

Zeki Çocuklar için

PHP'de Nesne Tabanlı Programlama'ya Hızlı Giriş Kılavuzu

Altan TANRIVERDİ & Tümay ÇEBER
<http://javam.org>

Kılavuzu mümkün olduğunca kısa tutmak için önsöz bir sonraki cümle ile bitiyor. Bu konuyu anlamak için PHP'nin temel yapıtaşları sayılan fonksiyonlar ve değişkenler konusunda net bir bilgiye sahip olmalısınız.

Sınıf yaratmak için anahtar kelime `class`'tır. `sinif.php` adında bir dosya yaratalım:

```
<?php
class otomobil
{
}
?>
```

Böylece otomobil adında bir sınıf yarattık. Şimdi hemen data ekleyelim:

```
<?php
class otomobil
{
    var $marka;
}
?>
```

Görüldüğü gibi şimdi de `$marka` adında bir değişken bilgisini sınıfımıza ekledik. Sırada fonksiyonlar var:

```
<?php
class otomobil
{
    var $marka;
    function marka_tanimla($yeni_marka)
    {
        $this->marka = $yeni_marka;
    }
    function marka_goster()
    {
        return $this->marka;
    }
}
?>
```

Görüldüğü gibi iki fonksiyon ekledik `marka_tanimla()` ve `marka_goster()`. Sanırım fonksiyonların görevlerini adlarından tahmin ediyoruz. Ama yeni bir kavram görüyoruz: `$this->`. `$this->` ifadesi geçerli nesneyi işaret eder. Diğer bir deyişle, `$this->` özel bir referans değişkenidir. `$this->` ifadesini bir değişkene erişirken ve mevcut sınıfın diğer fonksiyonlarına ulaşırken kullanırız. Örneği açıklamak gerekirse; `$this->marka = $yeni_marka;` ile `$marka` adlı değişkenimize `marka_tanimla()` fonksiyonuna gelen `$yeni_marka` değerini atamış olduk. `return $this->marka;` ile ise `$marka` değişkenini doğrudan `return` etmiş oluyoruz.

Buraya bir not düşerek devam edelim. Nesne tabanlı PHP için fonksiyonlara "**method**", değişkenlerde "**özellik**" diyeceğiz.

Şimdi **index.php** dosyası yaratalım ve **sinif.php**'yi **include** edelim. Sınıflarla, normal PHP kodlarını aynı PHP dosyası içerisine yerleştirmeyin.

Not: Başka bir php dosyasını eklemenin iki yöntemi vardır. Aralarında ki fark ise hangisini kullanacağımızı belirler.

include (ve include_once)

require (ve require_once)

Include; dosyayı eklerken, dosya bulunamadığı takdirde sadece uyarı vererek ana kodumuzun çalışmasına kaldığı yerden devam etmesine izin verir.

Require ise dosya bulunamadığında “fatal error” vererek kodu durdurmaktadır. Require genel olarak olmazsa olmaz dosyalar için kullanılır.

```
<?php
include 'sinif.php';
?>
```

Sıra geldi nesne tabanlı programlamanın temeli olan nesne'yi yaratmaya. Bu işleme **örnekleme**(instantiation) denir.

```
<?php
include 'sinif.php';
$ford = new otomobil();
?>
```

Burada **\$ford** "**özelligi**" (değişkeni)'ni kulp olarak düşünün ve otomobil sınıfında çaydanlık. Kulptan tutarak çaydanlığı kontrol etmiş oluyoruz. Onu kaldırıyoruz, eğiyoruz vs.

Kodu çalıştırdığınızda sonuç döndürmeyecektir çünkü PHP'ye hala yaratılan nesne ile ne yapması gerektiğini söylemedik. Buradaki "**new**" ifadesini sınıf dışından bir nesne yaratmak için kullanıyoruz.

