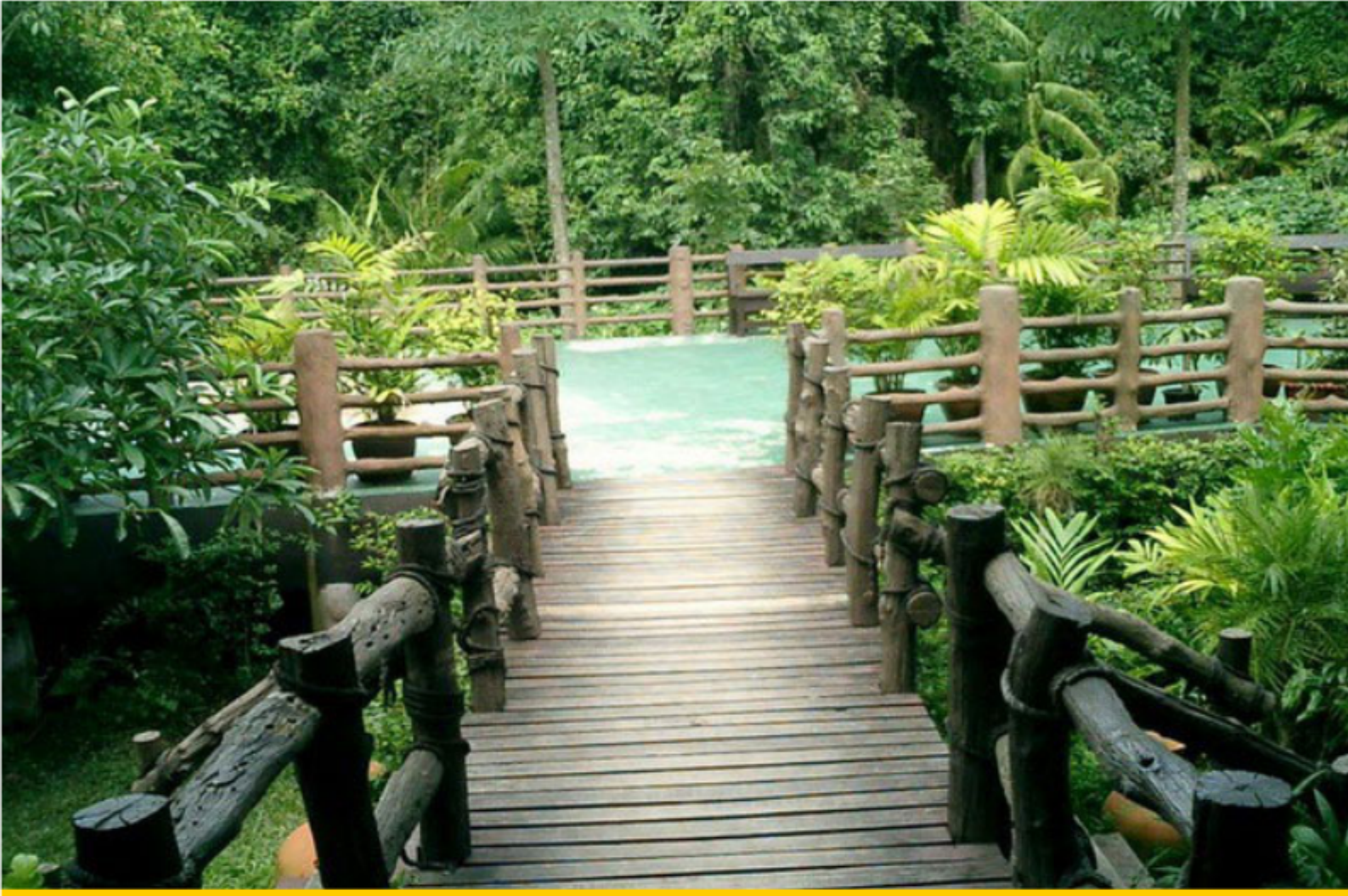

PHP->OOP

Emre Can ÖZTAŞ

www.emrecanoztas.com

Kapak Tasarımı: Emre Can ÖZTAŞ

PHP->OOP

Book Version: 1.0
PHP Version: >= 5.6

Yazan: Emre Can ÖZTAŞ

Kitap Hakkında

Bu kitap; temel ve ileri düzey PHP üzerinden OOP mantığını anlatmak için yazılmıştır. Kitabı takip edebilmeniz için: temel düzeyde olsa PHP bilginizin olması beklenmektedir. Konuların anlatımı basit düzeyde tutulmuştur. Yeni başlayanlar ve ileri düzey geliştiriciler kitabı rahatlıkla takip edebilir.

Memnun kalmanız dileğiyle.

Yazar Hakkında

Emre Can ÖZTAŞ, yazılım geliştirici, web girişimcisi, eğitmen ve yazardır. Çeşitli dillerde yazılım geliştirmektedir. Kendisine ait olan web projeleri vardır ve web girişimcilik alanında çalışmaları sürmektedir. Çeşitli alanlarda birebir veya grup olarak özel eğitimler vermektedir. Genellikle öğrencileri ondan memnundur. Bunun yarısıra; bilgisi ve tecrübesi dahilinde Türkçe kaynağa destek ve öğrenmeye istekli olan herkesin yararlanabilmesi için e-kitap'lar yazmaktadır. Yazdıklarından herhangi bir ücret talep etmemiştir ve kazanmamıştır.

Yazar hakkında daha fazlasını merak ederseniz; kendisi www.emrecanoztas.com adresinde çeşitli konularda yazılar yazmakta ve paylaşımlarda bulunmaktadır. Aynı zamanda kitap hakkındaki; görüş, öneri ve yanlışları da bu adrese iletebilirsiniz.

İçindekiler

BÖLÜM 1: OOP

Bu bölüm boyunca; OOP ve PHP'nin OOP desteğinden konuşacağız. Öncelikle cevabını vermemiz gereken bir soru var. OOP nedir?

OOP (Object Oriented Programming), herhalde son 10 yılın gözde programlama mantıklarından biridir desek yanlış olmaz. Günümüzde kullanılan programlama dillerinin büyük bir çoğunluğu ve belki de hepsi OOP desteğine sahip. Peki OOP nedir, ne işe yarar? Baştan şunu belirtelim. PHP yazmak için OOP gerekmez. Lakin yazmanız bence hoş ve güzel bir programlama alışkanlığıdır. OOP yazın, mantığını iyi kavrayın. Çünkü OOP mantığını anladığınız zaman; diğer bir programlama dilinde gördüğünüz OOP kavramına çok kolay adapte olacaksınız. Çünkü hepsinde mantık aynıdır.

Bir çok şey konuştuk lakin OOP'den bahsetmedik. Peki, o halde şimdi bahsedelim. OOP, bir programlama mantığıdır dedik. OOP aslında gerçek hayatın programlama dünyasına uyarlanmasıdır. Örneğin bir bisikletiniz var ve bu bisikletin de bir mekanik aksağamı var. Yani bir çalışma prensibi var. İşte bu bisikletin çalışma prensibini programlama diliyle anlatabilmek için OOP mantığına ihtiyaç var. Yani hayatı programlama dünyasına aktarabilmek için.
OOP'yi başka bir yönden ele alalım. OOP tasarladığınız yapıyı parçalara ayırmaya (modül) ve her parçanın diğer parçalarla ilişkiye girmesini sağlar. Yine bisiklet üzerinden konuşmak istersek; bu bisikletin tekerlekleri ayrı bir yapıdır. Pedallar, gidon, sele v.s hepsi birbirinden ayrı yapılardır. Bu yapılar tasarlanır, oluşturulur. Daha sonra bu yapıların birleştirilmesiyle ortaya bisiklet çıkar. İşte OOP yapısı da buna benzer.

OOP yapısının bir çok avantajı vardır. Şayet hayatınızı programcı olmaya adamışsanız veyahut "Arkadaş, bu iş benim hayatım!" diyorsanız, mutlaka OOP konusunu çok iyi öğrenin. Çünkü artık OOP olmadan program yazılmıyor. Hiç iş tecrübenüz yoksa size biraz iş dünyasından bahsedeyim. İş dünyasında yapılacak olan proje, öncelikle tasarlanır ve daha sonra modüllere ayrılır. Bu ayrılan modüller de programcılara yazılmaları için dağıtılır. Her yazılan modül projeye dahil edilir ve sonunda ortaya proje çıkar. Tıpkı bisiklet örneğinde olduğu gibi.

İş dünyasında, yapılacak olan projeler modüller üzerinden yürür dedik. Hiç kimse kalkıpta size "Al, bu işi de sen yap!" demez. Ya da projenin tamamını size vermezler. Böyle bir şey olduğu durumda ya çok iyi programcısınızdır, ya eleman yoktur, ya proje küçüktür ya da PM yani "Project Manager" işten anlamıyordur. O yüzden dikkatli olmakta yarar var.

OOP kodlarınızdaki bütünlüğü sağlar. Ve ayrıca yazdığınız kodların bakımını ve değiştirilmesini kolaylaştırır. İş dünyasında küçük projeler önünüze gelmez. Öyle büyük projeler gelir ki o "büyük resmi" düşünemezsiniz bile. Çok karmaşıktır. İşte burada devreye modüller girdiği zaman yaptığınız iş belli olur. Yani ne yapacağınızı bilirsiniz ve ona göre hareket edersiniz.

OOP mantığı, PHP 5'ten itibaren girmiştir. PHP'nin yol haritasına baktığınız zaman OOP'ye doğru gittiğini rahatlıkla görebilirsiniz. Lakin daha önce de dediğimiz gibi;

"OOP olmadan da PHP yazabilirsiniz". Örneğin "Ruby" programlama dilinde herşey OOP mantığına göre tasarlanmıştır. Yazdığınız "5" sayısı bile bir Object (Nesne)'dir.

Herneyse, çok konuştuk. Lakin konuşacağımız daha çok şey var. OOP mantığını tam anlamamış olabilirsiniz fakat ilerleyen bölümlerde daha iyi anlayacağınıza eminim.

BÖLÜM 2: Class

Class, Türkçe kelime anlamı olarak; Sınıf manasına gelmektedir. OOP dünyasında Class en temel yapıdır. Class'lar size bir taslak sunar. Bu taslak ile kodlarınızı yapılandırırsınız ve kodlarınızın bütünlüğünü sağlamış olursunuz. Aynı işi yapmak üzere özelleşmiş üyeleri veya elemanları birarada tutar. Diğer taraftan; class'lar olmadan OOP yapısını kuramazsınız.

Bir class aşağıdaki şekilde tanımlanır.

```
class ClassName {

}
```

Yukarıdaki yapımızı açıklayalım. Class'lar, class keyword (anahtar kelime) ile tanımlanırlar. Buradan bile class'ların bir tür olduğunu anlayabilirsiniz. Her class'ın bir name (isim)'i olur. Class için herhangi bir ismi verebilirsiniz lakin programlama dünyasında genel-geçer kurallardan birisi: class için isim veya isim soylu bir kelime-kelime grupları seçmeniz yönündedir. Yani bir class'a Yaz, Çiz, boz veya diğer herhangi bir fiil veya sıfat vermeniz mantıksız olur. Bir diğer genel-geçer kural ise: class'a verilecek kelimelerin baş harfleri büyük yazılır. Eğer class ismi bir kaç kelimeden oluşuyorsa her kelime birbirine bitişik ve her kelimenin baş harfi büyük yazılır. İsimlendirme konusu size kalmış. Bu kurallar genel olarak OOP programla yapan kullanıcıların seçtiği bir yöntemdir. Bu sayede kodun okunabilirliği artmış olur. Lakin iki veya daha fazla kelimenin bitişik yazılması kat'i bir kuraldır.

Class'ların scope (etki alanı) alanlarını küme parantezleri { } belirler. Buradaki küme parantezleri, class'ların sınırlarıdır. Class'in içerisinde olanlar class'a aittir. Bu üyeleri kullanabilmek için class'lar ile konuşmak gerekir. Bu konuşma işlemini de bir sonraki bölümde gerçekleştireceğiz.

Bir class içerisinde; fonksiyonlar ve değişkenler olabilir. Biz bunlara kısaca üye diyeceğiz. Yapacağımız işlemleri bu üyeler ile gerçekleştireceğiz. Bir class içerisinde, örneğin aşağıdaki gibi bir yapı kullanamayız.

```
class ClassName
{
    echo "Hello World!";
}
```

Yukarıdaki yapımızda; class içerisinde doğrudan echo komutu kullanılmaz. Sadece "echo" komutu değil, değişken veya fonksiyonlar dışında herhangi bir kod parçası yer alamaz. Çünkü class özel bir yapıdır ve kendine ait kuralları vardır. Yani klasik olarak yazdığımız PHP gibi değildir.

Aşağıdaki yapımızı inceleyelim.

```
class ClassName {

    $var1, $var2, $var2....

    function funcName1 () {
        // yapilacaklar...
    }

    function funcName2 ($params) {
        // yapilacaklar...
    }

    function funcName3 ($params1, $params2) {
        // yapilacaklar...
    }

    ...
}
```

Bir class içerisinde; istenildiği kadar üye oluşturulabilir. Burada herhangi bir sınırlama veya kısıtlama yoktur. Lakin aynı işi yapan üyeleri aynı sınıfta toplamak iyi bir alışkanlıktır. Zaten class'ların mantığı da budur. Aynı işi yapan özelleşmiş üyeleri birarada tutmak.

Bir .php uzantılı dosyada birden fazla class olabilir. Aşağıdaki yapımızı inceleyelim.

```
class ClassName1 {

}

class ClassName2 {

}

class ClassName3 {

}

class ClassName4 {

}

class ClassName5 {

}

...
```

Yukarıdaki yapımızda; bir .php uzantılı dosya içerisine birden fazla class açılmıştır. Herhangi bir class sayı sınırlaması yoktur. İstenildiği kadar class oluşturulabilir.

Son olarak göstermek istediğim şey ise include. Hani bilirsiniz. Klasik olarak PHP'de bir dosyayı import (eklemek) etmek için "include" kelimesi kullanılır. Class'lar da bir .php uzantılı dosya içerisinde yer aldığı için, bu class'ların bulunduğu dosyaları istediğimiz dosyaya ekleyebiliriz.

Örneğin aşağıdaki yapımızı inceleyelim.

```
<?php
# c1.php
class ClassName1 { }
?>

<?php
# c2.php
class ClassName1 { }
?>

<?php
# c3.php
class ClassName1 { }
?>

<?php
# main.php
include 'c1.php';
include 'c2.php';
include 'c3.php';
?>
```

Yukarıdaki yapımızda da görüldüğü üzere; class içeresen, .php uzantılı dosyalar istenilen herhangi bir .php uzantılı dosyaya include edilebilirler.

Bu bölümde temel olarak class'lar üzerinde durduk. Henüz class'ları kullanmaya başlamadık. Bir sonraki bölümde, yani Object konusunda oluşturmuş olduğumuz class'ları kullanmaya başlayacağız. Bu bölümün amacı sadece class yapısını göstermekti. Henüz bir şey öğrenmedik. Sadece temel olarak class yapısından bahsettik. Burada gördüklerimiz aslında temel şeyler. İlerleyen bölümlerde bu gördüklerimizin üzerine çıkacağız.

