PHP

Object Oriented Programming (OOP)

Emrullah TANIMA

Hakkımda

17 Mayıs 1994 Ordu/Kumru doğumluyum. İlk ve orta eğitimimi doğup büyüdüğüm ilçede bitirdim. Lise eğitimimi İstanbul Hadımköy Toki Lisesinde bitirdim. Ordu Üniversitesi Bilgisayar Programcılığı mezunuyum.

Çok küçük yaşlardan beri bilgisayar, internet ve müzikle uğraşıyorum. Ağırlık olarak Web Yazılım, Nesne Tabanlı Programlama ve VTYS (Veritabanı Yönetim Sistemleri) alanlarında proje üretiyor veya mevcut projeleri geliştiriyor ya da teknik destek sunuyorum. Web yazılım olarak 7 yıldır PHP kullanıcısı, Nesne Tabanlı olarak ise 4 Yıldır Java kullanıcısıyım. Üniversite yıllarında birkaç geliştirici arkadaşla ortak bir yazılım şirketi kurup daha sonra bazı sebeplerden dolayı kepengi kapattık. Şu an freelancer olarak çalışıyorum ve böyle daha mutluyum.

Müzik dalından bahsedecek olursak genelde yaz aylarında kafe, türkü bar, restorant ve düğünlerde şarkı söylüyorum. Gitar, Bağlama ve Org enstrümanlarıyla aram çok iyidir. Elimden geldiği kadar ve işlerden fırsat buldukça boş zamanlarımda müzikal anlamda bir şeyler yapmaya çalışıyorum.

Üzerimde emeği geçen herkese başta üniversite hocalarım ve özel kurs hocalarım olmak üzere herkese ayrı ayrı teşekkür ediyorum.

Tüm sorularınız ya da herhangi bir şey için info@emrullahtanima.com, emrtnm@gmail.com adresinden yada sosyal medya hesaplarımdan bana ulaşabilirsiniz.

Önsöz

Kişisel bloğum üzerinde yayınladığım yazıların site üzerinde çok dağınık durması araya başka konuları girmesi gibi sebeplerden dolayı bu yazıları bir kitap haline getirmek ve bu konuda ki kaynak sıkıntısına bir nebze de olsa katkıda bulunmak istedim.

Blog üzerinde ki içeriklerden farklı olarak ufak tefek değişiklikler ve eklemeler yaptım. Konuyu daha iyi özümseyeceğinizi düşündüğüm birkaç tane de örnek ekledim.

Kitap içeriğinde ki yazım ve imla hataları, kod hataları ya da yanlış olduğunu düşündüğünüz her hangi bir şeyi bana iletirseniz çok memnun olurum. Böylelikle amacına daha uygun bir kitap olmuş olur ve yanlış bilgi vermenin de önüne geçmiş oluruz.

Faydalı olması dileğiyle...

Bu kitap kimler için ?

Bu kitap PHP Object Oriented Programming Türkçe manasıyla nesne tabanlı programlama hakkında bilgiler içerir. Kitapta anlatılan konuları anlayabilmek için genel programlama bilgisine ve yeteğine sahip olmak, PHP temel yazım kuralları(syntax)nı bilmek yeterlidir.

Hiçbir zaman PHP diline ait konuların; temel php dersleri, orta seviye php dersleri, ileri seviye php dersleri gibi saçma sapan kategorilere ayrılmasından yana değilimdir. Zira bir programlama dilinin öğrenilmesi gerekiyor ise bütünüyle öğrenilmesi taraftarıyım.

İçindekiler

1.1 Giriş

OOP Object Oriented Programming kelimelerinin baş harflerinden oluşan bir programlama teknolojisidir. Türkçe manasıyla söyleyecek olursak Nesne Tabanlı Programlamadır. OOP teknolojisi kullanılan programlama dillerine en güzel örnek Visual Basic ve JAVA'dır. OOP teknolojisi kullanılan programlama dilleri tabi ki bu iki güzel örneğimiz ile sınırlı değil. Örneğin: Objective-C, C#, C++, Python ve tabii ki PHP.

Eğer yukarıdaki dillerden herhangi birini biliyorsanız PHP ile nesne tabanlı programlama mimarisine aşina olmanız çok kolay olacaktır. PHP nesne tabanlı programlama mimarisine tam olarak PHP 5 ile geçmiştir. PHP4 de obje (nesne) yapısı vardı ancak ihtiyaçları pek karşılamıyor ve modern yapıya pek ayak uyduramıyordu. Ancak PHP5 ile gerçek manada bir OOP mimarisi PHP diline eklendi.

PHP OOP mimarisini kullanarak daha profesyonel, daha esnek ve daha geliştirilebilir projeler oluşturabiliriz. Günümüzde artık tüm web teknolojilerinde OOP mimarisi kullanılıyor. Profesyonel olarak oluşturulan projeler bile OOP mimarisine dönüştürülüyor. Sizin de bugünden itibaren OOP mimarisine geçmemeniz için hiç bir sebep yok. OOP mimarisinin sizi cezbedeceğini düşündüğüm bir kaç örnek vereyim.

Örneğin basit bir blog scripti yazdınız. Burada blog yazılarınızı yayınlıyorsunuz. Scriptinize yorum özelliği gibi bir kaç tane de özellik eklediniz. Projeniz bitti ve arşive attınız. Daha sonra farklı bir proje için yine yazı ekleyip çıkarabileceğiniz daha ufak bir blog modülüne ihtiyacınız oldu. Bu durumda o proje için yeniden yazmak yerine daha önce yazmış olduğunuz yazılar modülü kullanabilirsiniz. Daha açık örneklemek gerekirse bugün yazmış olduğunuz bir modülü, eklentiyi, sınıfı ömür boyu çok daha kolay ve basit bir şekilde projenize dahil edebilirsiniz. Bu olay kod yazmadan proje çıkarmaya kadar gider.

PHP projelerinizde OOP mimarisini kullanmanızın bir diğer faydası da her şey yerli yerindedir. Bir de mvc mimarisini kullandığınızda 10 yıl sonra bile projeyi açtığınızda neyin nerede olduğunu çok rahatlıkla bulabilir çok daha profesyonel projeler oluşturabilirsiniz.

PHP ile OOP mimarisini kullanarak proje geliştirmek için bazı terimleri bilmemiz gerekiyor. Diğer dillerde olduğu gibi PHP dilinde de OOP mimarisi sınıf, fonksiyonlar(metot) ve parametreler üzerine kuruludur.

1.2. Sınıf nedir nasıl oluşturulur?

Sınıf (class) birçok fonksiyonu bir düzen içerisinde barındıran yapılardır. OOP mimarisinin temelidir diyebiliriz. Blog örneğimizden yola çıkarak bir yazı sınıfının içinde yazı ekleme, yazı silme, yazı düzenleme gibi fonksiyonlar bulunur. Sınıflar hakkında detaylı bilgiye http://php.net/manual/tr/language.oop5.php buradan ulaşabilirsiniz.

