely typed gedrag van PHP in te perken. Het enige dat we dan hoeven te doen, is vanuit de `__set()` method een foutmelding geven en op die manier voorkomen dat er geen properties gebruikt kunnen worden die vooraf niet gedeclareerd zijn. Dit zou er als volgt uit kunnen zien:

Code

```
1  <?php
2  class User {
3      private $_username;
4
5      public function __construct($username) {
6          $this->_username = $username;
7      }
8
9      public function __set($var, $value) {
10         echo 'Kan geen waarde toekennen aan onbestaande property $'.$var.'.';
11     }
12 }
13
14 $jan = new User('jan');
15 $jan->email = 'jan@email-example.nl';
16 ?>
```

Code: **output**

```
Kan geen waarde toekennen aan onbestaande property $email.
```

Deze methode sluit goed aan bij de gedachte dat elk object zijn eigen eigenschappen bepaalt en de enige is die daar verantwoordelijk voor is.

Opmerking: het is niet gebruikelijk om op deze manier foutmeldingen te geven vanuit een class. Dit doen we liever met behulp van de `Exception` class waar in het hoofdstuk over foutafhandeling op ingegaan zal worden.

`__isset()` en `__unset()`

De `__isset()` en `__unset()` methods kunnen we gebruiken om enerzijds te controleren of niet vooraf gedeclareerde properties bestaan (dus met behulp van `__set()` aangemaakt zijn) en anderzijds bestaande niet vooraf gedeclareerde properties unsetten. Dit kan er als volgt uit zien:

Code

```php
<?php
class User {
    public $username;
    private $_data;

    public function __construct($username) {
        $this->username = $username;
        $this->_data = array();
    }

    public function __set($var, $value) {
        $this->_data[$var] = $value;
    }

    public function __get($var) {
        if(isset($this->_data[$var])) {
            return $this->_data[$var];
        }
    }

    public function __isset($var) {
        return isset($this->_data[$var]);
    }

    public function __unset($var) {
        unset($this->_data[$var]);
    }
}

$jan = new User('jan');
$jan->email = 'jan@email-example.nl';

var_dump(isset($jan->email));
unset($jan->email);
var_dump(isset($jan->email));
?>
```

Code: **output**

```
bool(true) bool(false)
```

Dit voorbeeld spreekt redelijk voor zich. Zodra `isset()` aangeroepen wordt voor een properties die in eerste instantie niet gedclareerd is in de class, wordt automatisch de `__isset()` method aangeroepen. Hierin is vervolgens de functionaliteit geprogrammeerd dat bekeken wordt of de betreffende variabele in de `$_data` array voorkomt. Hetzelfde geldt voor het aanroepen van de `unset()` functie op niet vooraf gedeclareerde properties.

Let wel goed op met het gebruik van deze functionaliteit en dan vooral bij het gebruik van `unset()`. Het is namelijk ook mogelijk om `public` gedeclareerde properties te unsetten, hetgeen in veel gevallen niet gewenst is. Aanrader is om deze functionaliteit in de class zelf te programmeren in plaats van deze 'handige' `__unset()` method te gebruiken.

`__call()`

De `__call()` magic method doet voor methodes hetzelfde als `__set()` en `__get()` voor properties doen. Als een niet bestaande method aangeroepen wordt, wordt deze automatisch doorgegeven aan de `__call()` magic method.

Code

```php
<?php
class User {
    public $username;

    public function __construct($username) {
        $this->username = $username;
    }

    public function __call($method, $args) {
        echo 'De method User::'.$method.' is niet gedeclareerd';
    }
}

$jan = new User('jan');
$jan->setEmail('jan@email-example.nl');
?>
```

Code: **output**

```
De method User::setEmail is niet gedeclareerd
```

Dit voorbeeld laat zien dat `__call()` inderdaad de aanroep van een niet bestaande method opvangt. Het is natuurlijk niet nuttig om deze foutmelding te echoën, sterker nog dan kun je beter PHP de foutmelding laten genereren door geen `__call()` te gebruiken.

Waar deze method wel van pas komt is bijvoorbeeld bij het gebruik van meerdere classes die van elkaar afhankelijk zijn. Hier geven we geen voorbeeld van maar er zijn situaties te bedenken waarin je een method probeert aan te roepen op het ene object en er intern met behulp van `__call()` de method van een ander object uitgevoerd wordt. Deze toepassingen gaan echter te ver om hier in deze handleiding te bespreken.