BÖLÜM 3: Object

Object (Nesne), bir class ile iletişime geçmenin tabir-i caizse "konuşmanın" yoludur. Object'ler aktif birer elemanlardır. Bu elemanlar yardımıyla class'ın sahip olduğu üyelere erişilebilir ve bu üyeler kullanılabilir.

Şimdi sizden bir fabrika düşünmenizi istiyorum. Bu fabrika herhangi bir şey üretebilir. Ayrıca fabrika içerisinde; değişik üretim birimleri vardır. Bu birimlerde üretilen ürünler, birleştirilir ve daha sonra dış dünyaya satılmak-kullanılmak için açılır.

Şimdi OOP konusuna geri dönelim. Burada fabrika, class'lardır. Bir işi yapmak, bir şeyi üretmek için özelleşen fabrikalar gibi class'lar da özelleştirilir. Fabrikaların üretim birimleri olduğu gibi class'ların üretim birimleri ise; function'lardır. Bu fonksiyonlar, class'ların belli bir işi yapması için özelleşmesini sağlar. Fabrikada çalışan ara elemanlar vardır. Class'ların ise variables (değişkenler) vardır. Yani class'ların gerçek hayattaki farbikadan farkları yoktur. Bu sadece fabrikalar için geçerli değil. Hayatta olan her şey OOP dünyasında anlatılabilir. Bunu zaten ilerleyen bölümlerde göreceksiniz. Bu sadece basit bir örnekti. Peki, her şey tamam. Object'ler ne için? Fabrikalarla çalışabilmek için; ilk önce onlarla anlaşmanız gerekir. İşte burada anlaşma işinizi object'ler sağlar.

Basit bir örnek verelim. Örneğin; elimizde bir "araba"nın özelliklerinin saklı olduğu bir yapı olduğunu düşünelim. Class'ımız aşağıdaki gibi olsun.

```
class Car {

}
```

Bu class'ımızın, belirlediği işi yapması için özelleşmesi gerekir. Bu özelleşme işini de fonksiyonlar ile gerçekleştireceğiz. O halde class'ımıza belirli fonksiyonlar ekleyelim.

```
class Car {

    function  carCompany ()
    {
        echo "Welcome to car company!";
    }

    function carModel ($model)
    {
        echo "Model: " . $model;
    }

    function carColor ($color)
    {
        echo "Color: " . $color;
    }

    function carPrice ($price)
    {
        return ($price*2);
```

```
    }

}
```

Yukarıdaki class'ımızın belirli fonksiyonları var. Daha önce de kararlaştırdığımız gibi class içerisinde olanlar class'a ait ve biz onlara üye diyeceğiz. Class'ın fonksiyonları, parametreli ve parametresiz fonksiyonlardan oluşuyor. Peki class'ımızla iletişime geçelim.

Bir class'tan bir nesne aşağıdaki şekilde oluşturulur.

```
$objectName = new ClassName;
// veya
$objectName = new ClassName();
```

Şimdi yukarıdaki yapımız üzerinde biraz duralım. Bir nesne oluşturulurken; ilk olarak bu nesnenin ismi belirlenir. Nesnenin ismi; tipik olarak değişken tanımlaması gibidir. Object isimlendirmesinden sonra bu object'in kime ait olduğunu, kimi işaret edeceğini veyahut hangi class'la iletişime geçeceği belirtilmelidir. O yüzden object isminden sonra; eşittir işareti (=) konur ve daha sonra new anahtar kelimesi yazılır. Daha sonra da class ismi belirtilir. new kelimesi özel bir kelimedir. "new" ile aslında bir aitlik belirtimi yapılır. Veyahut; yeni bir türetilme gerçekleştirilir. Burada object'imizi bir class'tan türetiyoruz. Yani o object için hafızada (heap alanı) bir alan oluşturuyoruz ve o alan artık işaret ettiğimiz class'ı göstermeye başlıyor.

OOP dünyasında; bir class'tan oluşturulan object için instance deyimi kullanılır. Yani o class'ın bir örnek'i.

Object tanımlamasında gördüğünüz gibi iki farklı yapı var. Bu yapılardan şimdilik bahsetmeyeceğiz. İlerleyen konularda daha detaylı bahsedeceğiz. O yüzden şimdilik ikinci yapımızı kullanacağız.

Object isimlendirmeleri yapılırken; verilen kelimenin tümü küçük harfle yazılır. Verilecek isim iki veya daha fazla kelimede oluşuyorsa; ikinci kelimeden itibaren, her kelimenin ilk karakteri büyük harfle yazılır.

Buraya kadar heşey tamam. Şimdi class'ın üyelerine nasıl erişebiliriz, bu konu üzerinde konuşalım. Object oluşturduktan sonra, bir class'ın üyelerine erişmek için; object yapısı kullanılır. Şimdi aşağıdaki yapımızı inceleyelim.

```
$objectName->classMemberName();
```

Bir class'ın üyelerine erişmek için öncelikle oluşturulan object ismi yazılır ve daha sonra -> karakterleri konur. Daha sonrada; Object'in ait olduğu class'ın üyesinin adı yazılır. Bu üye; değişken veya fonksiyon adı olabilir. Şimdilik değişkenler konusundan bahsetmeyeceğiz. Çünkü değişkenlerin kullanımı ve çağrımında biraz farklılıklar var. Değişkenler için ayrı bir bölüm ayıracağız.

Herneyse, -> karakteri dikkat ederseniz bir ok işaretine benziyor. OOP'de de aynen öyle. Bir class'tan oluşturduğumuz object,-> işareti ile o class'ın üyesini icraa ediyor veyahut çağırıyor.
Diğer programlama dillerinden gelenler için -> karakteri, nokta (.) karakteri ile aynıdır.
Peki, yazdığımız class'tan bir nesne oluşturalım ve o class'ın üyelerini çağıralım.

```
// object oluşturuldu.
$carObject = new Car();
// uye cagirildi.
$carObject->carCompany();
```

Yukarıdaki yapımızda; Car class'ında, carObject adında bir object oluşturduk. Daha sonra da carObject'i kullanarak, carCompany () fonksiyonunu çağırdık. Her şey bu kadar basit yani.

Kodlarımızın tam hali de aşağıdaki gibidir.

```
<?php
    /*
        classes.php
    */
    class Car {

        function  carCompany ()
        {
            echo "Welcome to car company!";
        }

        function carModel ($model)
        {
            echo "Model: " . $model;
        }

        function carColor ($color)
        {
            echo "Color: " . $color;
        }

        function carPrice ($price)
        {
            return ($price*2);
        }

    }

    // object oluşturuldu.
    $carObject = new Car();
    // uye cagirildi.
    $carObject->carCompany();

?>
```

Yukarıdaki yapımız hakkında biraz konuşalım ve daha sonra ekran çıktısına bakalım. Yukarıdaki yapımız; .php uzantılı bir dosya içerisinde. Class'ımızı yazdık. Class'ımızın scope (etki) alanı dışında; object'imizi oluşturduk ve class'ın üyesini çağırdık. Buraya dikkat edin. Class içerisinde herhangi bir object tanımlaması gerçekleştirmedik. Çünkü object'in kullanılabilmesi için bağımsız bir yerde olması gerekir. Yani class'ın içerisinde tanımlarsak (tanımlamamıza izin vermez) kodlarımız çalışmaz.

Şimdi yazdığımız kodların ekran çıktısına bir bakalım.

```
Welcome to car company!
```

Ekran çıktımızda da görüldüğü gibi class'ın sahip olduğu carCompany () fonksiyonunu başarıyla çağırdık.

Class'ın sahip olduğu diğer fonksiyonları da çağırabiliriz. Burada; carModel ($model) ve carColor ($color) fonksiyonları bir parametre almış. Tabi ki bu fonksiyonları çağırırken parametre değerlerini de göndermemiz gerekiyor. Şimdi bu fonksiyonları da çağıralım.

```
$carobject->carModel("Audi A4");
$carObject->carColor("Black");
```

Burada şöyle bir şey de yapabiliriz. Fonksiyonlara default (varsayılan) parametre değerlerini atayarak, daha sonraki çağırma işleminde boş parametre gönderebiliriz. Yani demek istediğim:

```
function carModel ($model = "Mercedes")
{
    echo "Model: " . $model;
}

function carColor ($color = "White")
{
    echo "Color: " . $color;
}
```

şeklinde parametrelere varsayılan değer atandıktan sonra;

```
$carObject->carModel();
$carObject->carColor();
```

şeklinde yazabiliriz. Yani tipik PHP mantığı.

carPrice ($price) fonksiyonu, kendisine gelen değeri return etmekte. Burada da uygulayacağımız yine tipik PHP mantığı. Yani gelen değeri doğrudan yazdırabileceğimiz gibi herhangi bir değişkene de atayabiliriz. Biz bir değişkene atayalım.

```
$price = $carObject->carPrice (80);
```

Görüldüğü gibi her şey çok basit ve net. Sadece işlemlerimizi object kullanarak gerçekleştiriyoruz.

Burada size göstermek istediğim başka bir şey var. Herhangi bir class'tan oluşturulan bir object'in ait olduğu class yapısını; var_dump(), var_export() veya print_r() fonksiyonları ile öğrenebiliriz. örneğin bizim class'ımızın object'i için print_r() fonksiyonunu kullanalım.

```
print_r($carObject);
```

Ekran çıktımıza bir bakalım.

```
Car Object ( )
```

Görüldüğü gibi carObject, Car sınıfınfan türetilen bir object.

3.1 Birden Fazla Object Oluşturma

Her class için sadece bir object oluşturulur diye bir kural yoktur. Bir class'tan istenildiği sayı da object oluşturulabilir. Örneğin Car class'ı için konuşursak:

```
$carObject1 = new Car();
$carObject2 = new Car();
$carObject3 = new Car();
....
```

Şeklinde birden fazla object oluşturulabilir ve hepsi de Car class'ından oluşturulan object'lerdir. Yani hepsi aynı class'ı işaret eder. Ayrıca, oluşturulan her object aynı yetkilere sahiptir. Yani hepsi class'ın üyelerini kullabilir.

Oluşturulan her object'in, hafızada kendine ait olan bir alanı var. Bundan daha önce bahsetmiştik. new anahtar kelimesi, hafızada bir alan ayırır diye. Burada object, heap alanında tutulur. Yani stack alanındaki gibi yaz-sil bir yapıda değildirler.

3.2 Parametre Olarak Object

Burada anlatacaklarımızı dikkatli takip etmenizi öneririm. Zira biraz sonra bahsedeceğimiz konuyu her hangi bir yerde bulmanız biraz zordur. Benden söylemesi.

Object için; bir class'ın instanstance olduğundan bahettik. Bir class içerisinde değişkenler ve fonksiyonlar bulunabileceğinden de bahsettik. Üstelik class içerisindeki fonksiyonlardan hiç bir farkının olmadığından da bahsettik. Object konusunda şöyle bir şey var; bir class'tan oluşturulan object herhangi bir fonksiyona parametre olarak gönderilebilir. Yani; elimizde bulunan class'lardan oluşturulan object'ler, yine aynı class içerisindeki bir fonksiyona gönderilebileceği gibi başka bir class içerisindeki fonksiyona da gönderilebilir.

Bir class'tan object oluşturup bunu başka bir class içerisinde kullanmanın mantığı nedir? Şeklinde bir soru aklınızdan geçmiş olabilir. Haklısınız. Bir class içerisinde başka bir class'a ait olan üyeler ve durumlar olabilir. Örneğin; aynı işi yapan üyeleri ayırdığımız class'ların bazı durumlarda, deyim yerindeyse; birbirleriyle konuşmaları gerekir. İşte böyle durumlarda parametre olarak object'ler kullanılabilir. Daha akılda kalıcı olması için basit bir örnek yapalım. Örneğimiz de aşağıdaki gibi olsun.

```
<?php

class Main {

    function mainWork(Cleaner $calcObj, $str)
    {
        $writerObj = new Writer();
        $writerObj->writeStr($str);

        $result1 = $calcObj->replaceStr($str);
        $writerObj->writeStr($result1);
        $result2 = $calcObj->trimStr($str);
        $writerObj->writeStr($result2);
    }

}

class Writer {

    function writeStr($str)
    {
        echo $str . "<br>";
    }

}

class Cleaner {

    function trimStr($str)
    {
        $str = trim($str);
        return $str;
    }

    function replaceStr($str)
    {
        $str = str_replace(" ", "*", $str);
        return $str;
    }

}

$calcObj = new Cleaner();
$mainObj = new Main();
$str = "   emrecan-oztas    ";

$mainObj->mainWork($calcObj, $str);
```

```
?>
```

Yukarıdaki örneğimizi öncelikle inceleyelim. Örneğimizde neler yaptık? Aslında yapmış olduğumuz örneğimiz çok basit. 3 tane class'ımız var. Bu class'lar: Main | Writer | Cleaner. Main class'ımızdaki mainWork ( ) fonksiyonu iki tane parametre alıyor. Aldığı parametrelerden birisi ilginç değil mi? Cleaner $calcObj şeklinde bir parametre. Bu parametre aslında bir object. Parametrenin başına yazmış olduğumuz class ismi bu parametrenin kendisinden türetilmiş olan bir object olduğunu belirtiyor. Buraya dikkat! Fonksiyona parametre olarak gönderilen bu object ile object'in ait olduğu class'ın üyelerine erişebiliriz. Yani başka bir yerde oluşturulan bir object, istenilen bir class'ın istenilen herhangi bir fonksiyonuna parametre olarak gönderilebilir. Bu da fonksiyona büyük bir ayrıcalık tanıyor. Herneyse, örneğimizi incelemeye devam edelim. MainWork ( ) fonksiyonuna gelen $calcObj object'i yardımıyla, parametre olarak gelen $str değerini Cleaner sınıfındaki ilgili fonksiyonlar yardımıyla çeşitli işlemler yaptırıyoruz. Her şey bu kadar basit ve net yani.

3.3 get_class ( )

Bazen bazı durumlarda, object'lerin hangi class'a ait olduğunu bilmediğimiz durumlarla karşılaşabiliriz. Böyle durumlarda imdadımıza PHP'nin tanımlı olan fonksiyonları yetişir. Bu fonksiyonlardan birisi de get_class ( ) fonksiyonudur. get_class ( ) kendisine parametre olarak gönderilen herhangi bir object'in hangi class'tan oluşturulduğu bize söyler.
get_class ( ) fonksiyonunun kullanımı aşağıdaki gibidir.

```
get_class ($object parameter)
```

Yukarıdaki satırdan sonra dönen değer bir class ismi olacaktır. Yani string bir değer. O yüzden bu satırımızı bir değişkene bağlamamız yerinde bir karar olacaktır.

```
$className = get_class ($object parameter);
```

Tabiki doğrudan da yazdırabiliriz.

get_class () fonksiyonunu kullanmak yerine; var_dump ( ) | var_export ( ) | print_r ( ) fonksiyonlarından da birisini kullanabilirsiniz.