Sınıf tanımlamak için **class** komutunu kullanırız. Yukarıdaki örneğimizden devam ederek yazılar adında bir sınıf oluşturalım.

```
<?php

class yazilar {

   // php oop dersleri

}

?>
```

Sınıflar tıpkı fonksiyonlar gibi çağrılmayı beklerler. Bir sınıfı çağırmak için **New** komutu kullanılır. Bu komutumuzun baş harfinin küçük yada büyük olması herhangi bir farklılık yaratmaz. İki şekilde de kullanabiliriz.
Yukarıdaki yazılar sınıfımızı çağırmak için:

```
<?php

new yazilar();

?>
```

Yazmamız yeterlidir. Şu an yazılar sınıfımızı çağırdık ancak hiç bir değişiklik olmadı. Çünkü biz sınıfımıza ait bir metot tanımlamadık ve o metodu çağırmadık.

Sınıfımıza merhaba adında bir fonksiyon ekleyelim ve bu fonksiyon ekrana *"Merhaba"* yazsın. Bir sınıfa fonksiyon eklemek için **public function** komutunu kullanırız. (Şimdilik)

```php
<?php
class yazilar {
  public function merhaba(){
    echo "Merhaba";
  }
}
?>
```

Şimdi yazılar sınıfını bir değişkene atayıp çağıralım.

```php
<?php
$oop = new yazilar();
?>
```

Yazılar sınıfımız şu an çalışıyor. Bizim ekrana "Merhaba" yazmasını istediğimiz fonksiyonu çağırmadığımız için sınıfı çağırdığımız halde herhangi bir değişiklik olmadı. Bir sınıf altındaki fonksiyonu çağırmak için **->** işaretini kullanılırız. Yazılar sınıfımızdaki merhaba fonksiyonunu çağırmak için:

```php
<?php
$oop = new yazilar();
$oop->merhaba();
?>
```

Şeklinde kullanmamız gerekir.

Özetle ne yaptığımıza bir bakalım. **class** komutunu kullanarak yazilar adında bir sınıf oluşturduk. Daha sonra **New** komutuyla sınıfımızı çağırdık. Ardından **public function** komutu ile sınıfımıza merhaba adında bir fonksiyon tanımladık.

Son olarak da **->** işaretiyle çağırdığımız sınıfımızın merhaba fonksiyonu çalıştırdık. Kodlarla anlatmak gerekirse :

```php
<?php

class yazilar {

  public function merhaba(){

    echo "Merhaba";

  }

}

$oop = new yazilar();

$oop->merhaba();

?>
```

1.3. Yapıcı ve yıkıcı metot nedir?

Sınıfların ana metotları bir diğer adıyla yapıcı ve yıkıcı metotları bulunur. Bu ana metotlar *__construct* ve *__destruct* metotlarıdır. Bu arkadaşlardan __construct yapıcı, __destruct ise yıkıcı metot olarak bilinir. Yapıcı metot sınıf çağrıldığı anda çalışan metottur. Yani *New sinifadi();* diyerek bir sınıfı çağırdığınızda hiç bir metot belirtmeseniz bile çalışacak olan metottur.

Yıkıcı metot ise en son işlem bittiğinde çalışacak olan metottur. Yapıcı ve yıkıcı metot tanımlarken de **public function** komutunu kullanırız. Kısa bir örnekle konuya açıklık getirelim.

```php
<?php

class anametot {

  public function __construct(){
    echo "Sınıf çalıştı";
  }
```

```
  public function __destruct(){
    echo "Sınıf bitti";
  }

}

new anametot();

?>
```

Ekran çıktısı: *Sınıf çalıştıSınıf bitti*

Burada bir püf noktaya değinmek istiyorum. Eğer yapıcı metot parametre alıyorsa sınıfı başlatırken bu parametreleri girmek zorundayız. Örnek:

```
<?php

class anametot {

  public function __construct($arg){
    echo $arg;
  }

}

new anametot('parametre değeri');

?>
```

Eğer yapıcı metot parametre almıyorsa parantez kullanmadan da sınıfı çağırabiliriz. Yukarıda ki örneğimiz de ki yapıcı metodumuz parametre almamış olsaydı. **New anametot;** diyerek de sınıfı başlatabilirdik.

Yapıcı ve yıkıcı metotlar nerelerde kullanılır ?

Sınıfın çalışması için gerekli bilgiler varsa bu bilgiler yapıcı metoda gönderilir. Yada sınıf çalışmadan önce bazı işlemler yapılması gerekiyorsa kullanılır. Sınıf bitince silinmesi gereken değerler varsa bunlarda yıkıcı metotta halledilir. Mesela bir veri tabanı sınıfını başlatmak için sunucu adresi, kullanıcı adresi gibi bilgiler yapıcı metoda gönderilir.

Yapıcı ve yıkıcı metotları tanımlamak zorunlu değildir. Ancak ilerleyen konularda ve OOP mimarisini kavradığınızda çok fazla faydasını göreceksiniz.

2. Gizlilik

OOP mimarisinde gizlilik: sınıf içerisinde kullanılan sabit değişkenlerin ve metotların nerelerden erişilebileceğinin belirlenmesidir. Bir diğer deyişle sınıf içerisinde kullanılan tüm argümanların güvenliğinin belirlenmesidir.

Bir proje üzerinde birden çok yazılımcı çalıştığında, yazılımcıya erişebileceği ve erişemeyeceği sabit değişkenler ve metotlar tanımlayarak yanlış kullanımların önüne geçmiş oluruz.

PHP'nin bize sunmuş olduğu 5 adet gizlilik komutu bulunmaktadır. Bu komutlarımız *public, private, protected, static,* ve *final* komutlarıdır.

2.1. Public

Public komutuyla oluşturulan tüm sabit değişken ve metotlara her ortamdan erişilebilir. Örnek:

```php
<?php

class OopDersleri {

  public function __construct() {

  }

  public function merhaba() {

    echo "Merhaba";

  }

}

$oop = New OopDersleri();
$oop->merhaba();

?>
```

Bir önceki konumuza hatırlatma olması amacıyla ne yaptığımıza bir bakalım.

Class komutuyla yeni bir sınıf oluşturduk. **public function** komutuyla erişilebilir bir metot tanımladık. **$oop** değişkenine **New** komutuyla tanımladığımız sınıfı atadık. Ve son olarak da **$oop->merhaba()** koduyla da sınıfımız içinde ki merhaba adlı metodu çalıştırdık.

Bu arada sınıf içerisinden yada sınıf dışından erişilebilme de nedir diye soracak olursanız hemen ona da bir açıklık getirelim.

Yukarıdaki örneğimiz sınıf dışından erişime örnektir. Şöyle ki biz önce sınıfımızı yazdık daha *$oop* adında bir değişken oluşturup *New* komutuyla sınıfı başlattık. Daha sonra da *$oop->merhaba();* koduyla da sınıfın merhaba metoduna eriştik. Eğer ki merhaba metoduna sınıf içinden erişmeye çalışmış olsaydık kodlarımız şu şekilde olmalıydı.

```
<?php

class OopDersleri {

  public function __construct(){

     $this->merhaba();

  }

  public function merhaba() {

     echo "Merhaba";

  }

}

$oop = New OopDersleri();

?>
```

Burada dikkatinizi çektiği üzere **$this** adında bir komut kullandık. Sınıf içerisindeki public olarak tanımlanan bir metot veya sabit değişkene **$this** komutuyla erişebiliriz. Şimdide public bir argüman tanımlayalım.