`__toString()`

De laatste magic method die we behandelen, is `__toString()`. Deze method wordt automatisch aangeroepen wanneer een object omgezet wordt in een string, bijvoorbeeld bij het echoën van een object:

Code

```php
<?php
$table = new Table();
echo $table;
?>
```

We zien hier een referentie naar de eerdere HTML tabel voorbeelden. We zouden de verschillende `draw()` methods in de `Table`, `Row` en `Cell` classes namelijk kunnen vervangen door deze `__toString()` magic method:

Code

```php
<?php
class Table {
    private $_rows;

    public function __construct() {
        $this->_rows = array();
    }

    public function append($row) {
        $this->_rows[] = $row;
    }

    public function __toString() {
        $output = '<table border="1">'.PHP_EOL;

        foreach($this->_rows as $row) {
            $output .= $row;
        }

        $output .= '</table>'.PHP_EOL;

        return $output;
    }
}

class Row {
    private $_cells;

    public function __construct() {
        $this->_cells = array();
    }

    public function append($cell) {
        $this->_cells[] = $cell;
    }

    public function __toString() {
        $output = '<tr>'.PHP_EOL;

        foreach($this->_cells as $cell) {
            $output .= $cell;
        }

        $output .= '</tr>'.PHP_EOL;

        return $output;
    }
}

```

```php
class Cell {
    protected $_content;

    public function __construct($content) {
        $this->_content = $content;
    }

    public function __toString() {
        return '<td>'.$this->_content.'</td>'.PHP_EOL;
    }
}

/* Procedurele code */
$cellA1 = new Cell('Dit is cel A1');
$cellA2 = new Cell('Dit is cel A2');

$rowA = new Row();
$rowA->append($cellA1);
$rowA->append($cellA2);
$rowA->append(new Cell('Dit is cel A3')); // Zo kan het ook!

$table = new Table();
$table->append($rowA);

// Weergave gaat nu met behulp van echo!
echo $table;
?>
```

Dit script geeft dezelfde output als het eerste voorbeeld van de HTML tabel die we maakten, alleen wordt deze nu op een andere manier gegenereerd. Zodra een object geëchoed wordt, wordt de `__toString()` method aangeroepen. In de code zien we dat de echo's vervangen zijn en dat de objecten nu daadwerkelijk een string retourneren in plaats van zelf voor de output te zorgen. Precies zoals het hoort.

Hiermee wil sluiten we dit hoofdstuk over magic methods af. Naast de methods die hier zijn beschreven, zijn er nog een aantal die niet genoemd zijn. De functionaliteit daarvan overstijgt het niveau van deze handleiding, deze zijn:

- `__sleep()` en `__wakeup()` - te gebruiken bij het serialiseren en unserialiseren van objecten
- `__autoload()` - te gebruiken om classes automatisch te laten
- `__clone()` - te gebruiken om een kopie van een object te maken

Slotwoord en referenties

Tot zover dan deze beginnershandleiding over het object georiënteerd programmeren. Eigenlijk raak je over dit onderwerp niet uitgepraat. Er zijn dan ook nog een aantal onderwerpen die in de toekomst aan deze handleiding toegevoegd zullen worden:

- Typehinting
- Design patterns (met name MVC)
- OOP Foutafhandeling
- Documentatie van code (PHPDoc)
- Autoload van classes
- PHP5 SPL

En het zou zomaar kunnen dat er nog onderwerpen aan deze lijst toegevoegd worden. Het blijft dus zeker de moeite waard om deze handleiding in de gaten te houden om te zien of er nieuwe onderwerpen verschenen zijn.

Voor het schrijven van deze handleiding is gebruik gemaakt van een aantal bronnen waaruit op sommige punten teksten of voorbeelden letterlijk overgenomen zijn. Een referentie naar die bronnen vind je hier:

- 'Object georiënteerd denken' door Erik Duindam
  http://corni.ps/8m0Nh (http://www.phphulp.nl/php/tutorials/8/632/)
- 'Object Oriënted PHP for Beginners' van www.killerphp.com
  http://corni.ps/8m0Ni (http://www.killerphp.com/tutorials/object-oriented-php/)
- 'Learn to create a PHP5 class' van GeekFile
  http://corni.ps/8m0Nj (http://www.sunilb.com/php/php-tutorials/php5-oops-tutorial-learn-to-create-a-php5-class)
- 'Object Oriënted Programming with PHP' van www.phpro.org
  http://corni.ps/8m0Nk (http://www.phpro.org/tutorials/Object-Oriented-Programming-with-PHP.html)

Je leert past echt OO programmeren als je het veel doet. Ervaring is écht de sleutel tot succes. Dus ga aan de slag en leer iedere dag!

Na het lezen van deze handleiding ben je misschien helemaal overtuigd van OOP. Dan geven we nog een tip mee: kijk goed af van anderen. Er zijn genoeg programmeurs die jou voorgegaan zijn, daar kun je veel van leren. Bovendien - en dat is nu net het mooie van OOP - is de kans groot dat je classes tegenkomt die je zelf goed kunt gebruiken. Het is nergens voor nodig om het wiel opnieuw uit te vinden!

Overige literatuur

- PHP beginnershandleiding http://www.phptuts.nl/view/39/
- SQL beginnershandleiding http://www.phptuts.nl/view/41/