Object Oriented Programming

19 Februari 2008, voor Codequest.nl

Niet iedere webdeveloper kent dit, maar heeft er toch zeker van gehoord: OOP, oftewel object georiënteerd programmeren. Met dit artikel zal ik proberen de beginnende webdeveloper te helpen met zijn weg naar het programmeren van OOP code.

Waarom gebruik je klassen?

Voordat we beginnen aan het schrijven van klassen, willen we natuurlijk eerst weten waarvoor klassen gebruikt worden. In dit hoofdstuk zal ik antwoord proberen geven op deze vraag.

De fietsenfabriek

We hebben het allemaal vast wel eens gehad: je loopt de kroeg uit en je begint met je sleuteltje in een fiets te steken. En zoals elke avond, blijkt iemand anders dezelfde fiets als jouw te hebben en staat jouw fiets 2 meter verder. Nu denk jij dat je weer pech hebt maar in een fabriek stonden nog veel meer van diezelfde fiets op een nog kleinere ruimte. Denk je dat deze fietsen allemaal opnieuw ontworpen zijn voordat ze van de band kwamen rollen?

Toegepast in het programmeren

Dit voorbeeld zou je ook kunnen toepassen in het programmeren. Stel je hebt 1000 textuele presentaties van een fiets nodig. Deze moeten allebei een kleur en remtype kunnen hebben en hiernaast nog kunnen fietsen ook. Je kunt je voorstellen hoeveel code je hiervoor nodig hebt wanneer je al deze fietsen 1 voor 1 gaat programmeren. Ook brengt het gebruik van een array met al deze fietsen wat ongemak met zich mee. Deze zul je dan telkens door moeten geven aan de functies e.d..

Conclusie

De bovengenoemde fietsen willen wij dus niet één voor één openemen in de code. Het zou natuurlijk een stuk makkelijker zijn wanneer we kunnen zeggen wat een fiets moet hebben en kunnen. Als we deze code dan 1000 keer kopiëren zijn we er ook. Dit is precies wat je met klassen kunt doen. Je kunt een 'blauwdruk' maken van iets en er later daadwerkelijke objecten van maken.

- Een Fiets moet een kleur hebben
- Een Fiets moet een remtype hebben
- Een Fiets moet kunnen rijden
- fiets1 is een Fiets
- fiets2 is een Fiets

We hebben nu 2 fietsen, terwijl we maar één keer gezegd hebben wat een fiets inhoud. Klassen gebruik je dus wanneer je code hebt die je herhaaldelijk gebruikt.

De basis van OOP

Nu we weten waarvoor OOP gebruikt wordt, is het tijd om hier eens naar te kijken. Laten we eens kijken naar de opbouw van een klasse:

```php
<?php
class Fiets {
    // Eigenschap
    private $kleur;

    // Constructor
    // In andere talen kan dit de naam van de klasse als methode zijn
    public function __construct() {
    }

    // Functionaliteit
    public function getKleur() {
        return $this->kleur;
    }

    // Functionaliteit
    public function setKleur($nKleur) {
        $this->kleur = $nKleur;
    }
}
?>
```

Zoals je ziet bestaat een klasse uit eigenschappen (zoals hierboven de kleur) en methodes (zoals hierboven getKleur() en setKleur()). De methodes zijn de functionaliteiten binnen je klasse. Dit is het stadium waarin we nog steeds spreken van een klasse. Wat vaak gebeurd is dat mensen een klasse en een object met elkaar verwarren. Een klasse zou je kunnen zien als een blauwdruk van een 'iets'. Hierboven hebben wij bijvoorbeeld gedefinieerd waar een fiets uit bestaat. Echter hebben we nu nog niet echt een fiets. Deze moet eerst gemaakt worden uit deze blauwdruk. Een object is dus een instantie van een klasse.