Kodlarımız:

```php
<?php

class OopDersleri {

  public $merhaba = "Merhaba";

  public function __construct(){

    echo $this->merhaba;

  }

}

$oop = New OopDersleri();

?>
```

Tekrar ne yaptığımıza bir bakalım.

OopDersleri adında bir sınıf oluşturduk. *public $merhaba = "Merhaba";* ile public bir argüman oluşturduk ve __construct ana metodumuzda oluşturduğumuz argümanı ekrana bastık. Burada dikkat etmemiz gereken olay **$this->** komutuyla sınıf içerisindeki bir argümana erişmek istersek argüman adından sonra () kullanmıyoruz. Ancak bir metoda erişmek istersek () parantez kullanmalıyız.

2.2 Private

Yalnızca sınıf içerisinden erişilebilen metot veya argüman oluşturmak istediğimiz zaman **private** komutunu kullanırız. Private olarak tanımladığımız bir metot veya argümana sınıf dışından yada türetilen bir sınıf dışından erişmemiz mümkün değildir. Türetilen sınıf da nerden çıktı diyecek olursanız birazdan açıklayacağım. Örneğimizdeki merhaba metodunu private olarak tanımlayalım.

```php
<?php

class OopDersleri {

  public function __construct(){

  }

  private function merhaba(){
```

```
    echo "Merhaba";

  }

}

$oop = New OopDersleri();
$oop->merhaba();

?>
```

Bu kodumuzu çalıştırdığımızda *fatal error (ölümcül hata)* döndürecektir. Çünkü biz **private** olarak tanımladığımız bir metoda sınıf dışından *($oop->merhaba())* erişmeye çalıştık.

Private olarak tanımlanan bir metot veya argümana sınıf içerisinden erişmek istediğimiz zaman yukarıda yeni öğrenmiş olduğumuz **$this** komutuyla yada self::metotadı şeklinde erişebiliriz. Yukarıdaki kodumuzun çalışan halini yazarak private olayımızı da halledelim.

```
<?php

class OopDersleri {

  public function __construct(){
    self::merhaba();
  }

  private function merhaba(){

    echo "Merhaba";

  }

}

$oop = New OopDersleri();

?>
```

2.3. Protected

Protected komutuyla oluşturulan metot ve argümanlara yalnızca sınıf içerisinden ve türetilen sınıf içerisinden erişilebilir. Protected olarak tanımlanan bir metot veya argümana sınıf dışından asla erişilemez. Şimdi az önceki sözümüzü tutup türetilen sınıf nedir onu bir açıklayalım.

Bir sınıfı başka bir sınıftan türetmek için **extends** komutunu kullanırız. Örneğin a sınıfını b sınıfından türetmek için yazmamız gereken kod şudur:

```php
<?php

class b {

  public function merhaba(){

    echo "Merhaba";

  }

}

class a extends b {

  public function __construct(){
    parent::merhaba();
  }

}

$oop = New a();

?>
```

Peki, bir sınıfı bir başka sınıftan türetirsek ne olur ?

Eğer bir sınıfı bir başka sınıftan türetirsek türetilen sınıf içerisinde ki (private komutuyla oluşturulanlar hariç) tüm metot ve argümanları türettiğimiz sınıf içerisinden de kullanabiliriz. a sınıfımızın __construct metoduna yazdığımız **parent::** komutuyla türetilen sınıf içerisinde ki metodumuzu a sınıfı içerisinde de kullanabiliyoruz. Konuya açıklık getirmek gerekirse yukarıdaki kodumuzu *$oop = New a()* diyerek a adlı sınıfımızı başlattık. Sınıfımız ekrana *"Merhaba"* çıktısını verdi. Ancak biz a sınıfı içerisinde ekrana "Merhaba" yazan bir metot tanımlamadık.

Peki nasıl oldu ?

Aslında ekrana *"merhaba"* çıktısını veren metodumuz b sınıfımız içerisinde ki merhaba metodu. Biz a sınıfımızı b sınıfından türetmiştik. a sınıfımızın __construct metoduna da **parent::merhaba()** yazdığımız için doğal olarak a sınıfımızı başlattığımız an b sınıfı içerisinde ki merhaba metodu da çalışıyor. Sınıf türetme olayına genişletme de denilmektedir.

Protected komutuyla oluşturulan metot ve argümanlara yalnızca sınıf içerisinden ve türetilen sınıf içerisinden erişilebilir dedik ve bir örnek vererek konuyu netleştirelim.

```
<?php

class b {

  protected function merhaba(){

     echo "Merhaba";

  }

}

$oop = New b();
$oop->merhaba();

?>
```

Yukarıdaki kodumuzu çalıştırdığımız da **fatal error** verecektir. Çünkü protected olarak tanımladığımız bir metoda sınıf dışarısından erişmeye çalıştık. Şimdi bu kodumuzun çalışan halini yazalım.

```
<?php

class b {

  protected function merhaba(){

     echo "Merhaba";

  }

}

class a extends b {

  public function __construct(){
```

```
    parent::merhaba();
  }

}

$oop = New a();

?>
```

Bu şekilde ekrana *"Merhaba"* çıktısını verecektir.

2.4. Static

Static komutuyla oluşturulan metot ve argümanlara sınıfı başlatmadan sınıf içerisinden yada sınıf dışından erişebiliriz. Static özelliği genellikle en çok kullanılan metotların daha hızlı çalışması ve bir sınıfın sadece bir metodunu, sınıfı çağırmadan kullanacaksak **static** komutunu kullanırız. Static komutunun diğerlerinden farkı public static şeklinde bir söz dizimi olmasıdır. Örneğin *static $degisken* veya *static fonksiyon()* şeklinde yazarsak hata ile karşılaşırız.

Static olarak tanımlanan bir metodu yada argümanı sınıf içerisinden kullanmak istediğimiz zaman ise **self::$degisken** yada **parent::$degisken** şeklinde kullanabiliriz.

Burada dikkat etmemiz gereken static olarak tanımlanmış olan değerlerin başına **$** işaretini koymamız gerektiği.

Static olarak tanımlanan tüm metotlar ve argümanlar RAM'e atılır. Çağrıldığı zaman ise direk RAM'den çağrılarak diğer gizlilik komutlarına göre çok daha hızlı çalışır. Böylelikle performans kaybını önlemiş oluruz. Tabi ki bu her metodu static olarak tanımlamamızın doğru olduğu anlamına gelmez. Basit bir örnek vererek konumuzu netleştirelim.

```
<?php

class b {

  public static $sabit = "Oop Dersleri";

  public function yaz(){

    echo self::$sabit;
```

```
  }

}

$oop = New b();
$oop->yaz();

?>
```

Ekran çıktımız: *"Oop Dersleri"* olacaktır. Performanstan bahsetmişken yukarıda ki örneğimizi biraz daha farklılaştıralım. Kodlarımız:

```
<?php

class b {

  public static $sabit = "Oop Dersleri";

  public static function yaz(){

     echo self::$sabit;

  }

}

b::yaz();

?>
```

Burada ne yaptığımıza bir bakalım. *$sabit* adında bir static değişken oluşturduk, yaz adında bir metot oluşturduk ve bu metodumuz içinden *$sabit* değerini ekrana yazdırdık. Ancak burada New komutu kullanmayıp sınıfı başlatmadığımız halde yine ekrana *"Oop Dersleri"* değerini yazdırdı.