Doğru kullanım şekilleri:

```
$ford = new otomobil();
$ford = new otomobil;
```

Yanlış kullanım şekli:

```
$ford = new 'otomobil';
```

Şimdi üç farklı otomobil markası yaratalım:

```
<?php
include 'sinif.php';

$ford = new otomobil();
$fiat = new otomobil();
$opel = new otomobil();
```

```
$ford -> marka_tanimla('FORD');
$fiat -> marka_tanimla('FIAT');
$opel -> marka_tanimla('OPEL');
?>
```

Markaları görüntüleyelim:

```
<?php
include 'sinif.php';

$ford = new otomobil();
$fiat = new otomobil();
$opel = new otomobil();

$ford -> marka_tanimla('FORD');
$fiat -> marka_tanimla('FIAT');
$opel -> marka_tanimla('OPEL');

echo $ford -> marka_goster() . "<br />";
echo $fiat -> marka_goster() . "<br />";
echo $opel -> marka_goster() . "<br />";
?>
```

Nesne özelliklerine erişim için methodları kullanmak zorunda değilsiniz, doğrudan -> ve değişken ismi ile erişebilirsiniz. Örneğin: `echo $fiat->marka;` size **FIAT** sonucunu dönderecektir ancak bu kapsamlı projelerde karışıklıklara neden olabileceğinden iyi bir yazılım şekli sayılmaz ve methodlar kullanılarak bu işlemin yapılması daha düzgün bir yoldur.

Tüm nesnelere "**constructor**" olarak ifade edilen özel bir methoda sahip olabilir. Bir `__construct()` fonksiyonu yaratırsanız sınıfınızdan nesne yarattığınız zaman bu method otomatik olarak çağrılacaktır.

```
<?php
class otomobil
{
    var $marka;
    function __construct($marka_adi)
    {
        $this->marka = $marka_adi;
    }
    function marka_tanimla($yeni_marka)
    {
        $this->marka = $yeni_marka;
    }
    function marka_goster()
    {
        return $this->marka;
    }
}
?>
```

__construct() methodunu doğrudan çalıştırmak için

```
<?php
include 'sinif.php';
$jaguar = new otomobil('JAGUAR');
?>
```

Görüldüğü gibi nesnemizi yaratırken, marka_tanimla() methodunu kullanmadan otomobil markasını __construct() yardımı ile tanımlamış olduk. Kontrol için:

```
<?php
include 'sinif.php';
$jaguar = new otomobil('JAGUAR');
echo "En sevdiğim otomobil markası: " . $jaguar->marka_goster();
?>
```

Not: __construct() yerine sınıfımızın adını taşıyan bir fonksiyonda aynı işlevi görür.

Ör. :

```
<?php
class otomobil {
    function otomobil($marka_adi)
    {
        $this->marka = $marka_adi;
    }
}
?>
```

Not: __destruct() methodu ise sınıfımız yok edilmeden önce çalıştırılacak kodları içerir.

Ayrıca diğer sihirli methodları (*magic methods*) (__call, __callStatic, __get, __set, __isset, __unset, __sleep, __wakeup, __toString, __set_state ve __clone) sonraki kılavuz güncellemelerinde ayrıntılı olarak işleyeceğiz.

Zend Studio kullananlar, **sinif.php** içerisinde `var $marka;` satırının uyarı verdiğini farkedeceklerdir. Uyarıda public/private/protected niteleyicilerini (modifiers) kullanmamız önerilmektedir. ZS'nin önerdiği bu kavramlar özelliklere erişimi kontrol eden, sınırlandıran tanımlamalardır. Bu tanımlamalar ile sınıfı kullanan kodların bilgiye erişimi denetlenir. Bilgi güvenliği ve oluşabilecek hataların minimize edilmesi ana amaçtır. Bu niteleyeciler **public**, **protected**, **private** (ayrıca **static** niteleyicisini birazdan inceleyeceğiz) olarak sıralanır. Bunlar arasında **public** varsayılan niteleyicidir, yani niteleyeci kullanmadığınız takdirde PHP bunu **public** olarak hesaba katacaktır. Örneğimiz üzerinden devam edelim:

```
<?php
class otomobil
{
    var $marka;
    public $model;
    protected $fiyat;
    private $satis_miktari;

    function __construct($marka_adi)
    {
        $this->marka = $marka_adi;
    }
    function marka_tanimla($yeni_marka)
    {
        $this->marka = $yeni_marka;
    }
    function marka_goster()
    {
        return $this->marka;
    }
}
?>
```

Görüldüğü gibi örneğimiz otomobil galerisi için yazılabilecek bir uygulamaya dönüşmeye başladı. Galeri sahibi için otomobilin çeşitli özelliklerini tutmak mümkün iken bunlar arasında değerine göre bilgiyi korumayı da sağlamış oluyor. Anlamayı kolaylaştırmak için kabaca örneklendirelim.