Terugkomend op de klasse zien we nog iets opmerkelijks. Voor de eigenschappen staat het woord 'private' en voor de methoden 'public'. Dit heeft te maken met de toegang die je tot deze gegevens of functionaliteiten hebt. Je hebt in totaal 3 niveaus: public, private en protected. Hieronder zal ik een object maken van de klasse hierboven en hiermee uitleggen wat deze niveaus inhouden.

```php
<?php
// Hieronder maken we een instantie van de klasse 'Fiets'. Je zou
// de zin hieronder letterlijk kunnen oplezen om te zien wat er
// gebeurd: "$fiets_object is new Fiets." oftewel "$fiets is een
// nieuwe Fiets".
// Dit betekend dus ook dat, omdat $fiets_object een instantie is
// van de klasse Fiets, $fiets_object een object is.
$fiets_object = new Fiets();

// Nu gaan we kijken of het volgende lukt:
$fiets_object->kleur = 'blauw'; // Verander de kleur van de fiets.
// Is het gelukt? Waarschijnlijk krijg je nu een foutmelding. De
// indicator 'private' geeft aan dat de eigenschap 'kleur' binnen
// de klasse alleen aangepast mag worden BINNEN DE KLASSE ZELF. De
// kleur kan dus alleen aangepast worden vanuit de methodes die
// binnen de klasse gedefinieerd zijn.

// Natuurlijk kunnen we nu wel raden waar 'public' voor staat:
$fiets_object->setKleur('blauw'); // Roep de functie setKleur() aan.
// Werkt dit? Ja! Public functies en eigenschappen mogen vanuit overal
// aangeroepen worden. Het is publiekelijk, voor alle code.
```

```php

// En wat denk je nu dat het volgende weergeeft:
echo $fiets_object->getKleur(); // Is de kleur veranderd?
// Bovenstaande geeft 'blauw' weer; de kleur is veranderd. Hierboven
// hebben we gezegd dat private variabellen alleen veranderd mogen
// worden binnen de klasse. Aangezien dit gebeurde in de methode
// setKleur() uit DEZELFDE klasse, werd dit toegestaan.
?>
```

Nu zijn we nog 1 niveau vergeten: protected. Hier zal ik later in dit artikel op terug komen.

We hebben nu gezien dat je public en private eigenschappen en methoden hebt. In OOP is het gebruikelijk om de eigenschappen altijd private te maken en deze te manipuleren met get- en set-functies. Dit is omdat er voor de eigenschappen vaak voorwaarden gelden, en de klasse dan zelf verantwoordelijk is voor het nakomen hiervan (de eigenschap kan immers niet van buitenaf veranderd worden). De methodes bevatten dan meestal wat checks die uitgevoerd worden alvorens de eigenschap aangepast wordt. Bij deze opzet worden deze functies dan ook wel de *getters* en *setters* genoemd.

Overerving

Nu we weten hoe we klassen moeten maken, kunnen we OOP pas echt handig inzetten. Het is namelijk mogelijk de klassen uit te breiden naar nieuwe klassen. Stel we hebben een auto en een motorfiets. Deze hebben allebei een moter. Toch heeft de auto 4 wielen en de motorfiets maar 2. Hoe pakken we dit aan? Laten we eerst eens kijken naar wat deze 2 gemeen hebben.

```php
<?php
class MotorVoertuig {
    // Staat het voertuig aan, ja of nee?
    protected $ingeschakeld;

    // Start het voertuig
    public function starten() {
        $this->ingeschakeld = true;
    }

    // Zet het voertuig weer uit
    public function stoppen() {
        $this->ingeschakeld = false;
    }
}
?>
```

Wij hebben nu een blauwdruk voor een object met een motor. Nu gaan we 2 nieuwe klassen maken welke een extension (uitbreiding) zijn van deze klasse.

```php
<?php
class Auto extends MotorVoertuig {
    // Een auto heeft 4 wielen
    private $aantalWielen = 4;
}
?>
```

```php
<?php

class Motorfiets extends MotorVoertuig {
    // Een motorfiets heeft 2 wielen
    private $aantalWielen = 2;
}
?>
```

We hebben nu 2 klassen welke allebei een uitbreiding zijn van de MotorVoertuig klasse. Het resultaat hiervan is dus dat zowel de klasse Auto als de klasse Motorfiets gebruik kunnen maken van de eigenschappen en functies die deze klasse biedt. Je hebt dus de gedeelde eigenschappen en methodes apart, waardoor als deze veranderen je deze voor alle motorvoertuigen in 1 keer kunt aanpassen.

Private en protected bij overerving

Wat je nu nog tegoed hebt is de uitleg van het protected toegangs niveau. Ook zal ik het verschil tussen private en protected in deze duidelijk proberen te maken.

De methoden zijn allemaal public, dus sowieso vanuit overal aan te roepen. De private en protected eigenschappen zijn als volgt te benaderen:

```php
<?php
class Motorfiets extends MotorVoertuig {
    public function toegang() {
        // Onderstaande zal werken. De eigenschappen van
        // MotorVoertuig zullen aan te roepen zijn alsof ze deel
        // uitmaken van de klasse Motorfiets. Omdat deze eigenschap
        // protected is, heeft de klasse Motorfiets ook toegang tot
        // deze eigenschap. Ook in de klasse Auto zal dit werken.
        if($this->ingeschakeld)
            return true;
        // Zou het echter zo zijn dat de eigenschap ingeschakeld in
        // de klasse MotorVoertuig private zou zijn, dan zou deze
        // niet vanaf de overervende klassen te benaderen zijn. Met
        // private zijn de eigenschappen alleen beschikbaar vanaf
        // dezelfde 'blauwdruk' (klasse).
    }
}
?>
```

Wat weten we tot nu toe? We weten wat public, private en protected is en hiernaast kunnen we klassen uitbreiden. Laten we nu eens kijken naar het volgende voordeel van OOP.