PHP dilinde bir sınıfı çağırmadan da sınıf içerisinde static olarak tanımlanmış metot ve argümanlara erişebiliriz. Bunu yapabilmek için **sinifadi::arguman** veya **sinifadi::metot()** şeklinde kullanmamız gerekir. Bu şekilde bir kullanım bize sınıf içerisinde 100 tane metot olsa bile sadece bizim istediğimiz metodu yükler ve çalıştırır. __construct ve __destruct ana metotları dahi çalışmaz. Yani sınıfı komple yüklemez. Bu şekilde de performans artışı sağlayabiliriz.

Ayrıca içinde static bir değer bulunan bir sınıfı New diyerek başlattığımızda RAM'e her defasında yeni bir obje eklenir. Bu da hafıza da gereksiz bir yer kaplar. Sonuç olarak ne kadar New komutuyla tanımlarsak hafıza da o kadar fazla yer kaplamış oluruz.

2.5. Final

Final komutu türetilen sınıflarda en son kullanılacak sınıf, metot veya argümanlarda koruma amaçlı kullanılan bir komuttur. Özellikle bileşen (component) yapımında birden çok developer'ın kullanıldığı durumlarda olmazsa olmaz güvenlik mekanizmalarının başında gelir.

Final komutu sınıf tanımlamalarının başında kullanılır. Şöyle bir örnek verelim.

```
<?php

final class Bitis {

}

$oop = new Bitis();

?>
```

Yukarıdaki gibi final komutuyla tanımlanan bir sınıf başka bir sınıf tarafından türetilemez. Ancak bir başka sınıftan türeyebilir. Şu şekilde bir kullanım yanlıştır.

```
<?php

final class a {

}

class b extends a {

}

$oop = new b();

?>
```

Burada a sınıfımızı final olarak tanımladığımız için b sınıfımızı da a sınıfından türettiğimiz için hata alırız. Bu kodun doğru olan şekli ise şöyle olmalıdır.

```php
<?php

class a {

}

final class b extends a {

}

$oop = new b();

?>
```

Oop mimarisinde gizlilik kavramları bundan ibarettir arkadaşlar. Bu konuda neler yaptığımızı, öğrendiğimiz yeni komutların ne işe yaradığının kısa bir özet geçelim.

public : Her yerden erişilebilir.

private : Yalnızca sınıf içerisinden erişilebilir.

protected : Sınıf içerisinden ve türetilen sınıflardan erişilebilir.

static : Bir sınıfın yalnızca bir metodunu kullanacaksak static tanımlamalıyız.

final : Başka bir sınıf tarafından türetilemez. Koruma amaçlı kullanılır. Sınıfların başında tanımlanır.

$this : Sınıf içerisinde ki herhangi bir metoda erişmek için kullanılır.

parent:: : Türetilen sınıf içerisindeki metotlara erişmek için kullanılır.

self:: : $this komutunun php5'deki halidir. $this yerine self kullanılması önerilir.

sinifadi::metod() : Bir sınıfı komple çağırmadan sadece istenen sabit metodun çağrılmasını sağlar.

sinifadi::arguman : Bir sınıfı komple çağırmadan sadece istenen sabit argümanın çağrılmasını sağlar.

self::$sabit : static olarak tanımlanan bir objenin çağrılmasını sağlar. Unutmamamız gereken static olarak tanımlanan değeri çağırırken önüne **$** işareti koymamız gerektiği.

extends : Bir sınıfı başka bir sınıftan türetmek istediğimiz zaman kullanılır.

3. Sihirbaz Metotlar

Sihirbaz metot: sınıf ve olay çevresinde çalışan PHP dili ile tanımlı olarak gelen metotlardır.

Sihirbaz metotların sadece sınıf içerisinde kullanıldığı gibi yanlış bir bilgi oluşmasın kesinlikle. Sınıf dışında da bazı sihirbaz metotları kullanabilmek mümkündür.

Bazı sihirbaz metotların diğer metotlara nazaran geliştiriciler tarafından çok fazla kullanıldığını söylemek doğru değil ancak bazı gerekli durumlarda kesinlikle kullanılması kanaatindeyim. PHP 5 ile gelen toplam 15 adet sihirbaz metot bulunmaktadır. PHP 7 de de bir değişiklik şu an söz konusu değil.

Sihirbaz metotların önünde __ işlevi bulunur ve sihirbaz olmayan metotların önüne __ işlevi koyulması PHP tarafından önerilmez. İleride sorunlar çıkarabilir. Biz bu yazımızda 4 adet sihirbaz metodu inceleyeceğiz. Sihirbaz metotların hepsi *public* olarak tanımlanmak zorundadır.

3.1. __construct()

__construct() metodu bir sınıf çağrıldığı anda çalışan metodumuzdur. Bu metodumuzu giriş bölümünde yapıcı metot olarak adlandırmıştık. Basit bir örnek ile olayı netleştirelim.

```php
<?php

class OopDersleri {

  public function __construct(){

    echo "OopDersleri sınıfı çalıştı.";

  }

}

$oop = New OopDersleri();

?>
```

Bu betiği çalıştırdığımızda *"OopDersleri sınıfı çalıştı"* çıktısını verecektir. __construct() sihirbaz metodu nerelerde kullanılır? ne işimize yarayacak? diye soracak olursanız bu metodumuz genelde türediği sınıfın metotlarına erişmek, sınıf kullanılmadan önce yapılması gereken temizleme vb. gibi işlemler için kullanılır.

3.2. __destruct()

__destruct() metodu ise __construct() metodunun aksine sınıf en son işlemi yaptığında yani sınıf bittiğinde çalışan metottur. Bu metodumuza da giriş bölümünde yıkıcı metot demiştik. Şimdi bir önceki sihirbaz metodumuz ve __destruct() metodumuzu kullanarak bir örnek yapalım.

```php
<?php
class OopDersleri {
  public function __construct(){
     echo "OopDersleri sınıfı çalıştı.";
  }
  public function __destruct(){
     echo "OopDersleri sınıfı bitti.";
  }
}
$oop = New OopDersleri();
?>
```

Bu betiği çalıştırdığımızda ise *"OopDersleri sınıfı çalıştı.OopDersleri sınıfı bitti."* çıktısını verecektir.

3.3. __toString()

Bu sihirbaz metodumuz sınıf objesini direk ekrana bastığımızda, obje değeri yerine istediğimiz bir içeriğin veya argümanın çıkmasını sağlar. Bir objeyi direk ekrana basmak da nereden çıktı diyecek olursanız eğer hemen kısa bir örnekle önce onu açıklayalım.

```php
<?php

class OopDersleri {

}

$oop = New OopDersleri();
echo $oop;

?>
```

Burada echo *$oop* koduyla objeyi direk ekrana bastık. Ancak bu kodumuz bize bir hata döndürdü. Şimdi basit bir örnekle **_toString** sihirbaz metodumuzun ne işe yaradığına bir bakalım.

```php
<?php

class OopDersleri {

  public $name = "OopDersleri Sınıfı";

  public function __toString(){
    return $this->name;
  }

}

$oop = New OopDersleri();
echo $oop;

?>
```

Bu betiğimiz bize *"OopDersleri Sınıfı"* çıktısını verecektir. Bu metodumuzu da aslında sınıfın adını ve sınıfın özelliklerini ekrana basmak için kullanıyoruz.