3.4 instanceof

instanceof bir kontrol deyimidir aslında. Bir object'in bir class'a ait olup olmadığını gösterir. Tabiki sonuçları: true veya false olarak gösterir. instanceof fonksiyonun kullanımını aşağıdaki örnekte görelim.

```
if($calcObj instanceof Cleaner)
{
    echo "yes";
} else {
    echo "no";
```

```
}
```

Yukarıdaki örneğimizde: $calcObj instanceof Cleaner ifadesi if içerisinde kontrol edilmiştir. Sol taraf object, sağ taraf ise class ismi olacaktır. Belirtilen object belirtilen class'a ait ise yes değilse false yazacaktır.

BÖLÜM 4: Access Modifiers

OOP felsefesinde, her üyenin erişime açık olması istenmez. Çünkü bir üye erişeme açık ise tehlikeleri de beraberinde getirir. Zira dışarıdan erişime açık olan bir üye değiştirildiği zaman veya doğrudan müdahale edildiği zaman sistemin işleyişi de değişecektir. Dolayısıyla bu ve buna benzer durumlarda, bir class'ın üyeleri dışarıdan erişime kapatılır. Bir üyenin dışarıdan erişeme kapalı olması durumunu da Access Modifiers yani Erişim Belirleyicileri ile kontrol edilir.

Access Modifiers olarak 3 farklı durum vardır. Bunlar; public | protected | private idir. Bir dördüncü durum ise default durumudur. Yani herhangi bir ifade yazılmamasıdır. Böyle olduğu zaman ise; üyeler public yani herkese açıkmış gibi davranır. Lakin bir class'ın variable (değişkenleri), Access Modifiers olmadan tanımlanamaz. Bu konuya daha sonra detaylı olarak değineceğiz.

Hazırsanız, o halde Access Modifiers konusuna başlayalım.

4.1 public

public kelime anlamı olarak halka açık, genel manalarına gelmektedir. OOP dünyasında da aynen bu tanıma uygun bir yapısı vardır. Bir class'ın üyeleri, public tanımlandığı durumda, bu üye dışarıdan müdahaleye açılmış durumdadır. Yani isteyen herkes bu üyeye erişim sağlayabilir.

public bir keyword (anahtar kelime) idir. Bir üyenin herkese açık olması istenildiğinde, bu üyenin başına public yazılması kafîdir.

Aşağıdaki yapılarımıza bakalım.

```
<?php
public $var1 = 06;
public $var2 = "Turkey / Ankara";

public function countNumber ()
{

}

public function getMemberNumber ()
{

}

public function getSessionId ()
{

}
```

Yukarıdaki yapımızda görüldüğü gibi class'ın tüm üyeleri public durumdadır. Dolayısıyla buradan hareketle bu class'ın üyelerine dışarıdan erişim gerçekleştirilebilir ve herhangi bir sorun ile karşılaşılmaz.

Örneğin; class'ın properties (class değişkenleri)'ine, bu class'tan bir object oluşturarak erişebiliriz ve değerlerini değiştirebiliriz.

Aşağıdaki yapımızı inceleyelim.

```
$classObj = new ExampleClass();
echo $classObj->var1;
echo $classObj->var2;
```

Yukarıdaki yapımızda; ExampleClass class'ından $classObjt isimli bir object oluşturduk ve bu object yardımıyla class'ın üyeleri olan; $var1 ve $var2 değişkenleri yazdırdık. Dilersek bu değişkenlerin değerlerini değiştirebiliriz.

```
$classObj->var1 = "Turkey / Ankara";
$classObj->var2 = 06;
```

Yukarıdaki örneğimizde; dışarıdan class'ın iki değişkenin de değerlerini birbirleriyle değiştirdik. Herhangi bir proble karşılaşmadık.

Örneğimizin tamamız aşağıdaki gibi olacaktır.

```
<?php

class ExampleClass {

    public $var1 = 42;
    public $var2 = "Turkey / Konya";

    public function countNumber ()
    {

    }

    public function getMemberNumber ()
    {

    }

    public function getSessionId ()
    {

    }

}

$classObj = new ExampleClass();
echo $classObj->var1;
```

```
echo "<br>";
echo $classObj->var2;
echo "<br>";
$classObj->var1 = "Turkey / Ankara";
$classObj->var2 = 06;
echo $classObj->var1;
echo "<br>";
echo $classObj->var2;
```

Yukarıdaki örneğimizin ekran çıktısı da aşağıdaki gibi olacaktır.

```
Turkey / Konya
42
Turkey / Ankara
6
```

Ekran resmimiz aslında public konusunda ne demek istediğimizi özetliyor. Ama yine de değinelim. İlk değerler, değişkenlerin ilk hali ikinci değerler ise değişkenlerin sonraki yani biz değiştirdikten sonraki halidir.

OOP dünyasında, genellikle bir üye tanımlanırken bu üyenin public olarak tanımlanmasından kaçınılır. Çünkü dışarıdan erişimde, herhangi bir değişim gerçekleşirse bu class'ın daha doğrusu tüm sistemin işleyişi bozulabilir. Lakin sizin burda takip etmeniz gereken kural: önemli olan, sistemin işleyişine doğrudan katılan üyeleri public yapmamalısınız. Bu üyelerin değerlerini araya yazacağınız belli fonksiyon ve üyeler ile yapmalısınız. Tabiki bu üye ve fonksiyonları public olarak yazabilirsiniz.

4.2 protected

protected kelime anlamı olarak; korumalı, korunmuş manalarına gelmektedir. public anahtar kelimesinden farklı bir işleyiş olarak protected üyeler dışarıdan erişime kapatılmıştır. Yani public'te olduğu gibi dışarıdan erişime kapalıdır. Dışarıdan herhangi bir müdahale söz konusu değildir.

protected üyeler iki durumda kullanıma açıktır. Bunlar:

1. Aynı class içerisinde, diğer üyelerin kullanımına açıktır.
2. Inheritance (Kalıtım) yoluyla alt sınıflarda erişe açıktır.

Şimdilik Inheritance (Kalıtım) konusundan bahsetmedik. O yüzden Inheritance (Kalıtım) konusu için class'ın özelliklerinin diğer class'lara aktarılması olarak düşünebilirsiniz. Bir class'ın özellikleri yani üyeleri diğer class'lara aktarılır ve o class'ın üyesiymiş gibi kullanılabilir. Bu konudan ilerleyen bölümlerde sıklıkla bahsedeceğiz.

protected üyeler aynı class içerisinde kullanımında herhangi bir sıkıntı yoktur. Zaten protected'ın amacı: dışarıdan erişeme kapalı olup aynı class içerisinde bulunan üyelerin kullanımına açılmasıdır.

protected olarak tanımlanan üyelerin dışarıdan erişime kapalı olduğunu kanıtlamak için basit bir örnek yapalım. Örneğimiz aşağıdaki gibi olacaktır.

```
<?php

class ExampleClass {

    protected $var1 = "emrecan-oztas";

    protected function getUniqId ()
    {
        return uniqid();
    }

}

$classObj = new ExampleClass();
echo $classObj->var1;
$result = $classObj->getUniqId ();
```

Örneğimizin ekran çıktısına bakacak olursak;

```
Fatal error: Uncaught Error: Cannot access protected property
ExampleClass::$var1 in /opt/lampp/htdocs/untitled.php:15 Stack trace:
#0 {main} thrown in /opt/lampp/htdocs/untitled.php on line 15
```

protected olan üyelerin çağırımı konusunda hata aldığımızı görebiliriz. Hata mesajında bize dediği: "protected olan üyelere erişem sağlayamazsınız!". Buradan hareketle bir protected üyeye dışarıdan erişim sağlanamaz.

Peki protected olan üyelere erişemi nasıl sağlamalıyız? protected üyelere dışarıdan erişimi, public olarak tanımladığımız üyeler ile sağlamalıyız. Yani biz public olan üyeler ile konuşmalıyız ve public olan üyeler protected olan üyelerle aynı class içerisinde yer alacağı için doğrudan konuşma işlemini gerçekleştirebilirler. Bu dediklerimizin ışığında başka bir örnek yapalım.

```
<?php

class ExampleClass {

    /*  function sessionControl () {}
        olarak, default seviyesinde
        tanımlanabilir.
    */
    public function sessionControl ()
    {
        $this->startSession();
        $this->getSessionId();
        $this->getSessionStatus();
        $this->destroySession();
    }
```

```
    protected function startSession ()
    {
        session_start();
        echo "Session start <br>";
    }

    protected function getSessionId ()
    {
        echo session_id() . "<br>";
    }

    protected function getSessionStatus ()
    {
        echo session_status() . "<br>";
    }

    protected function destroySession ()
    {
        session_destroy();
        echo "Session is destroyed! <br>";
    }

}

$classObj = new ExampleClass();
$classObj->sessionControl();
```

Yukarıdaki örneğimizde; şimdiye kadar görmediğimiz bir anahtar kelime kullandık: this. Şimdilik üzerinde fazla düşünmeyelim. Bir sonraki bölümde detaylı olarak bahsedeceğiz. Lakin this anahtar kelimesinin class içerisindeki üyelerin çağırımı ve erişimi konusunda kullanıldığını bilelim.

Örneğimizin ekran çıktısı aşağıdaki gibi olacaktır.

```
Session start
nmkjqar8fmqk0or0na7hfc54t0
2
Session is destroyed!
```

Görüldüğü gibi public olan bir üye ile protected olan üyelere erişimi sağladık. Yukarıdaki örneğimizde; protected olan üyeleri, public olarak tanımlamış olsaydık örneğin, session başlamadan session'u sonlandırma işlemi gerçekleştirilebilirdi. Yani sistemin işleyişini sekteye uğratabilirdi. Daha öncede dediğim gibi sistemin işleyişinde önemli rolleri olan üyeler public yapılmamalıdır.

4.3 private

private kelime anlamı olarak; özel, kişisel, gizli gibi manalara gelmektedir. OOP dünyasında private anahtar kelimesi, bir class'ın üyelerinin tamamen dış dünyaya kapatılmasını gerçekleştirir. protected anahtar kelimesi, bildiğiniz gibi iki durumda kullanılıyordu. Lakin private, protected anahtar kelimesinden farklı olarak yalnızca bir durumda kullanılabilir. Bu durum ise:

1. Aynı class içerisinde, diğer üyelerin kullanımına açıktır.

private anahtar kelimesi ile tanımlanmış olan üyeler yalnız ve yalnızca aynı class içerisinde kullanılabilir ve bu class'ın üyelerinin kullanımına açıktır.

Dikkat ederseniz; Access Modifiers anahtar kelimeleri 3P ve public > protected > private şeklinde, level (seviye) olarak belirlenmiş durumda. Aşağı doğru inildikçe üyelerin erişim seviyeleri de düşmektedir.

Yine protected konusunda olduğu gibi private konusunu daha iyi anlamak içim, private olarak tanımlanan üyelere dışarıdan erişim sağlamaya çalışalım.

```
<?php

class ExampleClass {

    private $var1 = "emrecan-oztas";

    private function getUsername ()
    {
        return $this->var1;
    }

}

$classObj = new ExampleClass();
echo $classObj->var1;
$result = $classObj->getUsername();
```

Yukarıdaki örneğimizin ekran çıktısı da aşağıdaki gibi olacaktır.

```
Fatal error: Uncaught Error: Cannot access private property
ExampleClass::$var1 in /opt/lampp/htdocs/untitled.php:15 Stack trace:
#0 {main} thrown in /opt/lampp/htdocs/untitled.php on line 15
```

Yukarı ekran alıntısında da görüldüğü gibi private olan bir üyeye dışarıdan erişim söz konusu değildir. Zaten uyarı da dediği şey aşağı yukarı şu şekilde: "private bir üyeye dışarıdan erişim sağlanamaz!"

Peki private olan üyelere dışarıdan nasıl erişim sağlamalıyız? private olan üyelere, public veya inheritance (kalıtım) durumunda protected üyeleri kullanarak erişim sağlamalıyız.

O halde, basit bir örnek yaparak; private metotlara erişimi sağlayalım. Örneğimiz aşağıdaki gibi olacaktır.

```
<?php

class ExampleClass {
```

```
    private $name;
    private $surname;

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

    public function setSurname ($surname)
    {
        $this->surname = $surname;
    }

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

    public function getSurame ()
    {
        return $this->surname;
    }

}

$classObj = new ExampleClass();
$classObj->setName("emrecan");
$classObj->setSurname("oztas");
echo $classObj->getName();
echo $classObj->getSurame();
```

Yukarıdaki örneğimizde; basit olarak private tanımlanmış olan properties (class değişkenleri)'ine public olan fonksiyonlar yardımıyla değer ataması yaptık ve atadığımız değerleri geri çağırabildik. Yani bir nevi private olan üyelere, public fonksiyonları kullanarak erişim sağladık.

Örneğimizde kullandığımız bu yönteme shadowing (gölgeleme) denilmektedir. Bir sonraki bölümde bu konu üzerinde detaylı olarak duracağız.

4.4 default

Daha öncede değindiğimiz gibi Access Modifiers olarak, pekte sayılmayan lakin varolan bir de default durumu var. Yani herhangi bir Access Modifiers'ın yazılmaması. Herhangi bir Access Modifiers yazmayarakta üyelerimizi tanımlayabiliriz. Lakin burada karşımıza şöyle bir sıkıntı çıkacaktır. Properties'ler yani class'ın değişkenleri, herhangi bir anahtar kelime almadan tanımlanamaz. Yani illaki bir Access Modifiers almalılardır. Daha önceleri var anahtar kelimesi ile tanımlamalar yapılabiliniyordu. Hala da yapılabilinir durumda. Lakin var anahtar kelimesi artık depressed yani kullanım dışı duruma getirilmiştir. Kısa bir zaman sonra var anahtar kelimesi kaldırılacaktır. O yüzden kullanımını pek tavsiye etmiyorum.

BÖLÜM 5: Properties

Class değişkenlerine properties denilmektedir. Bu bölüme kadar class değişkenlerinde bahsetmedik. Şimdi sırası geldi. Artık class değişkenlerinden de bahsedelim. Peki class değişkenleri yani properties nedir? Bir class içerisinde, herhangi bir yapıya ait olmadan doğrudan class'ın kendi değişkenleridir.

Şöyleki:

```
class PropertiesClass {

    $var1;
    $var2 = "emrecan-oztas";
    $var3 = 639;

    public function getVal()
    {
        $value = "Turkey";
        return $value;
    }

}
```

şeklindeki oluşturmuş olduğumuz bu yapıda; $var1, $var2 ve $var3 birer class değişkeni. Lakin getVal () fonksiyonundaki; $value değişkeni bir class değişkeni değil. Neden? Çünkü bu değişken doğrudan class içerisinde oluşturulmayıp, bir fonksiyon içerisinde oluşturulmuştur. Dolayısıyla bu değişken, fonksiyonun bir değişkenidir.

Daha önceki bölümlerde bahsettiğimiz bir kural vardı. Bu kural; bir class içerisinde oluşturulan değişken mutlaka bir Access Modifiers almalıdır. Peki Access Modifiers'lar nelerdir? Bunları da 3p olarak nitelendirmiştik. Yani; public | protected | private. Bu 3 erişim belirleyiciden herhangi birini almadan bir class değişkeni oluşturulamaz. Buna çok dikkat edelim.

Şimdi anlattıklarımızın ışığında class değişkenlerimizi düzenleyelim.

```
public $var1;
protected $var2 = "emrecan-oztas";
private $var3 = 639;
```

Peki bu 3 erişim belirleyici olmadan properties tanımlayamaz mıyız? Bu sorunun cevabı bir sonraki alt başlıkta.