Implementeren

Een andere voordeel dat OOP biedt is interfaces. Een interface is een soort vooropgezette definitie voor één of meerdere klasses. Wat je eigenlijk doet, is een set methoden definiëren die je in de implementerende klassen moet invullen. Stel, we maken de volgende interface.

```php
<?php
interface Voertuig {
    public function inschakelen();
    public function uitschakelen();
}
?>
```

We hebben nu een interface 'Voertuig'. Deze beschrijft dat we 2 methoden moeten implementeren, t.n. inschakelen() en uitschakelen(). Nu maken we twee klassen die deze interface implementeren:

```php
<?php
class Auto implements Voertuig {
    private $motorLoopt;

    // Deze functie MOETEN we dus hebben
    // Anders krijgen we een fatal error
    public function inschakelen() {
        $this->motorLoopt = true;
    }

    // Idem voor deze functie
    public function uitschakelen() {
        $this->motorLoopt = false;
    }
}
?>
```

```php
<?php
class Fiets implements Voertuig {
    private slotOpen;

    public function inschakelen() {
        $this->slotOpen = true;
    }

    public function uitschakelen() {
        $this->slotOpen = false;
    }
}
?>
```

Zoals je ziet hebben we nu dus 2 klassen welke allebei de functies inschakelen() en uitschakelen() hebben. Echter geven beide klassen hun eigen implementatie aan deze functies. Nu zul je misschien denken: waarom niet gewoon 2 aparte klassen?

We hebben de volgende klasse:

```php
<?php
class VoertuigHandler {
    public function addVoertuig($voertuig) {
        $voertuig->inschakelen();
    }
}
?>
```

Hoe weet je nu zeker dat $voertuig ook daadwerkelijk de functie inschakelen() bevat? Dit kun je dus handmatig opvragen, maar waarom het wiel opnieuw uitvinden. Dit is iets wat je ook kunt laten doen:

```php
<?php
class VoertuigHandler {
    public function addVoertuig(Voertuig $voertuig) {
        $voertuig->inschakelen();
    }
}
?>
```

Met een kleine toevoeging voor het argument voor de methode addVoertuig(), laten we de functie weten wat er precies verwacht wordt in de methode, namelijk een klasse van het type 'Voertuig', of een implementerende klasse hiervan. Bovenin dit hoofdstuk hebben we beschreven hoe een klasse die 'Voertuig' gebruikt eruit moet zien, namelijk met de functies inschakelen() en uitschakelen(). Wanneer de methode addVoertuig() hierboven aangeroepen wordt, moet de parameter dus wel een object zijn die ook de methode inschakelen() bevat (anders ontstaat er een foutmelding voor een verkeerd argument). Dit fenomeen heet typehinting.

Naast het toepassen van typehinting met interfaces kan dit ook met klassen:

```
1. <?php
2. class VoertuigHandler {
3.     public function addVoertuig(Auto $auto) {
4.         $auto->inschakelen();
5.     }
6. }
7. ?>
```

Er moet nu een instantie van de klasse of interface 'Auto' meegegeven worden. In ons voorbeeld is dat dus de klasse die de interface 'Voertuig' implementeert.

Abstracte klassen

Naast het implementeren van een klasse, wil je soms ook het beste van 2 werelden: implementeren én uitbreiden uit 1 klasse. Dit is mogelijk met abstracte klassen. Omdat we al weten wat uitbreiden en implementeren inhoudt zal ik dit hoofdstuk kort houden.

We hebben de volgende klasse:

```
1. <?php
2. abstract class MotorVoertuig {
3.     protected $ingeschakeld;
4.
5.     public function isIngeschakeld() {
6.         return $ingeschakeld;
7.     }
8.
9.     abstract public function rijden();
10. }
11. ?>
```

In bovenstaande klasse zie je dat een deel al ingevuld is (er is een eigenschap $ingeschakeld en een methode isIngeschakeld()), en er nog een deel ingevuld moet worden (de methode rijden()). Met bovenstaande klassen kun je dus motorvoertuigen maken die sowieso ingeschakeld kunnen zijn, maar allen een aparte manier van rijden hebben.