3.4. __call()

Bu sihirbaz metodumuz ise sınıf içerisinde olmayan bir metodu veya argümanı çağırdığımız zaman çalışan metodumuzdur. Genellikle hata metni oluşturmak için kullanılır. __call sihirbaz metodu diğerlerinden farklı olarak *iki adet parametre* alır.

```php
<?php

class OopDersleri {

  public function __call($name, $attr){
    echo $name. " adlı metot bulunamadı.";
  }

}

$oop = New OopDersleri();
$oop->yaz();

?>
```

Bu betiğimiz bize çıktı olarak *"yaz adlı metot bulunamadı."* çıktısını verecektir. Eğer burada __call sihirbaz metodunu kullanmamış olsaydık sınıf içerisinde olmayan bir metodu çağırdığımız için PHP derleyicimiz hata verecekti.

Konunun başında bahsettiğim 15 adet sihirbaz metodu incelemek için http://php.net/manual/tr/language.oop5.magic.php adresini kullanabilirsiniz.

3.5. Örnek Sınıf

Bu konumuzda ise şimdiye kadar olan tüm konuların genel bir özetini kapsayacak örnek bir sınıf oluşturacağız ve nesne tabanlı programlama yapısını daha iyi anlayacağız.

Örnek sınıfımız oturum işlemlerini temel düzeyde yapacağımız bir session sınıfı olacak. Bu sınıfımızda oturum başlatma sonlandırma gibi işlemlerimizi yapacağız.

Ne yapacağımız ile ilgili kafanızda bir şeyler canlandı sanırım. Dilerseniz hemen başlayalım. İlk olarak **class** komutuyla yeni bir sınıf oluşturalım. Ancak bu sınıfımızı **final** komutuyla tanımlayalım.

Gizlilik konumuzdan hatırlayacağınız üzere final komutunu koruma amaçlı ve başka bir sınıf tarafından türetilememesi için kullanıyorduk. Şimdi sınıfımızı oluşturalım.

```php
<?php
  final class Session {

  }
?>
```

Sınıfımızı oluşturduktan sonra oturumları başlatacak olan metodumuzu yazalım. Bu metodumuz çok fazla bir şey yapmayacak şimdilik bizim için *session_start()* demesi yeterli. Bu metodumuza static olarak tanımlayalım. Gizlilik konusunda **static** komutunu sadece argümanlarda kullanmıştık şimdi de metotlarda kullanalım zaten yazım olarak pek bir farkı yok.

```php
<?php

  final class Session {

    public static function init(){
      session_start();
    }

  }
?>
```

Bu metodumuzu static olarak tanımlamamızın bir diğer sebebi de sınıf dışından erişimi sağlamak. Şu an sınıfımız oturumu başlatan bir metoda sahip.

Şimdi de oturumu sonlandıracak olan metodumuzu yazalım. Bu metodumuzu da *static* olarak tanımlıyoruz.

```php
<?php

  final class Session {

    public static function init(){
      session_start();
    }
    public static function logout(){
      session_destroy();
    }

  }
```

```
?>
```

Sihirbaz metotlar konumuzdan da bir şeyler yapalım ki bir faydası olsun. Şimdide sınıfımıza sınıfın adını ve özellikleri yazdıran **__toString** sihirbaz metodumuzu ekleyelim. Bunun için sınıfın adını taşıyan *name* adında bir private değişken, sınıfın özelliklerini taşıyan *properties* adında bir public değişken oluşturalım.

```php
<?php

  final class Session {

    private $name      = "Oop Dersleri Örnek Session Sınıfı";
    public $properties = "Oturum başlatma,kapatma vs.";

    public static function init(){
      session_start();
    }
    public static function logout(){
      session_destroy();
    }

    public function __toString(){
      echo $this->name." - ".$this->properties;
    }

  }
?>
```

Sınıf içerisinde ki bir metot veya değişkene ulaşmak için **$this** komutunu kullanıyorduk. Burada unutmamamız gereken şey ise bir metoda erişirken metot isminin yanına ( ) işaretlerini koymamız gerekiyordu.

Artık sınıfımız oturum başlatıp sonlandırabiliyor. Şimdi ise oturumumuza değer ekleyecek olan set metodumuzu ve bize oturumdan değer döndürecek olan get metodumuzu yazalım.

Oop mimarisinde set ve get metotları çok kullanılır. Yukarıda da söylediğim gibi bir sınıf içerisinde ki bir argümana veya metoda değer eklemek set ve sınıf içerisinden her hangi bir değer almak için ise get kullanılır. Daha doğrusu global olarak bu işlemleri yapan metotlara get ve set adı verilir demek daha doğru olur. Bazı kaynaklarda **getter** ve **setter** metotları olarak da geçmektedir.

Şimdi ilk olarak set metodumuzu yazalım.

```php
<?php

  final class Session {

    private $name      = "Oop Dersleri Örnek Session Sınıfı";
    public $properties = "Oturum başlatma,kapatma vs.";

    public static function init(){
      session_start();
    }
    public static function logout(){
      session_destroy();
    }

    public function __toString(){
      echo $this->name." - ".$this->properties;
    }

    // Set metodumuz
    public static function set($key, $value){
      $_SESSION[$key] = $value;
    }

  }
?>
```

Burada farklı olarak set metodumuz iki adet parametre almaktadır. Burada ki parametreler fonksiyonlarda ki gibi çalışmaktadır. Mantık olarak aynıdır. *$key* parametremiz dizinin anahtarını *$value* ise değerini belirtir.

Şimdi de get metodumuzu yazalım. Get metodumuz sadece bir parametre alacak ve bu parametre ile dizinin hangi elemanının verisini almak istediğimizi metodumuza söylemiş olacağız.

```php
<?php

  final class Session {

    private $name      = "Oop Dersleri Örnek Session Sınıfı";
    public $properties = "Oturum başlatma,kapatma vs.";

    public static function init(){
      session_start();
    }
    public static function logout(){
```

```
        session_destroy();
    }

    public function __toString(){
        echo $this->name." - ".$this->properties;
    }

    // Set metodumuz
    public static function set($key, $value){
        $_SESSION[$key] = $value;
    }

    // Get metodumuz
    public static function get($key){
        if(isset($_SESSION[$key])){
            return $_SESSION[$key];
        }else{
            return false;
        }
    }
  }
?>
```

Artık sınıfımız kullanıma hazır. Yukarıdaki kodlarımızı *class.session.php* olarak ana dizinimize kaydediyoruz. Şimdi index.php yada siz nasıl dilerseniz başka bir dosya oluşturalım ve içine yazmış olduğumuz sınıfımızı include edelim.