5.1 var

var anahtar kelimesi, bir properties tanımlamasında kullanılır. Daha doğrusu kullanılırdı dememiz daha doğru olacaktır. Çünkü var anahtar kelimesi depressed haline getirildi. Yani kısa bir zaman sonra PHP'den atılacak. Şuanlık kullanımda. Buna PHP 7 de dahil.

Giriş bölümünde oluşturduğumuz properties'lerimizi yeniden var anahtar kelimesiyle de aşağıdaki şekildeki gibi tanımlayabiliriz.

```
var $var1;
var $var2 = "emrecan-oztas";
var $var3 = 639;
```

var anahtar kelimesinin yakın bir zamanda PHP'den atılacağından bahsettik. Dolayısıyla kodlarınız arasında bulunmamasını tavsiye ederim.

5.2 $this

Class değişkenleri, class içerisinde doğrudan kullanılamazlar. Örneğin değişkeni tanımlayıp bunu getireyim bir fonksiyon içerisinde kullanayıp v.s gibi durumlar söz konusu değildir. Dolayısıyla burada class değişkenini kullanmak için bir anahtar kelime lazım gelir. Bu kelimede $this anahtar kelimesidir. $this anahtar kelimesi ile tanımlanan properties istenilen herhangi bir yerde kullanılabilir.

Basit bir örnek ile bu dediklerimizi anlatmaya çalışalım. Örneğimiz aşağıdaki gibi olacaktır.

```
<?php

class Circle {

    private $piValue = 3.14;

    public function calcCircleArea ($r)
    {
        $result = $this->piValue * ($r * + $r);
        return $result;
    }

    public function calcCircleAround ($r)
    {
        $result = 2 * $this->piValue * $r;
        return $result;
    }

}

$circleObj = new Circle ();
echo $circleObj->calcCircleArea(5);
echo $circleObj->calcCircleAround(5);
```

Yukarıdaki örneğimizi inceleyelim. Class'ımızın properties'i $piValue değişkeni. Bu değişken dışarıdan erişime karşı private olarak tanımlanmış durumda. Bu değişkeni çağırmak için: $this anahtar kelimesini kullanıyoruz. Burada dikkat ederseniz; $this->piValue şeklinde kullanıyoruz. Başka bir değişken de olsaydı yine aynı şekilde kullanacaktık. Yani bu kullanım temel bir kullanım. $this anahtar kelimesi class'ı işaret etmekte. Yani bir object'teki gibi -> işaretlerini kullanıp bu class'ın değişkenini

çağırıyoruz. Veyahut işaret ediyoruz. -> işaretlerinden sonra değişkenin adının, $ işareti olmadan yazıyoruz. Buna çok dikkat edelim.

Gelelim class'ımızın yaptığı işe. Class'ımız temel olarak bir dairenin alanını ve çevresini hesaplama işlemini yapmaktadır.

5.3 const

const yani constant bir class içerisinde sabit değişkenler tanımlamak için kullanılan keyword (anahtar kelime)'idir. OOP kullanmadığımız zaman standart olarak bir sabit değişken şu şekilde tanımlanır:

```
define("VARIABLE_NAME", "VALUE")
```

Lakin bir class içerisinde sabit bir değişken const anahtar kelimesi ile tanımlanır. Daha doğrusu tanımlanmak zorundadır. Tanımlanan bu değişken de $ işareti almaz. Şöyleki;

```
const VAR_ONE = 25;
const NAME = "emrecan-oztas";
const PYS = TRUE;
```

Sabit değişkenler tanımlanırken; değişkenin tüm isimleri büyük yazılır. Eğer iki veya daha fazla kelimeden oluşuyorsa isimlendirme, her kelimenin arasına alt çizgi (_) konulur. Tabi ki bu defacto bir kuraldır. Sadece kodların okunabilirliğini arttırmak için yapılır.

PHP, kendi kodlarında case sensitive (büyük/küçük harf duyarlılığı) özelliğine sahip olmadığı için; boolean değerler (true / false) büyük harfle yazılması iyi bir programcılık alışkanlığıdır.
Sabit değişkenlere, primitive (ilkel) tipler atanmalıdır. Ayrıca Sabit değişkenlere object ataması yapılamaz.

Sabit değişkenler, Access Modifiers almazlar. Çünkü gerek yoktur.

Bir class tanımlayalım. Bu class'a ait olan sabit değişkenler tanımlayalım. Class'ımız aşağıdaki gibi olsun. Bu örnekten göstermek istediğim şey: sabit değişkenler, normal değişkenler gibi çağrılmazlar ve kullanılmazlar. Herneyse örneğimize bir bakalım.

```
<?php

class ConstClass{

    const PI_VALUE = 3.14;
    const USERNAME = "emrecan-oztas";
    const STATUS = TRUE;

    public function writeConstVal ()
    {
        echo self::PI_VALUE;
        echo self::USERNAME;
```

```
        echo self::STATUS;
    }

}

$constObj = new ConstClass();
$constObj->writeConstVal();

echo ConstClass::PI_VALUE;
echo ConstClass::USERNAME;
echo ConstClass::STATUS;
```

Yukarıdaki örneğimizi dikkatli inceleyelim. Class'ımızda sabit değişkenler tanımladık. Normal değişkenler de tanımlayabilirdik. Lakin suyu bulandırmamak ve sabit değişkenlere yoğunlaşmak için bu yolu seçtik. Herneyse, sabit bir değişken diğer sıradan değişkenler de olduğu gibi $this anahtar kelimesi ile çağrılamazlar. Eğer sabit bir değişken sınıf içerisinde kullanılacaksa; self anahtar kelimesi kullanılmalıdır. Yani $this yerine self anahtar kelimesini kullanmalıyız. $this anahtar kelimesinde -> işaretlerini kullanırken, self anahtar kelimesinde: :: işaretleri kullanılmaktadır. self anahtar kelimesinden bir sonraki bölümde detaylı olarak bahsedeceğiz. Çünkü başka kullanım alanları da var. Hepsinden detaylıca bahsedeceğiz.

Class dışında sabit değişkenleri çağırmak için class'ın ismi yazılır. Daha sonra :: işaretleri konur ve sabit değişkenin adı yazılır. Class içerisinde, self yerine yine bu yöntemi de kullanabilirsiniz. Aynı yöntem ile self'e gerek kalmadan hem class içerisinde hemde class dışarısında, class'ın adı kullanılarak sabit değişkenler çağrılabilir.

5.4 Shadowing

Shadowing, kelime anlamı olarak Gölgeleme manasına gelmektedir. Burada kadar olan bölümlerde hep, class içerisindeki değişkenlerin public yerine private veya protected olması gerektiğinden bahsettik ve bunlara nasıl atama yapılması gerektiğini konuştuk. Bu bölüm aslında biraz da ekstra bir bölüm. Sadece sizlere iyi bir programlama mantığı verilmesi amacıyla yazılmıştır. Bunu baştan belirteyim. Yani okumak zorunda değilsiniz. Hemen diğer bölüme geçebilirsiniz.

Class içerisinde bir değişken tanımlaması yaparken; şayet bu değişkeni inheritance (kalıtım)'de kullanmayacaksınız mutlaka private ile korumaya almanızı tavsiye ederim. Çünkü public olan üyeler her zaman sistemi tehdit etmeye açıktır. Şayet değişkeni kalıtım'da kullanacaksanız o zaman da protected ile korumaya alın derim.

Tekrar ana konumuza dönecek olursak; private veya protected olan üyelere dışarıdan gelen değer ile doldurabiliriz. Çok basit bir örnek ile shadowing nedir tanımaya çalışalım. Örneğimiz aşağıdaki gibi olacaktır.

```
<?php
class ExampleClass {

    private $name;
    private $surname;
```

```
    private $location;

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

    public function setSurname ($surname)
    {
        $this->surname = $surname;
    }

    public function setLocation ($location)
    {
        $this->location = $location;
    }

}
```

Yukarıdaki örneğimizde; tanımlamış olduğumuz private değişkenlere, public olan fonksiyonlar ile ilk değer atamalarını gerçekleştirdik. Fonksiyonlar da private olsaydı maazAllah class dış dünyaya kapanır, Kuzey Kore gibi olurdu. Herneyse, bu işin esprisi tabi ki.

Bu örneğimizde shadowing yaptık. Peki nasıl yaptık bu işlemi? Her değişken için bir fonksiyon yazdık. Biz kısaca bu fonksiyonlara: setter fonksiyonları diyoruz. Fonksiyon isimlendirmelerine dikkat ederseniz; her fonksiyonun başında set ibaresi var ve daha sonra anlaşılması kolay olsun diye atama yapılacak olan değişkenin ismi yazılı. Bu fonksiyonlar parametre olarak atama yapılacak olan değişkenin ismi aynı olan parametreler. İşte burada shadowing başlıyor. $this->name = $name ibaresi gelen parametre değerinin, class içerisinde tanımlı olan $name parametresine atanması gerektiğini söylüyor. Yani aynı isimli iki değişken var ve yeni gelen değer eski değeri gölgeliyor. İşte bu işleme biz kısaca shadowing diyoruz. Shadowing işlemi ile yeni değişken tanımlamamıza gerek kalmıyor. Yani her şey açık ve net.
Böyle durumlarda, yani atama yapılması gereken durumlar mutlaka parametre değerlerine initialize (ilk değer ataması) işlemini gerçekleştirin. Zira boş parametre gelirse sistemin işleyişine zarar verecektir. Yani demek istediğim şu şekilde:

```
public function setName ($name = '')
{
    $this->name = $name;
}

public function setSurname ($surname = '')
{
    $this->surname = $surname;
}

public function setLocation ($location = '')
{
    $this->location = $location;
}
```

setter fonksiyonları yazdık. private olan bu değerleri de alabilmek için getter fonksiyonları da yazmamız iyi bir programcılık örneği olacaktır. Bunu da aşağıdaki gibi gerçekleştirebiliriz.

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

public function getSurname ()
{
    return $this->surname;
}

public function getLocation ()
{
    return $this->location;
}
```

Yine aynı şekilde fonksiyon isimlendirmesinde ön ek olarak get ibaresini kullanıyoruz. Yazdığımız kodların tamamı da aşağıdaki gibi olacaktır.

```
<?php

class ExampleClass {

    private $name;
    private $surname;
    private $location;

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

    public function setSurname ($surname = '')
    {
        $this->surname = $surname;
    }

    public function setLocation ($location = '')
    {
        $this->location = $location;
    }

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

    public function getSurname ()
    {
        return $this->surname;
```

```
    }

    public function getLocation ()
    {
        return $this->location;
    }

}
```

Shadowing anlatmak için yola çıktık lakin setter ve getter fonksiyonlardan da bahsettik. Laf lafı açıyor dostlar. Çok konuştuysak affola!

BÖLÜM 6: Static

Static aşağı yukarı tüm OOP tabanlı olan programlama dillerinde bulunan bir anahtar kelimedir. Peki bu anahtar kelime ne işe yarar şimdi bundan bahsedelim.

static bir fonksiyona veya bir properties'e gelebilir. static olarak tanımlanan üyeler, o class'tan herhangi bir object oluşturmadan çağrılabilir ve kullanılabilir. Yani static bizi object oluşturma derdinden kurtarıyor. Şöyleki:

```
<?php

class StaticClassExample {

    static $name;

    static function sayHello ()
    {
        echo("Hello World!");
    }
}

StaticClassExample::$name = "emrecan-oztas;
StaticClassExample::writeName();
```

Örneğimizi dikkatle inceleyelim. static olarak; $name adında bir değişken tanımladık. Ayrıca bir de static olarak sayHello ( ) isminde bir de fonksiyon tanımladık. Bu class dışında, bu class'tan bir instance oluşturmadan doğrudan static olan değişkenleri ve fonksiyonları başarıyla çağırdık. Burada çağırma işlemi: ClassName :: ClassMember şeklinde gerçekleştirilir. Bir kez daha tekrar edersek; önce class ismi daha sonra :: işaretleri ve sonra da class'ın kullanmak istediğimiz üyesinin ismi.

Bu söyleyeceğim pek bilinmez. Lakin PHP dünyasında :: [ing: double-colon] işareti: Paamayim Nekudotayim olarak adlandırılır. Bunun nedeni ise: şuan PHP'i üzerinde söz sahibi olan ve PHP'i geliştirmeye devam eden şirket İsrail menşeli Zend şirketi ve Hebrew (İbranice) dilinde Paamayim Nekudotayim de double-colon anlamına gelmektedir.

static olarak tanımlanan üyeler, herhangi bir Access Modifiers (Erişim Belirleyiciler) ile ödüllendirilebilirler. Daha önce erişim belirleyiciler için 3p demiştik. İşte static olarak tanımlanan bu üyeler 3p'den herhangi birisini alabilirler. Şöyleki:

```
private static $name;

public static function sayHello ()
{
    echo("Hello World!");
}
```

Burada static size bir özellik daha sağlıyor. Erişim belirleyiciler ile static anahtar kelimesinin yerleri değiştirilebilir. Şöyleki:

```
static private $name;

static public function sayHello ()
{
    echo("Hello World!");
}
```

Lakin ilk kullanım daha yaygındır. Dolayısıyla; önce erişim belirleyici yazılım ardından static anahtar kelimesinini yazılması daha doğru olacaktır.

6.1 self

self anahtar kelimesinden bir önceki bölümde bahsetmiştik. self anahtar kelimesi, const yani sabit değişkenlerin çağrımında ve kullanımında da kullanılıyor idi. self anahtar kelimesi aynı zamanda; bir class içerisindeki static üyelerin çağrımında ve kullanımında da geçerlidir. self'i class içerisinde kullanabilirsiniz. Class dışında, self yerine o class'ın isminin kullanılması gerekir. Şöyleki:

```
class StaticClassExample {

    static $name = "emrecan-oztas";

    static function sayHello ()
    {
        echo self::$name;
    }
}
```

şeklinde kullanılması daha doğru olacaktır. Yani bir class içerisinde static bir üye varsa ve o class içerisinde o static üyeler ile işlem yapılmak isteniyorsa; self anahtar kelimesini kullanmanız yerinde bir karar olacaktır. Aynı zamanda iyi bir programcılık örneği sayılacaktır. Buna dikkat edelim.

BÖLÜM 7: Constructors ve Destructors

Class ve class içerisindeki üyelerin bazen ilk değer atamalarının yapılması gerekir. Ya da siz öyle istersiniz. Veyahut başlangıç için elinizde belli değerlerin olmasını isteyebilirsiniz.
Class ve class üyeleri bellekte belli bir yer kaplarlar. Bu kapladıkları alanlar: heap ve stack olmak üzere değişir. Bazen bazı durumlarda, bu durumlar genellikle belleğin yetersiz olduğu zamanlar için geçerlidir, yaptığınız işlemlerden sonra belleği temizlemek ve yeni değerler için yer açmak isteyebilirsiniz.

Aslında kısaca Constructors ve Destructors özetledik. Lakin daha konuşulacak çok şey var. O yüzden dinlemeye hazır olun, benden söylemesi.

7.1 Constructors

Construct kelime anlamı olarak; İnşaa etmek, yapmak manalarına gelmektedir. Dolayısıyla, Constructor için yapıcı dememiz yerinde bir karar olacaktır. Constructor'lar aslında birer fonksiyondur. Bu fonksiyonlar, class'ın öz ve öz fonksiyonudur. Yani tamamen class için kullanılan yapılardır. Peki bu fonksiyonlar ne iş yaparlar? Constructor, class'ı inşaa eden, ayağa kaldıran yapılardır. Örneğin bir class'tan bir object oluşturduğunuz zaman ilk çalışan fonksiyon constructor fonksiyonudur. PHP çekirdeği, object oluşturulan class'ın fonksiyonuna bakar ve o fonksiyonda ne tanımlıysa yerine getirmeye çalışır.

Constructor'ı çalıştıran, class'tan bir instance oluşturulmasıdır. Class'tan bir object oluşturulduğu zaman çalışırlar. Bir constructor aşağıdaki şekildeki gibi tanımlanır.