Alfa Romeo markasının Brera modelini 60000 YTL'ye satıyoruz ve bu yıl bunlardan 2 adet sattık. Brera modelini sattığımızın herkes tarafından bilinmesinin bir sakıncası yok, ancak fiyatını herkes bilsin istemeyiz, çünkü koşullara, faiz oranlarına, müşteriye göre fiyat değişiklik gösterebilir. Bu noktada onu **protected** olarak niteliyoruz. Peki kaç adet sattığımız? Bunu özellikle rakipler başta olmak üzere herkesten korumak için tamamen **private** olarak gizleyebiliriz.

Bir ek not ekleyelim; **var \$marka;** satırı aslında PHP tarafından **public \$marka;** olarak algılanmaktadır. **var**'ı **public**'e çevirip kaydettiğinizde ZS'nin uyarıyı kaldırdığını göreceksiniz.

Bir özelliği **private** olarak tanımladığımızda bu özelliğe sadece mevcut sınıf içerisinde erişilebilir.

protected olarak tanımladığımızda ise ancak mevcut sınıf ve onu genişleten sınıflardan erişilebilir.

public'te ise bir erişim kısıtlaması yoktur. Konuyu daha iyi anlamak için özelliklere varsayılan değerler atayalım. Örneğin:

```
<?php
class otomobil
{
    public $marka = "Alfa Romeo";
    public $model = "Brera";
    protected $fiyat = "60000";
    private $satis_miktari = "2";

    function marka_tanimla($yeni_marka)
    {
```

```

        $this->marka = $yeni_marka;
    }
    function marka_goster()
    {
        return $this->marka;
    }
}
?>

```

Bu değerleri **index.php** ile almaya çalışalım:

```

<?php
include 'sinif.php';
$degeroku = new otomobil();
echo "Marka: " . $degeroku->marka;
echo "Model: " . $degeroku->model;
echo "Fiyat: " . $degeroku->fiyat;
echo "Satış Miktarı: " . $degeroku->satis_miktari;
?>

```

Bu kodu çalıştırdığımızda Zend kullananlar debug output'ta şöyle bir sonuç alacaklardır:

```

Marka: Alfa RomeoModel: Brera<br />
<b>Fatal error</b>: Cannot access protected property otomobil::$fiyat in <b>/home/altan/Zend/
workspaces/DefaultWorkspace/kilavuz/oop/index.php</b> on line <b>6</b><br />

```

Görüldüğü gibi Marka ve Model değerleri gelirken, sınıf dışından **protected** veya **private** bir özelliğin değerini alamıyoruz. Tüm bunlar methodlar (fonksiyonlar) için de geçerlidir.

Şimdi genişletilmiş (extended) sınıflara gelelim. Herhangi bir sınıfı **extends** ifadesiyle genişletebiliriz. Böylece sınıfa yeni sınıflar entegre etmiş olacağız. **sinif.php** için önceki örneğimize dönelim:

```

<?php
class otomobil
{

    public $marka;

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

    public function marka_tanimla($yeni_marka)
    {
        $this->marka = $yeni_marka;
    }

    public function marka_goster()
    {

```

```

        return $this->marka;
    }
}

// şimdi otomobil sınıfını genişletelim:

class ucuzOtomobil extends otomobil
{
    //...ifadeler...
}
?>

```

Böylelikle otomobil sınıfına bağımlı bir ucuzOtomobil sınıfı oluşturmuş olduk. ucuzOtomobil otomatik olarak otomobil sınıfının **private** harici tüm method ve özelliklerine sahip oldu. Genişletilmiş sınıflarda daha önce kullanılmış bir method ismini, işleyiş değiştirme amacınız yoksa kullanmayın. Şimdi kodlarımızın son halini verelim:

index.php:

```

<?php
include 'sinif.php';

$otomobil = new otomobil('Alfa Romeo');
echo "Otomotiv: " . $otomobil->marka_goster();
echo "<br />";

$ucuzotomobil = new ucuzOtomobil('Ford');
echo "Ucuz otomotiv: " . $ucuzotomobil->marka_goster();
?>

```

Zend Debug Output:

Otomotiv: Alfa Romeo
Ucuz otomotiv: Ford

Görüldüğü gibi ucuzOtomobil sınıfı içerisinde `__construct()` veya `marka_goster()` fonksiyonları olmadığı halde bunları otomobil sınıfından yararlanarak kullanabiliyor.