İlk olarak sınıfımızı başlatmadan önceki konularımızda bahsettiğimiz gibi sınıf dışından init metoduyla oturumu başlatıp set metodumuzla içine bir şeyler ekleyelim. Daha sonra eklediğimiz değeri ekrana basalım ve logout metodumuzla oturumu kapatalım.

```
<?php

  include('class.session.php');

  Session::init();

  Session::set('login', true);

  echo Session::get('login');

  Session::logout();

?>
```

Bu betiği çalıştırdığımızda ekrana *"1"* çıktısını verecektir. Sınıf dışından bir metoda erişmek istediğimiz zaman *Sinifadi::metot()* şeklinde erişiyorduk. Sınıfımızın yaptığı işlemi normal şartlarda bu şekilde yapıyoruz.

```php
<?php
  session_start();
  $_SESSION['login'] = true;
  echo $_SESSION['login'];
  session_destroy();
?>
```

Şeklinde de yine *"1"* çıktısını verecektir. Şimdi de sınıfı başlatıp aynı işlemi yapalım.

```php
<?php
  include('class.session.php');
  $session = New Session();
  $session->init();
  $session->set('login', true);
  echo $session->get('login');
  $session->logout();
?>
```

Ekran çıktısı: 1

4. Autoload

Yukarıda ki örneğimizde bir sınıf oluşturup bu sınıfımızı sayfaya include etmiştik. Peki 50-60 tane sınıfımız olmuş olsaydı her sınıf için tek tek include mi edecektik? Tabi ki de hayır. Zaten bunu elle yapmış olsaydık vay halimize. Peki, bunun için ne yapabiliriz. PHP'nin bize sunmuş olduğu __autoload fonksiyonunu kullanarak bu soruna kökten çözüm getirebiliriz.

Kod standartlarında genel olarak sınıflar ayrı bir klasörde tutulur. Örneğin classes, class, lib ve library gibi. Ben genelde library adında bir klasör oluşturup onun içinde tutuyorum.

Lafı fazla uzatmadan konumuza dönelim. Library adında bir klasörünüz var ve siz sınıflarınızı bu klasörde tutuyorsunuz diyelim. 40-50 tane sınıf yazdınız ve tek tek yüklemek istemiyorsunuz. O halde sizi şöyle alayım.

```php
<?php
  function __autoload($class){
    $file = 'library/'.$class.'.php';
    if(file_exists($file)){
      require_once($file);
    }else{
      echo "sınıf bulunamadı";
    }
  }
?>
```

Şeklinde kullanabilirsiniz. Burada klasör adresinize dikkat etmelisiniz.

Peki 50-60 tane dosyayı her seferinde yüklemek sistemi yavaşlatmaz mı?

Tabi ki yavaşlatır. __autoload fonksiyonu yeni bir sınıf başlatıldığı zaman çalışır.

Şöyle ki *New Session();* diyerek yeni bir sınıf başlattığınızda yalnızca library klasörünüzde bulunan session.php dosyası yüklenmiş olur. Yani her seferinde tüm dosyaları yüklemez. Böylelikle hem performans sağlamış oluruz hem de yeni bir sınıf yazdığımızda elle eklemekten kurtulmuş oluruz.

Burada dikkat etmemiz gereken bir diğer konuda sınıf adıyla dosya adının aynı olması. Mesela oop.php adında bir dosya oluşturduk içine de *class OopDersleri* diyerek OopDersleri adında bir sınıf oluşturduk diyelim. *New OopDersleri()*; diye sınıfımızı başlattığımızda **__autoload** fonksiyonu çalışacak ve library klasörü altında OopDersleri.php dosyasını arayacaktır. Öyle bir dosya olmadığı için de doğal olarak else kısmında yazdığımız hatayı verecektir. Windows (:p) işletim sisteminde herhangi bir sorun yaşamazsınız ancak kodlarınızı sunucuya attığınızda hata alırsınız. Bu kısmı aklımızdan çıkarmıyoruz.

Peki bir sınıfı başlattık ancak sadece belirli yerlerde kullanıyoruz. Mesela kategoriler sınıfımızı sistemimiz başlarken otomatik olarak başlattık. Yazılar kısmında işlem yapıyoruz diyelim. Yazılar kısmında çalışırken kategoriler sınıfına ihtiyacımız yok.

PHP'de kullanılmayan sınıfların otomatik olarak yüklenmesinin önüne geçmek için *spl_autoload_register* fonksiyonu kullanılır. Frameworklerde ve OOP uygulamalarında genel olarak bu fonksiyonumuz autoloader'ların altına eklenir. Biz de standartlara uyarak kodumuzu şu şekilde değiştiriyoruz.

```php
<?php
  function __autoload($class){
    $file = 'library/'.$class.'.php';
    if(file_exists($file)){
      require_once($file);
    }else{
      echo "sınıf bulunamadı";
    }
  }
  spl_autoload_register('__autoload');
?>
```

5. Overloading

Overloading: aynı isimde birden çok metot veya argüman tanımlama özelliğidir. Sınıf içerisinde olmayan bir değişkene değer atamak da diyebiliriz. PHP'nin overloading konusu diğer nesne tabanlı programlama dillerinden çok daha farklı bir yapıya sahiptir.

"Sınıf içerisinde olmayan bir değişkene değer atamak" derken neyi kastettiğimi basit bir örnekle anlatayım.

```php
<?php

class overloading {

}

$overloading = new overloading();
$overloading->deneme = "php";

echo $overloading->deneme;

?>
```

Bu kodu çalıştırdığınızda ekrana *"php"* çıktısını verecektir. Burada sizin de dikkatinizi çektiği gibi sınıf içerisinde deneme adında bir değişken tanımlamadığımız halde bu değişkene değer atayabiliyoruz. Overloading'i tanım olarak böyle de anlatabiliriz. Ancak bu şekilde bir kullanım doğru değildir.

Hataları minimize etmek ve ileride oluşabilecek sorunların önüne geçmek için overloading konusunda PHP'nin bize sunmuş olduğu bazı metotlar bulunmaktadır. Bu metotlar*: __get __set __isset __unset __call* ve *__callStatic* metotlarıdır. Bu metotların bazıları parametre aldığı gibi bazıları da void olarak yani parametresiz olarak çalışabilir. Burada ki bazı metotlar, size sihirbaz metotlar konumuzdan tanıdık gelebilir. Hatta bu metotlarda sihirbaz metotlardır.

Overloading konusunda kullanacağımız metotları önce kısa kısa açıklayacak olursak eğer:

__get: Sınıf içerisinde olmayan bir değişkenden veri almaya çalıştığımızda tetiklenen metot.

__set: Sınıf içerisinde olmayan bir değişkene veri atamaya çalıştığımızda tetiklenen metot.

__isset: Sınıf içerisinde olmayan bir değişken veya metot için isset ve empty kontrolleri yapıldığında tetiklenen metot.

__unset: Sınıf içerisinde olmayan bir değişken için unset komutu çağrıldığında tetiklenir.

__call: Sihirbaz metotlar konumuzdan hatırlayacağınız üzere sınıf içerisinde olmayan bir metot çağrıldığında tetiklenen metot.