```
function __construct()
{

}
```

Yukarıdaki yapıda da görüldüğü gibi bir constructor, __construct () anahtar kelimesi ile tanımlanır. Örneğin: Java'da bir constructor, class'ın adıyla aynı isime sahip olması gerekir. Lakin PHP'de durum öyle değildir. Çift alt çizgi (__) ve bitişiğine construct( ) anahtar kelimesinin yazılması gerekir.

Bir constructor, erişim belirleyici olarak sadece public anahtar kelimesini alabilir. private ve protected constructor olmaz. Veyahut herhangi bir erişim belirleyici almadan tanımlanırlar. Seçim size kalmış yani.

Bir diğer konu da bir class'ın yalnızca bir tane constructor'ı olur. Birden fazla constructor tanımlamaya kalkarsanız aşağıdaki hata ile karşırsınız.

```
Fatal error: Cannot redeclare AClass::__construct()
```

Bir class'tan bir object tanımlandığı anda constructor çalışmaya başlar dedik. Şimdi bu dediğimizi ispat edelim. Örneğimiz aşağıdaki gibi olacaktır.

```php
<?php

class ConstructorExample {

    public function __construct ()
    {
        echo("Hello World!");
    }

}

$consObj = new ConstructorExample();
```

Yukarıdaki örneğimizi tarayıcıda görüntülediğimizde aşağıdaki ekran alıntısıyla karşılaşırız.

```
Hello World!
```

Yukarıdaki örneğimizde de görüldüğü gibi, contructor tanımlı bir class'tan bir object oluşturulduğu zaman hemen contructor fonksiyonu çalışır.

Contructor şöyledir, böyledir diyoruz ama biz contructor nerede kullanabiliriz? Güzel soru. Contructor'u örneğin: class içerisindeki properties (class değişkenleri)'lerin initialize (ilk değer ataması) edilmesinde kullanabiliriz. Şöyleki:

```php
<?php

class ConstructorExample {

    private $name;
    private $surname;

    public function __construct ()
    {
        $this->name = "Emre Can";
        $this->surname = "ÖZTA$";
    }

    public function writer()
    {
        echo $this->name . " " . $this->surname;
    }

}

$consObj = new ConstructorExample();
$consObj->writer();
```

Yukarıdaki örneğimizin ekran çıktısı da aşağıdaki gibi olacaktır.

```
Emre Can ÖZTAŞ
```

Görüldüğü gibi constructor ile değişkenlerimize ilk değer ataması işlemini gerçekleştirdik.
Constructor fonksiyonu parametre olarak değerler de alabilir. Aşağıdaki örneğimizi inceleyelim.

```
public function __construct ($name, $surname)
{
    $this->name = $name;
    $this->surname = $surname;
}
```

Yukarıdaki örneğimizde: değişkenlerin ilk değer atamalarını gerçekleştirebilmek için contructor'da parametreler tanımladık ve bu değerleri class değişkenlerimize atadık. Peki buradaki en önemli soru şu: constructor'lara parametre olarak geçececiğimiz değerleri nerde sağlamalıyız? Güzel soru. Constructor'a parametre olarak değişken ataması yapmak için bunu class'tan object tanımlarken belirtmeliyiz. Yani:

```
$consObj = new ConstructorExample("Emre Can", "ÖZTAŞ");
```

şeklinde olmalıdır. Burada size şöyle bir şey göstermek istiyorum. Daha önce bahsetmiştik. Bir class'tan object tanımlarken iki farklı şekilde tanımlayabiliriz. Bu tanımlama şekilleri aşağıdaki gibidir.

```
$consObj = new ConstructorExample;
// veya
$consObj = new ConstructorExample();
```

Şayet contructor tanımlamışsanız ve bu contructor bir veya birden fazla parametre alıyor ise ikinci şekildeki gibi bir tanımlama yapmalıyız ve bu parametreleri parantezler içerisinde belirtmeliyiz. Lakin contstructor tanımlamamışsanız veyahut tanımladığınız contstructor herhangi bir parametre almıyor ise birinci kullanımdaki gibi kullanabilirsiniz. Benim size tavsiyem her zaman ikinci kullanımı seçmeniz.

Son olarak; constructor'lar ile örneğin bir class içerisinde gerekli olduğu durumda, session (oturum), database (veri tabanı) ve diğer herhangi bir işlemleri de yerine getirebilirsiniz. Constructor kullanımı size oldukça kolaylıklar sağlayacaktır. Tabiki doğru kullanılırsa.

7.2 Destructors

Destructors için Contstructor tam karşıtı diyebiliriz. Destructor, kelime anlamı olarak Tahrip etme, yoketme manalarına gelmektedir. Biz de destructor için yıkıcı diyelim. Şimdi hatırlayalım. Constructor fonksiyonlar, bir class'ı ayağa kaldıran yapılardı. Örneğin ilk değer atamalarını yapabilirler, çeşitli fonksiyonları çağırabilirler, sistemle iletişime geçebilirler ve aklınıza gelebilecek her türlü işlemi; class'tan bir object tanımlamasıyla yerine getirebilirler. Destructors ise tam tersi bir görevdedirler. Constructor ile dolan bellek alanlarını boşaltırlar. Yani oluşturulan ne varsa hepsini yokederler.

Constructor'lar için ne konuşmuş isek hepsinin arkasındayız ve anlattıklarımızın hepsi destructor'lar içinde geçirli olduğunu beyan ederiz. İlginç bir cümle kurmuşum. Sanırım siyaseti denemeliyim! Tabi ki bu işin esprisi.

Bir destructor aşağıdaki şekildeki gibi tanımlanır.

```
public function __destruct ()
{

}
```

Görüldüğü gibi: __destruct () anahtar kelimesi ile bir destructor tanımlamasını gerçekleştirdik. Tekrar bahsedelim. Çift alt çizgi (__) ve destruct () anahtar kelimesinin yazılmasıyla bir destructor oluşturabiliriz. Farkettiğiniz gibi destructor bir fonksiyon. Tıpkı contructor gibi.

Destructor'ların bir fonksiyon olduğundan bahsettik. Peki bu fonksiyon ile neler yapabiliriz? Destructor, class'ın son raddesidir. Yani çıkış kapısı. İşlemlerimizi tamamlamak için bir fırsattır. Örneğin büyük bir yazılım platformunda görevlisiniz. Çeşitli işlemler yaptırdınız, kullanıcıya. Kullanıcı tam sistemden çıkacakken bu kullanıcının yaptığı işlemlerin yedeğini alabilirsiniz.

Akılda kalıcı olması için basit bir örnek yapalım. Örneğimiz aşağıdaki gibi olacaktır.

```
class DestructorExample {

    private $username;

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

    public function __destruct ()
    {
        $this->sessionDestroy();
        echo("Session is destroyed!");
    }

    public function sessionStart()
    {
        session_start();
        echo("Session is started!<br>");
        $_SESSION['username'] = $this->username;
        echo("Welcome! " . $_SESSION['username'] . "<br>");
    }

    public function sessionDestroy()
    {
        session_destroy();
    }
```

```
}
```

Her şey tamam şimdi class'ımızdan bir object tanımlayalım ve daha sonra bu object'i yok edelim.

```
$consObj = new DestructorExample("jsawyer");
unset($consObj);
/**
* veyahut:
* $consObj = NULL
* olarak yazilabilir.
*/
```

Şimdi biraz konuşalım. Class'ımızı ve bu class'a ait olan üyeleri yazdık. Daha sonra bu class'tan bir instance oluşturduk. Ve object oluştururken bir de parametre gönderdik. Constructor bu verdiğimiz bilgiler ışığında işlemlerini yerine getirecektir. Oluşturmuş olduğumuz bu object'i daha sonra unset ( ) fonksiyonu ile yokettik. Burada bu object'i yok etmemizin sebebi destructor fonksiyonunu çalıştırmak. Şimdi kodlarımızın ekran çıktılarına bir bakalım.

```
Session is started!
Welcome! Jsawyer
Session is destroyed!
```

Görüldüğü gibi herşey bizim planladığımız şekilde gitti.

Size çok önemli bir şeyden bahsetmek istiyorum. PHP yapısı gereği, yapılan her işlemden sonra otomatik olarak destructor fonksiyonu çalıştırmaktadır. Yani bizim object'imizi yok etmemize gerek yok. PHP zaten otomatik olarak bu işlemi yapmaktadır. Dolayısıyla destructor fonksiyonunu kullanmamıza gerek yok. Size de tavsiyem destructor fonksiyonunu kullanmayın. Kullanın ama ne yaptığınızı bilerek kullanın. Örneğimizdeki gibi bir kullanımda: constructor'da session açılacak hemen ardından destructor çalışacak ve session sonlanacaktır. Yani yanlış bir kullanım oldu. Tabiki biz bu kullanımı örnek olması açısından gösterdik.

BÖLÜM 8: Inheritance

Inheritance kelime anlamı olarak Miras, Kalıtım manalarına gelmektedir. OOP'de de aynen bu görevdedir. Bir class, başka bir class tarafından miras alınabilir. Miras alınan class'taki private olmayan üyeler alt class'a geçer.

Şöyle bir örnek verelim. Mehmet amca'nın 3 tane çocuğu vardır. Amcamızın sahip olduğu mallar, bu 3 çocuğun da malları olarak kabul edilir. Hukuken de böyledir. Babanın sahip olduğu mallar üzerinde, çocuklarının da hakları vardır. Zaten kişi öldükten sonra bu mallar kendinden sonra kalanlara eşit şekilde pay edilir. Herneyse, Mehmet amcanın sahip olduğu mallara çocukları da erişebilir ve kullanabilir. İşte OOP dünyasında da böyledir. Tepedeki class, alt class'lara miras olarak devredilebilir. Devirolan bu class'ın üyeleri alt class'larda da kullanılabilir. Çünkü artık o üyelere ya da kısaca mal olarak nitelendirebilirsiniz, alt class'larda sahiptir.

Peki bu bize nasıl bir özellik kazandırır? Güzel soru. Kalıtım sayesinde aynı fonksiyonların ve değişkenlerin tekrar tanımlanmasına gerek kalmaz. Gerekli olduğu yerde devralınan üyeler diğer class'ların kullanımına açıktır.

Örneğin elimizde şöyle bir yapı olduğunu farzedelim.

```
class Person {

    protected $name;
    protected $surname;
    protected $location;

    public function __construct()
    {
        echo("Person Class");
    }

    // setter function
    public function setName($name)
    {
        $this->name = $name;
    }

    public function setSurname($surname)
    {
        $this->surname = $surname;
    }

    public function setLocation($location)
    {
        $this->location = $location;
    }

    // getter function
    public function getName()
    {
        return $this->name;
```

```
    }

    public function getSurname()
    {
        return $this->surname;
    }

    public function getLocation()
    {
        return $this->location;
    }

}
```

Yukarıdaki class'ımızın properties'leri protected olarak tanımlı. Bu properties'leri doldurmak ve değerlerini almak için getter | setter fonksiyonlar tanımlı. Hatırlarsanız; protected ile oluşturulmuş olan üyeler kalıtım sırasında alt sınıflara geçebiliyordu. public olarak tanımlanmış olan üyeler de geçebilir lakin private olarak tanımlanmış olan üyeler hiç bir halukarda alt sınıflara geçemezler.

Şimdi yazdığımız class hakkında konuşalım. Person mahlasıyla tanımladığımız class'ımızdaki özellikler aslında ortak ve yeniden kullanılması gereken durumlar. Her class için bu özellikleri yeniden tanımlamak oldukça külfetli olur. Külfetten kastım hem zaman hem performans hemde disk alanı kullanımı bakımındandır. O yüzden gerekli olduğu durumlarda bu class'ı miras alarak bu külfetten kurtulabiliriz. Şimdi inheritance konusunu daha iyi anladığınızı sanıyorum.

Person class'ı tepede olan bir class ve içerisinde genel özellikleri barındırıyor. Peki bunu nereden anlıyorum? Mantıken. Çünkü bir insan herşey olabilir. Örneğin: doktor, mühendis, işçi, aşçı, dişçi... Ama temelde hepimiz insanız. Ve insanın da belli ortak özellikleri var. İşte biz bu ortak özellikleri bir yerde topladık ve adına Person class'ı ismini verdik. Şimdi ise asıl konuya geliyoruz. Hepimiz temelde insanız dedik. İşte biz insanlarda aramızda belli sıfatlara göre ayrılıyoruz. Biz şimdilik iş konusu hakkında konuşacağız.

İnsan sınıfının bir alt sınıfı olarak employee yani işçi sınıfını belirleyebiliriz. O zaman bir class yazalım ve bir işçinin özellikleriyle class'ımızı dolduralım. Class'ımız aşağıdaki gibi olacaktır.

```
class Employee {

    private $age;
    private $length;
    private $eyesColor;

    public function __construct()
    {
        echo("Employee class");
    }

    // setter function
```

```
    public function setAge($age)
    {
        $this->age = $age;
    }

    public function setLength($length)
    {
        $this->length = $length;
    }

    public function setEyesColor($eyesColor)
    {
        $this->eyesColor = $eyesColor;
    }

    // getter function
    public function getAge()
    {
        return $this->age;
    }

    public function getLength()
    {
        return $this->length;
    }

    public function getEyesColor()
    {
        return $this->eyesColor;
    }

}
```

Employee class'ımızı yazdık. Employee sınıfında yeni üyeler tanımladık. Bu üyeler doğrudan bu sınıfta kullanılacak. Person class'ı içerisinde tanımladığımız değerler aslında Employee class'ı içinde gerekiyor. Person class'ı içerisinde ne varsa hepsini Employee class'ı içerisinde tanımlamamıza gerek var mı? Tabiki de yok. Çünkü; Employee class'ı Person class'ının bir alt class'ı. Yani Person'da olan özellikler Employee içinde geçerli.

Employee class'ın da tanımladığımız properties'ler private. Çünkü bu class'ı bir başka kalıtımda kullanmayacağım. O yüzden dışarıdan erişime kapattım. Peki Employee class'ını kalıtımda kullanabilir miydim? Elbette istediğim class'lar için kullanabilirdim.

Herneyse sanırım çok konuştuk. Bir sonraki alt başlıkta bu işi tatlıya bağlayalım.

8.1 extends

Önce Person class'ını yazdık ve daha sonra da Employee class'ını yazdık. Employee class'ımız Person class'ımızın bir alt class'ı. Şimdi bunu belirtmemiz lazım. Belirtme işlemi içinde tabi ki bir anahtar kelime lazım. Bu anahtar kelime de extends kelimesi.

extends kelimesi: genişletmek ve açmak gibi manalara geliyor. Alt class'ı üst class ile genişleteceğiz.

Tekrar bahsedelim. Üst class'ımız (baba class olarak nitelendirebilirsiniz) ve alt class'ımız (çocuk class olarak nitelendirebilirsiniz). Şimdi extends anahtar kelimesini kullanalım.