Voorbeeld:

```php
<?php
class Fiets extends MotorVoertuig {
    public function rijden() {
        // Trappers bewegen e.d.
        echo "trap, trap, trap";
    }
}
?>
```

```php
<?php
class Auto extends MotorVoertuig {
    public function rijden() {
        // Motor laten werken e.d.
        echo "vroem, vroem, vroem";
    }
}
?>
```

Zoals je ziet kun je met abstracte klassen dus een soort van interface en uitbreidbare klasse in 1 maken. Ook dit is weer erg handig wanneer je typehinting (zie vorige hoofdstuk) gebruikt. Het voordeel is dat je nu als het ware op je 'interface' al gemeenschappelijke eigenschappen of methodes kunt opnemen.

Statische klassen

Binnen het object georiënteerd programmeren kun je ook statische eigenschappen op methodes toevoegen. Deze eigenschappen of methoden zijn, zoals het wordt het eigenlijk al zegt, statisch. Deze kun je niet gebruiken binnen een instantie van een klasse, maar roep je aan via een :: (*Paamayim Nekudotayim*).

Voorbeeld:

```php
<?php
class Zon {
    private static $temperatuur;

    public static function getTemperatuur() {
        // Let op: ook de eigenschappen roepen we anders aan.
        // Je hebt GEEN instantie, dus $this->temperatuur zal niet
        // werken. Binnen static methoden roep je andere componenten
        // van de eige klasse aan met self::$eigenschap of
        // self::methode().
        return self::$temperatuur;
    }
}
?>
```

Je hebt maar 1 zon. Van de zon is dus helemaal geen instantie nodig. Je kunt deze klasse als volgt aanroepen.

```php
<?php
// De klass is statisch, dus een instantie maken mag niet. Tevens
// moeten we, omdát er geen instantie is, Klasse::methode()
// (oftewel de Paamayim Nekudotayim) gebruiken.
$temperatuur = Zon::getTemperatuur();
?>
```

Het voordeel van dit soort klassen, is dat ze altijd beschikbaar zijn. Dit is een stuk netter en overzichtelijker dan telkens instanties te moeten doorgeven, of te werken met globale variabelen. Hiernaast zit er nog een ander voordeel aan het gebruik van statische eigenschappen. De volgende klasse representeert een communicatie laag naar een database.

```php
<?php
class Database {
    // Een statische eigenschap waarin een instantie wordt opgeslagen
    private static $instantie;

    // Een private constructor om het direct maken van een instantie
    // tegen te gaan
    private function __construct() { }

    // Een public methode om de instantie te verkrijgen
    public static function getInstantie() {
        if(self::$instantie == null) {
            self::$instantie = new Database();
        }
        return self::$instantie;
    }
}
?>
```

Wanneer je een database hebt wil je niet dat er voor alle queries die je doet een nieuwe instantie van je Database klasse komt. Ook wil je niet telkens de instantie van het database object doorgeven. Hiervoor zijn statische eigenschappen een mooie oplossing. In bovenstaande klassen hoef je namelijk geen instantie aan te maken; sterker nog, dit mag niet eens. De volgende code geeft een instantie van de Database klasse terug, is overal in de code aan te roepen en zorgt er ook voor dat er altijd maar één instantie bestaat:

```php
// Instantie van de Database klasse aanmaken
$database = Database::getInstantie();

// Nu kun je de reguliere niet-statische functies uitvoeren, bijv.
// het doen van een query.
$database->query();
```

Conclusie

Dit waren dan de beginselen van het object georiënteerd programmeren. We hebben het gehad over public, private en protected eigenschappen en methodes, overerving, implementeren en abstracte en statische klassen. Waarschijnlijk zal het 'netjes' (tussen aanhalingstekens, ieder zijn smaak natuurlijk) programmeren van je code, zeker in het begin, erg moeilijk zijn. Ik raad je aan hier gewoon goed mee te oefenen en het ook zeker niet op te geven. Wanneer je dit in de vingers krijgt, zul je zien dat dit je uiteindelijk een hoop tijd zal besparen en je zelfs ook in andere projecten code kunt hergebruiken. Rest mij niets anders dan je succes te wensen met het verdere programmeren. Tot het volgend artikel!

Thanks goes out to:

Lees deze artikelen ook eens wanneer je meer over dit onderwerp wilt weten, of de punten besproken in dit artikel vanuit een ander oogpunt wilt zien:

- http://wiki.phpfreakz.nl/Object_Oriented_Programming
- http://wiki.phpfreakz.nl/OOP_Toepassen