__callStatic: Bir üstte ki metodumuzla aynı işi yapar ancak static olarak çalışmaktadır.

Bu metotlarımızı kullanabilmek için hepsinin public olarak tanımlanması gerekir. __callStatic hariç tabi ki. Açıkçası overloading konusu sürekli kullanabileceğimiz bir özellik değil. Ancak çok karışık parametrelerin gelmesi gereken sınıflarda olmazsa olmaz bir özellik ve kesinlikle kullanılması gerekir. Şimdi basit bir örnekle overloading kullanımını daha iyi anlayalım.

```php
<?php

class Overloading {

  // Overloading yapılacak değişken
  private $data = [];

  // Overloading set edecek olan metodumuz
  public function __set($key, $val){
    $this->data[$key] = $val;
  }

  // Overloading get edecek metodumuz
  public function __get($key){

    if(array_key_exists($key, $this->data)){
      return $this->data[$key];
    }

  }

  // Overloading isset kontrolu yapacak metodumuz
  public function __isset($key){
    return isset($this->data[$key]);
  }
```

```php
  // Overloading unset edecek metodumuz
  public function __unset($key){
    unset($this->data[$key]);
  }


  // Overloading call metodumuz
  public function __call($key, $detail){

    echo $key." adında metodun değeri:\n";
    print_r($detail);

  }

  // Overloading callStatic metodumuz
  public static function __callStatic($key, $detail){

    echo $key." adında metodun değeri:\n";
    print_r($detail);

  }

}

$overloading = New Overloading();
$overloading->dil = "php";
$overloading->konu = "overloading";

echo $overloading->dil;
echo "<br/>";
echo $overloading->konu;
echo "<br/>";

$overloading->olmayanmetot();
echo "<br/>";
unset($overloading->dil);
$overloading->deneme = array(1,2,3,4,5);
print_r($overloading->deneme);

?>
```

Şimdi ne yaptığımıza tekrar bir bakalım. Sınıf içerisinde olmayan dil ve konu değişkenlerine değer atarken __set metodu tetikleniyor. *echo $overloading->dil;* dediğimiz zaman da *__get* metodu tetikleniyor ve çağırdığımız değişkeni geri döndürüyor. *$overloading->olmayanmetot()* dediğimizde sınıf içerisinde ki __call metodu tetikleniyor. Son olarak da *unset($overloading->dil)* dediğimiz de ise *__unset* metodu tetikleniyor.

6. Soyut Sınıflar

Soyut sınıf; içinde en az bir soyut metot bulunan ve türetilen sınıflarda bu metodun bulunması zorunlu olan sınıflardır. Burada ki soyut kavramı bu metotların somut işlemleri içermemeleri bu somut işlemleri türetilen sınıflara bırakmasıdır. Daha kesin bir tanım yapılmak gerekirse soyut sınıflar sadece tanımlanıp bırakılır içerisinde herhangi bir işlem yapılmaz.

OOP mimarisinde soyut sınıflar sistematik bir sistem kurmak ve hataları minimize etmek amacıyla kullanılır. Component(bileşen) mantığında olmazsa olmaz yapılardan biridir. Soyut sınıflar normal sınıf tanımlamalarından farklı olarak tanımlanır ve New komutuyla çağırılamazlar. Örnek vermek gerekirse:

```php
<?php

abstract class Soyut {

}

?>
```

Soyut sınıflar bu şekilde tanımlanmaktadır. Bu noktaya lütfen dikkat edin. Soyut sınıfları türeyen sınıflar üzerine çekilen bir katman olarak düşünebilirsiniz. Başka bir deyişle türeyen sınıflarda soyut metotlar tanımlanmak zorundadır. Kısa bir örnek verelim.

Diyelim ki bir mvc yapınız var. Controller model ve view adında çekirdek işlemleri yapan sınıflarınız var. Ve tüm controller dosyalarını bu çekirdekte yer alan controller sınıfından, tüm model dosyalarını da yine çekirdekte ki model sınıfından türetiyorsunuz. Her şey harika olması gereken de budur zaten. Sistem üzerinde mvc çalışma mantığı temel olarak controller/metot/parametre olarak çalışıyor diyelim. Ve controller dosyası çağrıldığı zaman bir metot belirtilmezse genel de index yada home adında bir metot oluşturup bu metodu varsayılan olarak çağırırız.

İşte tam bu noktada ana controller dosyası içinde varsayılan olarak çağırılan metodun ismini **abstract** olarak tanımlayarak türetilen sınıflarda bu metodun olmasını zorunlu hale getirebiliriz. Böylece daha sistematik bir yapı oluşturup, olası hataların önüne geçebilir ve doğru olanı yapmış oluruz. Kafanızda bir şeyler canlandıysa konumuza tekrar dönüş yapalım.

Soyut sınıflar içinde soyut bir metot tanımlamak için metodun önüne **abstract** komutunu koymamız yeterlidir. Burada ince bir noktaya da değinelim. Soyut metotların *public* yada *protected* olarak tanımlanması gerekir. Gizlilik konusunda yer alan private, static gibi metot tipleriyle soyut metotlar tanımlanamaz. Bir diğer husus ise soyut sınıflar sadece tanımlanıp bırakılır. Örnek:

```php
<?php
abstract class Controller {
  // Yanlış soyut metod
  abstract public function Metod(){ }
  // Doğru soyut metod
  abstract public function Metod();
}
?>
```

Aslına bakarsanız soyut sınıf denilmesinin sebebi de tam olarak budur. Şimdi yukarıda ki kullanım örneğinde bahsettiğimiz mvc mimarisinde yer alan soyut sınıf yapısının bir örneğini yaparak konuyu daha da netleştirelim.

```php
<?php
abstract class Controller {
  abstract public function index();
}
class MainController extends Controller {
  public function index(){
    // İşlemler
  }
}
new MainController();
?>
```

Eğer ki MainController sınıfı içerisinde index adında bir metot tanımlamasaydık PHP derleyicimiz bize *"Abstract function Controller::index() cannot contain body..."* hatası verecekti.

Peki, şöyle bir sınıfımız olduğunu düşünelim.

```
<?php

abstract class Controller(){

  abstract public function index();

  public function Load(){

    return true;

  }

}

class MainController extends Controller {

  public function index(){ }

}

new MainController();

?>
```

Controller soyut sınıfımız içerinde iki adet metot bulunuyor. Bu metotlardan index metodunu türettiğimiz MainController sınıfı içerisinde tanımladık.

Diğer metodumuzu ise tanımlamadık sizce çalışır mı?

Evet çalışır. Hani soyut sınıflar içerisinde ki metotlar türetilen sınıflarda kullanılmak zorundaydı? Burada bir ayrıntıya daha yer verelim.Soyut sınıflar içerisinde yalnızca soyut olarak tanımlanan metotların türetilen sınıflarda tanımlanması zorunludur. Yukarıda index metodunu abstract olarak tanımladığımız için MainController sınıfı içerisinde bu metodu tanımlamak zorundayız. Eğer Load metodunu abstract olarak tanımlamış olsaydık onu da kullanmak zorundaydık.

7. Nesne Arayüzleri

7.1. Giriş

Nesne arayüzleri: Bir sınıf içerisinde zorunlu bir şekilde kullanılmasını istediğimiz metot ve argümanları barındıran sınıflara denir.

Nesne arayüzleri diğer sınıf tanımlamalarından farklı olarak **interface** anahtar sözcüğü ile tanımlanır. Örneğin:

```
<?php

interface arayuz {

}

?>
```

Aslına bakarsanız nesne arayüzleri ve soyut sınıflar birbirine çok benzer hatta çok karıştırılır. Nesne arayüzlerini soyut sınıflardan farklı kılan şey nesne arayüzleri içerisinde yalnızca tanımlama metotları kullanılır ve tüm metotlar public olarak tanımlanmak zorundadır. Interface'ler New komutuyla çağırılamazlar.