```
class Employee extends Person{

}
```

Yukarıda miras alma işlemini gerçekleştirdik. Kodlarımızın tamamız aşağıdaki gibi olacaktır.

```
<?php

class Person {

    protected $name;
    protected $surname;
    protected $location;

    public function __construct()
    {
        echo("Person Class");
    }

    // setter function
    public function setName($name)
    {
        $this->name = $name;
    }

    public function setSurname($surname)
    {
        $this->surname = $surname;
    }

    public function setLocation($location)
    {
        $this->location = $location;
    }

    // getter function
    public function getName()
    {
        return $this->name;
    }

    public function getSurname()
    {
        return $this->surname;
    }
```

```
    public function getLocation()
    {
        return $this->location;
    }

}

class Employee extends Person{

    private $age;
    private $length;
    private $eyesColor;

    public function __construct()
    {
        echo("Employee class");
    }

    // setter function
    public function setAge($age)
    {
        $this->age = $age;
    }

    public function setLength($length)
    {
        $this->length = $length;
    }

    public function setEyesColor($eyesColor)
    {
        $this->eyesColor = $eyesColor;
    }

    // getter function
    public function getAge()
    {
        return $this->age;
    }

    public function getLength()
    {
        return $this->length;
    }

    public function getEyesColor()
    {
        return $this->eyesColor;
    }

}
```

Kodlarımızdan da anlaşıldığı üzere; Employee class'ı Person class'ını miras aldı. Artık Person class'ından bir object oluşturabiliriz.

```
$employee = new Employee();
```

Şimdi sırasıyla bu değerlerin atamasını yapalım ve ekrana yazdıralım.

```
// set
$employee->setName("James Ford");
$employee->setSurname("SAWYER");
$employee->setLocation("USA / Lost Island");
$employee->setAge("40");
$employee->setLength(1.85);
$employee->setEyesColor("Green");
// get
echo("<br>");
echo $employee->getName();
echo("<br>");
echo $employee->getSurname();
echo("<br>");
echo $employee->getLocation();
echo("<br>");
echo $employee->getAge();
echo("<br>");
echo $employee->getLength();
echo("<br>");
echo $employee->getEyesColor();
```

Kodlarımız tamam. Şimdi ekran çıktısına bir bakalım.

```
Employee class
James Ford
SAWYER
USA / Lost Island
40
1.85
Green
```

Görüldüğü gibi kalıtım işlemini gerçekleştirdik.

Ekran çıktısına dikkat ederseniz; ilk sırada yazdığımız, Employee class'ına ait olan Constructor çalıştı. Dilersek; Person class'ının constructor'unu da Employee class'ında çalıştırabiliriz. Lakin bunu bir sonraki başlık altında detaylıca inceleyeceğiz.

Şimdi inheritance konusundan konuşmaya devam edelim. Yazmış olduğumuz Person class'ına ait olan üyeler bir alt class olan Employee class'ına da geçti. Yani Person class'ının üyeleri artık Employee class'ının da üyesi oldu. Person class'ı herhangi bir class'ı devralmadığı için fonksiyonları sabit. Yani kendi içinde tanımlı olan fonksiyonlardan başka fonksiyonlara sahip değil. Örneğin; Person sınıfından bir nesne oluşturualım ve fonksiyonlarına bakalım.

Yukarıdaki ekran alıntısında da görüldüğü üzere; Person class'ı herhangi bir class'ı devralmadığı için ekstra yeni üyelere sahip değil. Properties'leri de protected olduğu için dışarıdan kullanılamıyor.

Size altın niteliğinde bir kural. Bu kuralı unutmayın! Bir class yalnız ve yalnızca bir class'ı miras alabilir. Sadece bir class'ı üst class olarak seçebilirsiniz. Peki bu neden böyle? OOP için gerçek hayatın programlama dünyasına uyarlamasıdır dedik. O halde sorarım size: iki tane biyolojik babası olan var mı dünyada? Yok! İşte kalıtım'da da böyle. Her class'ın bir üst class'ı olabilir.

Her class'ın bir üst class'ı olabilir dedik. Peki birden fazla class'ta olan ortak üyeleri kullanmak istersek? O zaman bir kalıtım hiyerarşisi oluşturmanız gerekir. Örneğin; AClass, BClass ve CClass'ın hepsi DClass'ında birleştirilmek istenirse; waterfall (şelale) modeline göre bir yapı kurmanız gerekir. Yani:

```
class AClass { }
class BClass extends AClass { }
class CClass extends BClass { }
class DClass extends CClass { }
```

gibi bir yapı kurmalısınız. Bu yapımızda; DClass'ı AClass, BClass ve CClass'ın tüm özelliklerine sahip. Burada dikkat etmeniz gereken konu; kalıtım hiyerarşisi kurmak istediğiniz class'ların birbirleriyle aynı prensiplere veyahut amaçlara sahip olması gerekir. Yani yapacağınız dizaynı bozacak herhangi bir durum olmamalıdır.

Son olarak inheritance konusunda bir kaç şey söylemek istiyorum. inheritance çok tehlikeli bir kavramdır. Kuracağınız hiyerarşi çok önemlidir. O yüzden çok iyi tasarlanmalıdır. Her türlü durum hesaba katılmalıdır. En önemli durumda; yazdığınız kodların yeniden yapılandırılması durumudur. Yaptığınız tasarım daha sonraki düzeltme ve bakımları kolaylaştırmalıdır. Lakin kalıtım bu işi oldukça zorlaştırmaktadır. Açıkcası bir geliştirici olarak; inheritance konusunu kullanmadan önce bir kez daha düşünün derim.

8.2 parent

parent anahtar kelimesi, inheritance yapısında üst sınıfın üyelerine alt sınıflarda erişmek için kullanılan bir yapıdır. Öyle ya üst sınıfın üyeleri (private dışında) alt sınıfın üyeleri oldular.
8.1 extends alt başlığı altında, dilersek üst sınıfın constructor'unu çağırabileceğimizi söylemiştik. Ve bu konudan daha sonra bahsedeceğimizi söylemiştik. İşte o gün bugündür. Şimdi bu konudan detaylı olarak bahsedeceğiz.

Kalıtım hiyerarşisinde, alt class üst class'ın üyelerini kullanabilir. Çünkü artık o üyelere sahip oldu. İşte bir alt class, bir üst class'ın üyelerini çağırmak için parent anahtar kelimesini kullanır. Peki anahtar kelimesini nasıl kullanabiliriz? Şimdi bu konu hakkında konuşalım.

Basit bir kalıtım hiyerarşisi oluşturalım. Yapımız aşağıdaki gibi olsun.

```
<?php

class AClass
{
```

```
    public $publicVar;
    protected $protectedVar;
    private $privateVar;

    public function __construct()
    {
        echo("AClass Constructor<br>");
    }
}

class BClass extends AClass
{
    public function __construct()
    {
        parent::__construct();
        echo("BClass Constructor<br>");
    }

    public function setPublicVar($publicVar)
    {
        $this->publicVar = $publicVar;
    }

    public function setProtectedVar($protectedVar)
    {
        $this->protectedVar = $protectedVar;
    }

    public function getPublicVar()
    {
        return $this->publicVar;
    }

    public function getProtectedVar()
    {
        return $this->protectedVar;
    }

}

class CClass extends BClass
{

    function __construct()
    {
        parent::__construct();
        echo("CClass Constructor<br>");
    }

}

$cObject = new CClass();
$cObject->setPublicVar("emrecan-oztas");
$cObject->setProtectedVar(639);
```

```
echo $cObject->getPublicVar();
echo("<br>");
echo $cObject->getProtectedVar();
```

Peki yukarıdaki örneğimizde ne yaptık? Örneğimizde; AClass class'ında bir takım properties'ler tanımladık. Daha sonra AClass class'ını BClass'ına miras olarak verdik. AClass içerisinde tanımlı olan properties'leri BClass'ın da $this anahtar kelimesi ile çağırdık ve kullandık. Tabiki burda; AClass'ında yer alan properties'ler const ve static olabilirdi. O zamanda self anahtar kelimesini kullanmamız gerekirdi. Yani normal OOP class'larında yaptığımız işlemlerin ve kullandığımız anahtar kelimelerin hepsi inheritance kavramı içinde geçerli. Ekstra yaptığımız bir şey yok. Tabiki parent anahtar kelimesi dışında.

Örneğimizde de görüldüğü üzere; üst class'ın fonksiyonlarını ve contructor'larını çağırmak için parent anahtar kelimesini kullandık. Evet bu her zaman böyledir. Üst class'ın fonksiyonlarına çağırmak veya erişmek için parent anahtar kelimesi kullanılır. Ek bilgi olması için tekrar bahsedelim. Üst class'ın üyeleri, static veya const olduğu durumda: self, diğer durumlarda $this anahtar kelimesini kullanılır.

Sanırım çok konuştuk. Yazdığımız örneğin ekran çıktısına bakalım ve bu başlığı tamamlayıp diğer başlığa geçelim. Ekran çıktımız aşağıdaki gibi olacaktır.

```
AClass Constructor
BClass Constructor
CClass Constructor
emrecan-oztas
639
```

8.3 Override

Override kelime anlaşı olarak; ezmek anlamına gelmektedir. Buraya kadar olan bölümde yaptığımız örneklerimizde; hep üst class'ın fonksiyonlarını ve değişkenlerini alıp olduğu gibi kullandık. Peki bu üst class'ın üyelerini alt sınıflarda değişikliğe uğratabilir miyiz? Elbette değiştirebiliriz. Bu işleme kısaca override yani ezme işlemi diyoruz.

Peki ezme işlemini nasıl yapacağız. Hemen bir class hiyerarşisi kuralım ve örneğimiz üzerinden bu işlemi anlatmaya çalışalım. Örneğimiz aşağıdaki gibi olacaktır.

```
<?php

class SuperClass
{

    public $name = "Emre Can Öztaş";

    public function nameUpper()
    {
        $this->name = strtoupper($this->name);
```

```
    }

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

    public function nameWriter()
    {
        echo($this->name . "<br>");
    }

}

class SubClass extends SuperClass
{

    /**
    * @override
    */
    public function nameUpper()
    {
        $this->name = strtolower($this->name);
    }

    /**
    * @override
    */
    public function nameLower()
    {
        $this->name = strtoupper($this->name);
    }

    /**
    * @override
    */
    public function nameWriter()
    {
        echo("Name: " . $this->name . "<br>");
    }
}

$superObject = new SuperClass();
$subObject   = new SubClass();

$superObject->nameLower();
$superObject->nameWriter();
$superObject->nameUpper();
$superObject->nameWriter();

$subObject->nameLower();
$subObject->nameWriter();
$subObject->nameUpper();
$subObject->nameWriter();
```

Örneğimizi yazdık. İlk olarak; SuperClass adında bir class açtık ki bu class bizim üst class'ımız olacak. Daha sonra da SubClass adında bir class açtık. Bu class'ımızda SuperClass class'ını miras olarak alacak. SuperClass içerisinde tanımlı olan bir değişken var ($name) ve çeşitli fonksiyonlar var. Örneğin bu fonksiyonlardan; nameLower() tanımladığımız değişkeni küçük harflere çevirecek, nameUpper() fonksiyonu değişkeni büyük harflere çevirecek ve son olarak nameWriter() fonksiyonu da değişkeni ekrana yazdıracak. SuperClass'ını miras olarak alan SubClass, aldığı fonksiyonları override etti. Yani üst class'tan gelen fonksiyonları aldı ve kendine göre çevirdi. SuperClass fonksiyonları içerisinde yaptıklarımızın tam tersini yaptı. Burada istersek; üst class'ın public ve protected olan değişkenlerini de değiştirebiliriz.

SubClass içerisinde fonksiyonların üzerlerine yazdığımız açıklama satırları ve @override annotation (not veya dipnot olarak nitelendirebiliriz)'ı kendimiz yazdık. Yani böyle bir kural yok. Sadece override işlemini gerçekleştirdiğimizi göstermek için oluşturduk.

Sanırım ezme işlemini anladık. Şimdi ekran çıktısına bir bakalım. Ekran çıktımız aşağıdaki gibi olacaktır.

```
emrecan öztaş
EMRE CAN ÖZTAŞ
Name: EMRE CAN ÖZTAŞ
Name: emre can öztaş
```

Yukarıdaki ekran alıntısına hata olduğunu sanabilirsiniz lakin Türkçe karakter problemi olduğu için bazı Türkçe karakterler yanlış yazıldı, PHP tarafında.

8.4 final

final anahtar kelimesi; class, properties ve fonksiyonlara gelelir. Örneğin:

```
final class SupperClass {}
final $variable;
final function doPost(){}
```

Peki final kelimesi ne yapar ya da kullanım amacı nedir? final anahtar kelimesi, kullanıldığı üyelerin kalıtım yoluyla aktarılmasını engeller. Yani bir üyenin veya class'ın miras alınmasına izin vermez. Tabiki bu da bize class'ların ve üyelerin güvenliğini sağlamamıza izin verir.

Öncelikle bu 3 durum içinde örnekler verelim.

Class için:

```
final class SuperClass {

}

class SubClass extends SuperClass
```

```
{

}
```

Yukarıdaki örneğimiz için ekran çıktısını alalım.

```
Fatal error: Class SubClass may not inherit from final class
(SuperClass) in /opt/lampp/htdocs/untitled.php on line 9
```

Ekran alıntısında da görüldüğü gibi; SuperClass, final olduğu için kalıtım yapılmasına izin verilmedi.

Properties için:

```
class SuperClass {

    public final $variable;

}

class SubClass extends SuperClass
{

}
```

Yukarıdaki örneğimiz için ekran çıktısını alalım.

```
Fatal error: Cannot declare property SuperClass::$variable final, the
final modifier is allowed only for methods and classes in
/opt/lampp/htdocs/untitled.php on line 4
```

Yukarıdaki ekran alıntısında da görüldüğü gibi; $variable adlı değişken final olduğu için bir alt class'a geçmesine izin verilmedi.

Fonksiyon için:

```
class SuperClass {

    final function writer()
    {
        echo("Hello World!");
    }

}

class SubClass extends SuperClass
{

}
```

```
$subObject = new SubClass();
$subObject->writer();
```

Yukarıdaki örneğimizin ekran çıktısını alalım.

```
Hello World
```

Ekran alıntısında da görüldüğü gibi final olan fonksiyon miras olarak alın... Durun bir dakika o da ne? final olan bir fonksiyon miras olarak alınmış. Bu olamaz! Panik yok. Tabiki şaka yapıyorum. final olarak tanımlanmış olan bir fonksiyon miras olarak alınabilir. Burada herhangi bir sıkıntı yok. Lakin alınan bu fonksiyon override yani ezme işlemine tabi tutulamaz. Yani fonksiyonu başka bir class içerisinde yeniden yapılandırmamıza izin verilmez. Daha açıklayıcı olması için aşağıdaki örneğimizi inceleyelim.