Sıra geldi method işleyişlerini değiştirmeye... Bazı durumlarda ana sınıftaki bir method'un işleyişini değiştirmeye ihtiyaç duyarsınız. Bunu yapmanın yolu, ana sınıftaki method adı ile genişleten sınıf içerisinde bir method açmaktır.

```

<?php
class otomobil
{
    public $marka;

    public function marka_tanimla($yeni_marka)
    {
        $this->marka = $yeni_marka;
    }

    public function marka_goster()

```

```
        {
            return $this->marka;
        }
    }
}
```

```
class ucuzOtomobil extends otomobil
{
    public function marka_tanimla($yeni_marka)
    {
        $this->marka = trim(strtoupper($yeni_marka));
    }
}
?>
```

Görüldüğü gibi marka_tanimla() adlı bir fonksiyonu genişleten fonksiyonumuza da eklemiş olduk. Test edelim:

index.php:

```
<?php
    include 'sinif.php';

    $otomobil = new otomobil();
    $otomobil -> marka_tanimla(' alFa RoMeo ');

    echo "Otomotiv: " . $otomobil->marka_goster();
    echo "<br />";

    $ucuzotomobil = new ucuzOtomobil();
    $ucuzotomobil -> marka_tanimla(' alFa RoMeo ');
    echo "Ucuz otomotiv: " . $ucuzotomobil->marka_goster();
?>
```

Zend Debug Output:

Otomotiv: alFa RoMeo
Ucuz otomotiv: ALFA ROMEO

Görüldüğü gibi genişleten fonksiyonuz ile trim ve strtoupper düzenlemeleri yapmış olduk.

Şimdi sıra :: ifadesinde. İngilizce'de "Scope Resolution Operator" olarak tanımlanıyor. Türkçe'de genelde erim veya çözünürlük operatörü olarak kullanılıyor. Ana sınıfımızdaki özellik ve methodlara erişim için kullanılan bir operatördür. Örnekleyelim:

sinif.php:

```
<?php

class otomobil
{
    static $fiyat="60000";
}
```

```
class ucuzOtomobil extends otomobil
{
    public function operator_ornegi()
    {
        return $this->fiyat = otomobil::$fiyat * 3;
    }
}
?>
```

index.php:

```
<?php
include 'sinif.php';

$ucuzotomobil = new ucuzOtomobil();
echo "Yeni Fiyat: " . $ucuzotomobil->operator_ornegi();
?>
```

Çıktı: Yeni Fiyat: 180000

Görüldüğü gibi yeni bir kavramımız var **static**. **static** ile tanımladığımız sınıf üyelerine örnekleme (instantiation) yapmadan ve :: aracılığıyla ulaşabiliriz. Örnekleme yaptığımız takdirde ise **static** üyelere ulaşamayacağız.

Şimdi birde fonksiyon örneği yapalım:

sinif.php:

```
<?php
class otomobil
{
    static $fiyat="60000";
    public function dortKati()
    {
        return self::$fiyat * 4;
    }
}

class ucuzOtomobil extends otomobil
{
    public function operator_ornegi()
    {
        return $this->fiyat = otomobil::dortKati();
    }
}
?>
```

index.php:

```
<?php
    include 'sinif.php';

    $ucuzotomobil = new ucuz0tomobil();
    echo "Yeni Fiyat: " . $ucuzotomobil->operator_ornegi();
?>
```

Çıktı: Yeni Fiyat: 240000

Görüldüğü gibi `self::` ile mevcut sınıfın bir üyesini çağırabiliyoruz.

ÖNEMLİ NOT: Bu kılavuz düzenli olarak güncellenecektir. En güncel hali için <http://javam.org> adresini takip ediniz.