Yalnızca genişletilerek kullanılabilir. Burada bir diğer fark ise nesne arayüzleri extends anahtar kelimesiyle değil **implements** anahtar kelimesiyle kullanılır. Yine burada bir fark daha var ki oda şu eğer bir interface başka bir interface'den türeyecekse extends komutu normal bir sınıf bir interface'den türeyecekse implements komutu kullanılır. Kafanızın çok karıştığının farkındayım ve hemen bir örnek ile bu karışıklığı giderelim.

```
<?php

  interface Arayuz {

  }


  interface BaskaBirArayuz extends Arayuz {

  }

  class Arayuzler implements BaskaBirArayuz {

  }
```

```
?>
```

Interface'ler New komutuyla çağırılamazlar dedik ve ona da çok kısa bir örnek verelim.

```
<?php
  interface Arayuz{
  }
  new Arayuz();
?>
```

Bu şekilde bir kullanım hata oluşturur.

7.2. Nesne arayüzleri neden kullanılır ?

Interface'lerin Türkçe manasıyla arayüz anlamına gelmesi interface'lerin genellikle tema motorlarında yada tema yapımında kullanıldığı gibi bir algı oluşturmuştur ancak olay burada biraz daha farklı. Interface'ler tabii ki tema yapımında da kullanılabilir ancak nesne arayüzlerini bununla sınırlamak doğru değildir.

Nesne arayüzleri genellikle modelleme yapılmak istenildiğinde ve birden çok yazılımcının geliştirdiği projelerde standart bir yapı ortaya konulmak istenildiğinde kullanılır. Eğer bir sınıf bir interface'den türetilmiş ise türetilen interface'ye uygun kod yazmaya bizi zorlar böylelikle standart bir yapı oluşturulur. Bu kadar ön bilgi ve somut örnekten sonra basit bir interface yazalım.

```
<?php

  interface Arayuz {
     const Qwerty = 1;
     public function set($name);
  }
```

```php
interface BaskaArayuz extends Arayuz {
    public function get();
}

class Veri implements BaskaArayuz {

    private $name;

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

    public function get(){
        return $this->name;
    }

}

$interface = new Veri();
$interface->set('Php oop dersleri');
echo $interface->get();

?>
```

Arayuz adında bir interface oluşturduk ve içerisinde *set* metodumuzu birde *Qwerty* adında bir sabit tanımladık. Daha sonra *BaskaArayuz* adında bir interface daha oluşturduk ve bu interface'yi Arayuz ile genişlettik ve get metodumuzu ekledik. Daha sonra Veri adında bir sınıf oluşturup bu sınıfımızı da **"implements"** komutuyla BaskaArayuz interface'sinden türettik ve içine get,set metodumuzu ve name argümanını ekledik.

Eğer Veri sınıfı içerisinde get ve set metotlarımızı tanımlamasaydık tıpkı soyut sınıflar konumuzda olduğu gibi PHP derleyicimiz hata verecekti.

Peki, Arayuz içinde ki sabit değişken ne olacak ?

Aslında onu oraya öylesine koymadım ☺ Burada bir istisnaya değinmek istiyorum. Nesne arayüzleri içerisinde yalnızca **const** komutuyla oluşturulan değerlerin türetilen sınıf içerisinde kullanılması zorunlu değildir. Son olarak ise *$interface* değişkenine Veri sınıfını atadık ve set ve get metotlarını kullandık.

Temel olarak nesne arayüzlerinin çalışma mantığı bu şekildedir. Nesne arayüzlerini ilk öğrendiğim zamanlar bana çok gereksiz gelmişti aslında ancak zamanla kullandıkça öğrenmeye çalıştıkça nesne arayüzlerini daha iyi kavramış oldum ve gerçekten önemli olduğunu anladım. Siz benim gibi yapmayın ☺

8. Nesne Klonlama

Nesne klonlama PHP 5 ile birlikte gelmiştir ve pek fazla bir değişime uğramamıştır. En basit manasıyla bir sınıfı yeniden tanımlamadan kopyalamamızı sağlar.

Bir sınıf çağırdığımızda sınıfın RAM'de tutulduğunu daha önceki konularımızda söylemiştik. Bir sınıfı defalarca çağırmak sistem üzerinde gereksiz bir yük oluşturur ve hantal bir sisteme sahip olursunuz. Maliyet artar. En önemlisi canınız sıkılır. Nesne klonlama tam olarak bunun için vardır. PHP topluluğu canımız sıkılmasın maliyetimiz artmasın diye bu olayı da düşünmüşler sağ olsunlar ☺ Şimdi çok basit bir örnekle nesne klonlama konumuzu netleştirelim.

```php
<?php

class Oop {

   public function hi(){
      echo "Hello World";
   }

}

$sinif = new Oop();
$kopya = clone $sinif;

?>
```

Bu örneğimiz ile konuyu çok güzel özetleyebiliriz. Klasik olarak ne yaptığımıza şöyle bir bakalım. Oop adında bir sınıf oluşturduk hi adında bir metot ekledik. Daha sonra *$sinif* değişkenine Oop sınıfını atadık. Ardından *$kopya* değişkenine clone *$sinif* diyerek Oop sınıfımızı bu değişkene kopyalamış olduk.

Buradan anlaşıldığı üzere bir sınıfı kopyalamak için **clone** komutunu kullanıyoruz. Burada şunu belirtmekte fayda var. *$kopya = $sinif;* şeklinde bir kullanım ile yukarı da yaptığımız olay kesinlikle aynı şey değildir.

Eşittir (=) işareti kullanılarak obje kopyalamak mümkün değildir ! Bu en az bilinen ve en çok yapılan hatalardan biridir. Eğer clone komutu yerine bu hatayı yaparsanız proje illa ki bir yerde patlar yada sınıflar doğru dürüst çalışmayabilir. Bu hatanın en kötü yanı da herhangi bir hata oluşturmamasıdır. Bu yüzden de hatanın nereden kaynaklandığını bulmak maalesef zaman alabilir.

9. Sonsöz

Nesne tabanlı programlama son yıllarda çok hızlı şekilde gelişen bir teknoloji. Hal böyle olunca yalnızca PHP değil hemen hemen tüm dillerde artık projeler OOP mimarisiyle geliştiriliyor. Mevcut projeler bile OOP mimarisine dönüştürülüyor.

PHP 7 ile birlikte obje yapısı daha da geliştirildi ve artık PHP hemen hemen tüm ihtiyaçlarımızı rahatlıkla karşılıyor.

Çoğu iş ilanlarında artık OOP bilen yazılımcı arandığını da görürsünüz. Uzun lafın kısası projelerimizde OOP mimarisini kullanmak son derece önemlidir. Zaten mantığını tamamen kavrayıp el alışkanlığınız oluştuğunda ne demek istediğimi gayet iyi anlayacaksınız ve klasik sistem sizin için o andan sonra bitmiş olacak. Zaten oturup mantıklı düşündüğünüzde kullanmamanız için hiçbir neden yok.

Bu kitapta yada blog üzerinde ki konuları özümseyip kavradığınızda teorik olarak OOP mantığını kavrayacağınızı ve projelerinizi OOP mimarisiyle yazabileceğinize inanıyorum. Umarım bu kitap bir nebze de olsa faydalı olmuştur ve size bir şeyler katabilmişimdir. Görüşmek üzere kendinize iyi bakın ☺