```
class SuperClass {

    public final function writer()
    {
        echo("Hello World!");
    }

}

class SubClass extends SuperClass
{

    public final function writer()
    {
        echo("Moikka World!");
    }

}

$subObject = new SubClass();
$subObject->writer();
```

Yukarıdaki örneğimizin ekran çıktısı da aşağıdaki gibi olacaktır.

```
Fatal error: Cannot override final method SuperClass::writer() in
/opt/lampp/htdocs/untitled.php on line 19
```

Yukarıdaki ekran alıntısında da görüldüğü gibi final olan üst class'ın fonksiyonu alt class'larda override edilemez.

BÖLÜM 9: abstract

abtract kelime anlamı olarak; soyut veya soyutlama gibi manalara gelmektedir. OOP dünyasında bir class abstract olabileceği gibi bir fonksiyonda abstract olabilir. Lakin değişkenler için böyle bir durum söz konusu değildir.

Peki abstract bize ne gibi yararlar sağlar. abstract aslında özel bir kavram. Normal bir class yapısını ve aynı zamanda ilerde göreceğimiz interface yapısını bünyesinde birleştirir. Kullanımını yazacağımız örneklerimiz ile daha iyi anlayacağınıza eminim.

Bir abstract class aşağıdaki şekildeki gibi tanımlanır.

```
abstract class ClassName {

}
```

Görüldüğü gibi class'ımız öntakı olarak; abstract anahtar kelimesini aldı. Bu her zaman böyledir. Böylelikle class'a: sen bir abstract class'sın! mesajı veriyoruz.
abstract class'lardan object oluşturulamaz. Sadece extends ile kullanılabilirler. Yani bildiğimiz inheritance kavramında bir yapıtaşıdır.

Bir abstract class içerisinde yazacağımız fonksiyonlar ikiye ayrılır. Bunlar: normal fonksiyonlar ve gövdesi olmayan fonksiyonlar. Gövdesi olmayan fonksiyonlar, extends edilen class'ta gövdeye kavuşması lazım. Ayrıca gövdesi olmayan bir fonksiyon abstract anahtar kelimesiyle tanımlanmalı. Yani anlatmak istediğimiz, şu şekilde:

```
abstract class ClassName {

    abstract public function publicFunc();
    abstract protected function protectedFunc($name, $surname,
$location);

    public function sayHello ($username){
        echo("Hello! " . $username);
    }

}
```

Yukarıdaki yapımızı incelediğimizde; abstract anahtar kelimesi ile tanımladığımız fonksiyonların gövdesi yok. Çünkü bu fonksiyonlar gövdeye; extends edildikleri class veya class'larda kavuşacaklar. Öte yandan abstract class'lar içerisinde gövdesi olan fonksiyonlar da tanımlanır. Hatta ve hatta değişkenler de tanımlayabilirsiniz. Burada bir şeye dikkatinizi çekmek istiyorum. abstract olan fonksiyonlar kesinlikle ve kesinlikle private olarak tanımlanamazlar. Peki neden? Daha önce demiştik. Gövdesi olmayan metotlar extends edilen class'larda gövdeye kavuşacaktır. Dolayısıyla private değişken veya fonksiyonlar, kalıtım sırasında alt class'lara geçemezler. Dolayısıyla bu fonksiyonlar gövdesiz kalacaktır. Böyle bir duruma da abstract class izin vermeyecektir.

Bir class içerisinde, abstract anahtar kelimesi ile tanımlı olan bir fonksiyon var ise bu class'ta abstract olmalıdır. Zira abstract anahtar kelimesi ile tanımlı olan bir fonksiyonu yalnızca abstract olan bir class içerebilir. Tıpkı atasözünde olduğu gibi: Damdan düşenin halinden yine damdan düşen anlar!

abstract class ve fonksiyonlardan yeteri kadar bahsettikten sonra basit bir örnek yapalım. İlk olarak abstract class'ımızı yazalım. Class'ımız aşağıdaki gibi olacaktır.

```
abstract class CalClass {

    protected $result;

    abstract public function topla($number1, $number2);
    abstract public function cikar($number1, $number2);
    abstract public function carp($number1, $number2);
    abstract public function bol($number1, $number2);

    public function showResult (){
        echo("Result: " . $this->result);
    }

}
```

abstract class'ımız hazır. Sizin de gördüğünüz üzere bu class'ımız; 4 işlemi yerine getiriyor. 4 işlemi yerine getirecek olan fonksiyonlarımız abstract olarak tanımlanmış durumda. Sonucu da gösterecek olan fonksiyon, normal-sıradan bir fonksiyon.

Bu yazmış olduğumuz abstract class'ımızı, yazacağımız başka bir class'a extends edip fonksiyonları gövdelerine kavuşturacağız, inşAllah.

Diğer yazacağımız class'ımıza da aşağıdaki gibi olacaktır.

```
class UserClass extends CalClass {

    function __construct() {}

    public function topla($number1, $number2){
        $this->result = $number1 + $number2;
    }

    public function cikar($number1, $number2){
        $this->result = $number1 - $number2;
    }

    public function carp($number1, $number2){
        $this->result = $number1 * $number2;
    }

    public function bol($number1, $number2){
        $this->result = $number1 / $number2;
    }
```

```
}
```

UserClass isimli bir class tanımladık. Bu class'ımıza, abstract olan CalClass'ımızı extends ettik. Burada dikkat ederseniz: abstract içerisinde abstract olarak tanımladığımız fonksiyonlara gövdelerini verdik. Fonksiyonlarımızı aynen tanımlıyoruz. Tabi ki abstract anahtar kelimesini çıkararak. abstract class'ta bir fonksiyon protected tanımlanmışsa, alt class'ta bu fonksiyonu public olarak tanımlama şansımız yok. abstract class'ta neyse aynen bu şekle bağlı kalıyoruz.

Ekran alıntısında da görüldüğü üzere bize uyarı veriyor. Uyarı da: "4 tane abstract metot var bunları implement (uygula) et!" şeklinde.

Örneğimizin tam hali aşağıdaki gibi olacaktır.

```
<?php

abstract class CalClass {

    protected $result;

    abstract public function topla($number1, $number2);
    abstract public function cikar($number1, $number2);
    abstract public function carp($number1, $number2);
    abstract public function bol($number1, $number2);

    public function showResult (){
        echo("Result: " . $this->result);
    }

}

class UserClass extends CalClass {

    function __construct() {}

    public function topla($number1, $number2){
        $this->result = $number1 + $number2;
    }

    public function cikar($number1, $number2){
        $this->result = $number1 - $number2;
    }

    public function carp($number1, $number2){
        $this->result = $number1 * $number2;
    }

    public function bol($number1, $number2){
        $this->result = $number1 / $number2;
    }

}

$user = new UserClass();
```

```
$user->topla(25, 15);
$user->showResult();
```

Örneğimizin de ekran çıktısı aşağıdaki gibi olacaktır.

```
Result: 40
```

Ekran alıntısında da görüldüğü üzere; başarılı bir şekilde abstract class'ımızı uyguladık.

BÖLÜM 10: interface

Class'lar arasında inheritance (kalıtım) yaparken; sadece bir class'ı üst class olarak seçmemiz oldukça kısıtlayıcı bir durum. Bir class'a birden fazla class extends edileceği zaman, bu class'lar arasında deyim yerindeyse; waterfall (şelale) misali bir hiyerarşi kurmamız gerekir. Yani "onu oraya koy, bunu şuraya yerleştir, şunu ver..." gibi bir çalışma içerisinde oluyoruz.

abstract class'lar için de durum yine aynı. Bir önceki bölümde, abstract class'lar için: normal class'lar ve interface'lerin özelliklerini taşır demiştim. Bildiğiniz gibi abstract class'larda hem gövdesi olan hemde gövdesi olmayan fonksiyonlar bulunur. interface'ler de ise yalnızca gövdesi olmayan fonksiyonlar bulunur. Bu nasıl olur veya ne işimize yarayacak? Gibi sorular aklınıza gelebilir. Hemen açıklayayım. Bir interface, yapılacak olan projeye bir taslak getirir. Bünyesinde olan fonksiyonlar, istenilen class'ta uygulanarak tekrar tekrar farklı şekillerde kullanılabilir. Örneğin bir interface'de write() adında gövdesi olmayan bir fonksiyon varsa, bunu; A class'ında kullanıcı bilgilerini yazdırmak için kullanabilirim, B class'ında veri tabanına kayıt işleminde kullanırım veyahut C class'ın session başlatmak için kullanabilirim. Yani interface'te tanımlanan gövdesiz fonksiyonlar ile istediğim herşeyi yapmakta özgürüm.

Bir interface aşağıdaki gibi tanımlanır.

```
interface Example {

}
```

Görüldüğü gibi interface tanımlamasında class anahtar kelimesi yok. Yerine sadece interface yazacağız ve bu interface'nin adını yazacağız. Yani normal class tanımlamasından biraz farklı.
Bir interface içerisinde yalnızca gövdesi olmayan fonksiyonlar bulunur dedik. Şimdi bunu uygulayalım.

```
interface Example {

    public function getVal();
    function setVal($val1, $val2, $val3);

}
```

interface içerisinde tanımlanacak olan fonksiyonlar 2 şekilde tanımlanırlar. 1. Tanımlanacak olan fonksiyon public olarak tanımlanabilir. 2. Tanımlanacak olan fonksiyon default olarak tanımlanabilir.

Peki interface içerisinde variable (değişken) tanımlanabilir mi? Elbette tanımlanabilir. Mesela tanımlayalım.

```
interface Example {
```

```
    public $var1;
    protected $var2;
    private $var3;

    public function getVal();
    function setVal($val1, $val2, $val3);

}
```

Ekran çıktısına bir bakalım.

```
Fatal error: Interfaces may not include member variables in
/opt/lampp/htdocs/untitled.php on line 4
```

O da ne? Bir hata aldık. Bu hatada bize dediği şey: interface'ler, üye değişken içermez! Evet, aldığımız hatanın da belirttiği gibi bir interface değişken içermez. Aslında içerir. Şöyle içerir: interface içerisinde tanımladığımız değişkenler sabit değişken olursa! O zaman değişkenlerimizi sabit yapalım ve tabiki public | protected | private erişim belirleyicileri kaldıralım.

```
interface Example {

    const VAR1 = "Emre Can ÖZTAŞ";
    const VAR2 = 42;
    const VAR3 = 639;

    public function getVal();
    function setVal($val1, $val2, $val3);

}
```

Değişkenlerimiz sabit olduğu için ilk değer atamalarını gerçekleştirdik. Yani; interface içerisinde sadece ve sadece const anahtar kelimesi ile tanımlanmış olan sabit değişkenlere izin verilmektedir.

O halde bir interface tanımlayalım. Daha sonra bu interface'yi bir örnekte kullanalım. interface'miz aşağıdaki gibi olsun.

```
interface Example {

    const PI_VALUE = 3.14;

    function sayHello();
    function writeWhatEver($value);

}
```

interface'mize basit olarak 2 tane fonksiyon ve bir de sabit değişken tanımladık. Artık bu interface'mizi bir class'ta kullanalım. Bir interface, inheritance (kalıtım) konusunda

olduğu gibi extends anahtar kelimesi ile bir başka class'a eklenemez. Peki nasıl eklenir? Cevabımız bir sonraki alt başlıkta!

10.1 implements

Bir interface, başka bir class'a extends anahtar kelimesi ile eklenemez dedik. Dolayısıyla interface'i bir class'a implements anahtar kelimesi ile eklemeliyiz. O halde bir class tanımlayalım ve tanımladığımız Example isimli interface'i bu class'ımıza implement edelim.

```
interface Example {

    const PI_VALUE = 3.14;

    function sayHello();
    function writeWhatEver($value);

}

class Main implements Example {

}
```

Evet, dediğimizi yaptık ve Example isimli interface'i, Main class'ımıza implement ettik. interface içerisinde gövdesi olmayan (abstract) fonksiyonlar bulunduğu için bu fonksiyonların Main class içerisinde gövdeye kavuşması gerekiyor. Biz de öyle yapalım.

```
class Main implements Example {

    function sayHello(){
        echo("Hello<br>");
    }

    function writeWhatEver($value)
    {
        echo("WTF: " . $value);
    }

}
```

Her şey tamam. Şimdi, Main class'tan bir instance oluşturalım. Daha sonrada fonksiyonlarımızı yazalım. Örneğimizin tam hali aşağıdaki gibi olacaktır.

```
<?php

interface Example {

    const PI_VALUE = 3.14;

    function sayHello();
    function writeWhatEver($value);
```

```
}

class Main implements Example {

    function sayHello()
    {
        echo("Hello<br>");
    }

    function writeWhatEver($value)
    {
        echo("WTF: " . $value);
    }

}

$main = new Main();
$main->sayHello();
$main->writeWhatEver("The weather is hot!");
```

Örneğimizin ekran çıktısı da aşağıdaki gibi olacaktır.

```
Hello
WTF: The weather is hot!
```

Yukarıdaki ekran alıntısında da görüldüğü üzere; interface'i başarıyla implement ettik. Konumuzun başında size bir şey söylemiştim. Bir interface'i implement ettiğimiz her class'ta farklı işlemler yaptırabiliriz. Şimdi bunun hakkında bir örnek yapalım. Örneğimiz aşağıdaki gibi olacaktır.

```
<?php

interface Example {

    const PI_VALUE = 3.14;

    function sayHello();
    function writeWhatEver($value);

}

class Main implements Example {

    function sayHello()
    {
        echo("Hello<br>");
    }

    function writeWhatEver($value)
    {
        echo("WTF: " . $value);
    }
```

```
}

class Reserved implements Example {

    function sayHello()
    {
        echo("Moikka<br>");
    }

    function writeWhatEver($value)
    {
        echo("Q: " . self::PI_VALUE * $value);
    }

}

$main = new Main();
$main->sayHello();
$main->writeWhatEver("The weather is hot!");

$reserved = new Reserved();
$reserved->sayHello();
$reserved->writeWhatEver(639);
```

Yukarıdaki örneğimizde; Reserved class'a da Example interface'i implement ettik ve kullandık. Bu class içerisinde de interface içerisinde tanımlı olan fonksiyonlara farklı işlemler yaptırdık. Örneğin; interface içerisinde tanımlı olan PI_VALUE değerini, parametre olarak gelen $value değeri ile çarptık ve ekrana yazdırdık. Sözün kısası dostlar; bir interface, yapılacak uygulamaya bir şablon sunar. Yani sınırları çizer. Sizde bu sınırlar içerisinde kalarak gerekli işlemleri yerine getirebilirsiniz. interface'i implement eden class'lar içerisinde dilenirse yeni üyeler de eklenebilir. Burada herhangi bir kısıtlama yok. Lakin daha öncede dediğimiz gibi şayet interface'i implement edecekseniz; interface içerisindeki fonksiyonların gövdesini, implement ettiğiniz class içerisinde yazmalısınız.

Bir class birden fazla interface'i implement edebilir. Burası çok önemli. Bir daha söyleyelim. Herhangi bir class, istediği kadar interface'i implement edebilir. Yani burada, ìnheritance konusunda olduğu gibi sadece bir class'ın miras alınması yoktur. Söylemek istediklerimizi aşağıdaki gibi bir uygulama açıklayacaktır.

```
interface A {

}

interface B {

}

interface C {

}
```

```
class MainClass implements A, B, C {

}
```

Yukarıdaki yapımızda; A | B | C interface'lerini, MainClass implement etti. Burada dikkat ederseniz; implement edilecek olan interface'ler arasına virgül (,) konularak interface isimleri yazılır. Tabiki de interface'ler içerisinde olan abstract fonksiyonların, implement edildikleri yerde gövdeleri yazılmalıdır.

Şimdi başka bir yapıyı sizlere göstermek isterim. Öncelikle yapımızı görelim.

```
abstract class ResultClass {

}

interface A {

}

interface B {

}

interface C {

}

class MainClass extends ResultClass implements A, B, C {

}
```

Yukarıdaki yapımızda da görüldüğü üzere; hem extends hem de implements anahtar kelimelerini bir arada kullanabilir ve çeşitli class ve interface'leri bir class'a ekleyebiliriz. Lakin öncelikli olarak extends anahtar kelimesi kullanılmalıdır, böyle durumlarda.
Sizlere son olarak göstermek istediğim bir kavram var. interface'ler kendi aralarında inheritance (kalıtım) yapabilirler. Kalıtım yine aynı bilindik şekilde yapılır. Yani interface'ler arasında kalıtım kullanılacaksa; extends anahtar kelimesi kullanılmalıdır. Aşağıdaki yapımızı inceleyelim.

```
interface A {

}

interface B extends A{

}
```

Yukarıdaki yapımızda; interface A, interface B tarafından extends edilmiş. Burada şöyle bir özellik var. Bir interface birden fazla interface'i extends edebilir. Böyle bir kural

normal class'lar için geçerli değildir lakin interface'ler için böyle bir durum söz konusudur. Yani demek istediklerimiz aşağıdaki gibidir.

```
interface A {

}

interface B {

}

interface C {

}

interface D extends A, B, C {

}
```

Aklıma gelen son bir şey daha var, sevgili dostlar. Bir interface'den instance oluşturulamaz. Bunun nedenini siz de ben de çok iyi biliyoruz. Lakin yine de açıklayayım. Bir interface içerisinde sadece ve sadece abstract fonksiyonlar olacağı için object tanımlaması yapılamaz. Buna dikkat edelim.

BÖLÜM 11: Polymorphism

Polymorphism kelime anlamı olarak; Çok biçimlilik manasına gelmektedir. Polymorphism, OOP dünyasında önemli bir yer tutar. Birden fazla class'tan instance tanımlamak ve her class'ta, kalıtım hiyerarşisindeki fonksiyonları kullanmak yerine daha kestirme yoldan yani polymorphism kullanarak işlerimizi kolaylaştırabiliriz. Kısaca; polymorphism ile yazılan kodun miktarı azaltılabilmektedir.

Aslında PHP'deki polymorphism'in ilginç bir yapısı vardır. Örneğin; Java'da bu kavram daha farklı kullanılmaktadır. Herneyse, basit bir örnek ile konumuzu anlatmaya çalışalım.
Öncelikle bir super class yazalım ve bu class'ı extends eden sub class'lar yazalım. Yapımız aşağıdaki gibi olsun.

```
class People {

    function sayYourJob()
    {
        echo("Here is super class!");
    }

}

class Engineer extends People
{

    function sayYourJob(){
        echo("I'm an Engineer!");
    }

}

class Doctor extends People {

    function sayYourJob()
    {
        echo("I'm a doctor!");
    }

}
```

Kalıtım şemamızı kurduk. Şimdi gelelim buradaki polymorphism kullanımına. People class'ı bizim super class'ımız. Bu class'ı diğer class'lar (Engineer || Doctor) class'ları extends ettiler. Şimdi burada; sub class'lardan ayrı ayrı object'ler oluştururarak kullanabilirim. Ya da burada polymorphism yapabilirim. Gelin burada polymorphism uygulayalım. Bunun için aşağıdaki kodları yazmam yeterli olacaktır.

```
function whichClass ($obj)
{
    if($obj instanceof People)
```

```
    {
        $obj->sayYourJob();
    } else
    {
        echo("Error!");
    }
}
```

Fonksiyonumuz, class'lardan ayrı, herhangi bir class'ın scope (etki alanı) alanı dışında. Peki bu fonksiyonumuz ne yapar? Bu fonksiyon içerisinde, kendisine parametre olarak gelen object'in People class'ı ile arasında bir bağ var mı yok mu kontrol eder. Eğer gelen object'in People class'ı ile bir yakınlığı varsa; sayYourJob() fonksiyonunu çağıracaktır. Yoksa da Error! şeklinde bir uyarı verecektir.

Şimdi fonksiyonumuzu nasıl kullanacağımızı gösterelim. Daha sonrada örneğimizin tam halini paylaşalım ve ekran çıktımıza bir bakalım.

```
whichClass(new Engineer());
whichClass(new Doctor());
```

Yukarıdaki yapımızda; whichClass isimli fonksiyonumuzu çağırdık ve parametre olarak, ilgilendiğiniz class'ların nesnelerini gönderdik. Burada basit olması için: new objectName() yapısını kullandık. Zira klasik olarak; bir object oluşturup bu object'i de gönderebilirsiniz. İkisi de aynı şeydir. Peki şimdi fonksiyonumuza geri dönelim. Fonksiyonumuz içerisindeki if deyimiyle; bu gönderdiğimiz object'ler nasıl kontrol edilebiliyor ve gerekli işlemler yerine getirilebiliyor? Burada bakılan şey; extends. Engineer ve Doctor class'ları People class'ından extends edildiği gibi otomatik olarak; People class'ının child (çocuk)'ları statüsüne dönüşüyorlar. Bunu bir ağaç yapısı olarakta düşünebilirsiniz. Ağaç, People ve dalları da Engineer | Doctor class'ları olacaktır. Bu her iki class'tan oluşturulan object'ler de dolayısıyla People class'ını da işaret edecektir.

Yazdığımız örneğimizin tam hali aşağıdaki gibi olacaktır.

```
<?php

class People {

    function sayYourJob()
    {
        echo("Here is super class!");
    }

}

class Engineer extends People
{

    function sayYourJob(){
        echo("I'm an Engineer!");
    }
```

```
}

class Doctor extends People {

    function sayYourJob()
    {
        echo("I'm a doctor!");
    }

}

function whichClass ($obj)
{
    if($obj instanceof People)
    {
        $obj->sayYourJob();
    } else
    {
        echo("Error!");
    }
}

whichClass(new Engineer());
whichClass(new Doctor());

?>
```

Yazdığımız örneğimizin ekran çıktısı da aşağıdaki gibi olacaktır.

```
I'm an Engineer!I'm a doctor!
```

Görüldüğü gibi; polymorphism sayesinde yazdığımız kodların sayısı azalmakta ve aynı zamanda, inheritance kavramının bir ağaç yapısına benzediğini, sub class'ların bu ağacın dalları olduğunu daha yakından görebilmekteyiz.

BÖLÜM 12: OOP Magic Member

Bu bölümde; OOP işlemlerinde oldukça yararlı olan, PHP tarafından default (varsayılan) olarak bizlere sunulan bir takım, deyim yerindeyse Magic (Büyülü) fonksiyonlardan ve değişkenlerden bahsedeceğiz.

Hazırsanız, o halde başlayalım.

12.1 Variable

Bu alt başlık altında; işlerimizi kolaylaştıran, daha doğru bazı durumlarda bilgi almak için kullanacağımız, PHP tarafından default olarak bizlere sunulan variable (değişken)'lerden bahsedeceğiz. Bu değişkenlerin bazıları OOP dünyası için geçerli değil. Belki de bazılarını biliyorsunuz. Lakin fazla bilgiden kimseye zarar gelmez. O yüzden bu değişkenlerden de bahsedeceğiz.

12.1.1 __CLASS__

__CLASS__ değişkeni içerisinde bulunulan class'ın ismini verir.

Örneğin:

```
class Example {
    function __construct() {
        echo(__CLASS__);
    }
}
```

Yukarıdaki kullanımda ekran çıktısı:

```
Example
```

Olacaktır.

12.1.2 __METHOD__

__METHOD__ değişkeni, içinde bulunduğu fonksiyonun ve bu fonksiyonun içerisinde bulunduğu class'ın ismini verir.

Örneğin:

```
class Example {
    function __construct() {
        echo(__METHOD__);
    }
}
```

Yukarıdaki kullanımda ekran çıktısı:

```
Example::__construct
```

olacaktır.

12.1.3 __FUNCTION__

__FUNCTION__ değişkeni, içerisinde bulunduğu fonksiyonun ismini verir.

Örneğin:

```
class Example { function construct() { echo(__FUNCTION); } }
Yukarıdaki kullanımda ekran çıktısı:
__construct
```

olacaktır.

12.1.4 __DIR__

__DIR__ değişkeni, dosyanın bulunduğu dizini verir.

Örneğin:

```
class Example {
    function __construct() {
        echo(__DIR__);
    }
}
```

Yukarıdaki kullanımda ekran çıktısı:

```
/opt/lampp/htdocs/foldername
```

olacaktır (en azından bana göre).

12.1.5 __FILE__

__FILE__ değişkeni, dosyanın tam ad ve yolunu verir.

Örneğin:

```
class Example {
    function __construct() {
        echo(__FILE__);
    }
}
```

Yukarıdaki kullanımda ekran çıktısı:

```
/opt/lampp/htdocs/filename.php
```

olacaktır.

12.1.6 __LINE__

__LINE__ değişkeni, bulunduğu satırın numarasını verir.
Örneğin:

```
class Example {
    function __construct() {
        echo(__LINE__);
    }
}
```

Yukarıdaki kullanımda ekran çıktısı:

```
5
```

olacaktır (bende böyle!).

12.1.6 __NAMESPACE__

__NAMESPACE__ değişkeni, dosyanın namespace adını verir.

Örneğin:

```
namespace Package;
class Example {
    function __construct() {
        echo(__NAMESPACE__);
    }
}
```

Yukarıdaki kullanımda ekran çıktısı:

```
Package
```

olacaktır.

12.1.6 __TRAIT__

__TRAIT__ içinde bulunduğu trait yapısının ismini verir.

Örneğin:

```
trait User {

    function sayHello(){
        echo(__TRAIT__);
    }

}
```

Yukarıdaki kullanımda ekran çıktısı:

```
User
```

olacaktır.

12.2 Function

Bu alt başlık altında çeşitli, yararlı ve PHP'nin bizler içinde default olarak sunduğu çeşitli fonksiyonları inceleyeceğiz. Bu fonksiyonlar, OOP dünyasında sıklıklar kullanılır. Bence sizin de hoşunuza gidecek. Zaman kaybetmeden başlayalım.

12.2.1 __autoload($ClassName)

__autoload($ClassName) fonksiyonu, Bir class'tan instance oluşturulurken, şayet bahsedilen class yok ise; bu class'ı içeren dosyanın include edilmesini sağlar.

Örneğin.

```
$main = new Main();
```

şeklinde bir object oluşturmaya çalıştığımızı düşünürsek; şayet elimizde böyle bir class yok lakin herhangi bir dosyada ise, aşağıdaki şekilde bunu ekleyebiliriz.

```
function __autoload($ClassName)
{
    /**
    * $className: instance olusturulmak istenen
    * class'in adi.
    * class, $className ile ayni isimdeyse:
    * include($className . '.php');
    * ismi farkli ise:
    * include('fileName.php');
    */
}
```

Main adında bir class'ımız olmadığı için, __autoload($ClassName) fonksiyonu tanımlanan dosyayı eklemeye çalışacaktır.

12.2.2 __get($variable)

__get($variable) fonksiyonu, bir class içerisinde tanımlı olmayan lakin varmış gibi çağrılan bir properties (class değişkeni) olmadığı durumda çalışır. Hepimiz insanoğluyuz. Bir class'ta olmayan bir değişkeni çağırdığımızda sistem durabilir. Dolayısıyla bu fonksiyonu kullanmak oldukça yerinde bir karar olacaktır.

Örneğin:

```
class Example {
    function __get($var)
    {
        echo("I don't have like this variable!");
    }
```

```
}

$example = new Example();
echo($example->var);
```

Yukarıdaki örneğimizde; tanımlı olmayan $var değişkeni çağrılmıştır. Bu da __get($var) metodu ile tespit edilerek: I don't have like this variable! şeklinde bir uyarı verilmiştir. Tabiki daha farklı şeylerin de yapılması söz konusudur.

12.2.3 __set($variable, $value)

__set($variable, $value) fonksiyonu, __get($variable) fonksiyonunun aksine; tanımlı olmayan herhangi bir değişkene sanki o değişken tanımlıymış gibi atama yapıldığı sırada çalışmaya başlar.

Örneğin:

```
class Example {
    function __set($var1, $var2)
    {
        echo("Error! ". $var1 . " " . $var2);
    }
}

$example = new Example();
$example->var = 42;
```

Yukarıdaki örneğimizde; Example class'ı içerisinde tanımlı olmayan $var değişkenine atama yapılmak isteniyor. __set($var1, $var2) fonksiyonu ise bu olmayan değişkene atama işlemine karşı çıkar. Burada; __set($var1, $var2) fonksiyonunun iki tane parametreyi mutlaka almak zorundadır. Bu parametrelerden birisi; atama yapılmak istene değişlken ve diğer parametre de atanmak istenen değerdir.

12.2.4 __call($function, $args)

__call( ) fonksiyonu, bir class içerisinde tanımlı olmayan bir fonksiyon çağrıldığında çalışmaya başlar.

Örneğin:

```
class Example {
    function __call($function, $args){
        echo("Hello");
    }
}

$example = new Example();
$name = "Emre Can ÖZTAŞ";
$example->sayHello($name);
```

Yukarıdaki örneğimizde; Example class'ı içerisinde, sayHello () adında bir fonksiyon yoktur. Dolayısıyla; __call($function, $args) çalışmaya başlayacaktır. Burada; $function: fonksiyon ismi ve args: olmayan fonksiyona gönderilen parametrelerdir.

12.2.5 __toString( )

__toString( ) fonksiyonu, herhangi bir class'tan bir instance oluşturulduğu zaman bu instance ekrana yazdırılmaya çalışıldığı zaman hata verir. Çünkü bir object, string'e dönüştürülemez. İşte bu fonksiyon yardımıyla, bir object ekrana yazdırılmaya çalışıldığında ekrana istediğimizi yazdırabiliriz.

Örneğin:

```
class Example {
    function __toString(){
        return("This is Example class!");
    }
}

$example = new Example();
echo($example);
```

__toString( ) fonksiyonu, mutlaka return ike ekrana yazdırılacakları vermesi gerekir.

BÖLÜM 13: Object Serialization

Object Serialization, herhangi bir class'tan instance oluşturulduktan sonra; bu instance'nin kayıt edilebilir düzeyde String bir ifadeye çevrilmesidir. Bu size; herhangi bir object'in daha doğrusu object'in arkasında olan class'ın bilgilerini elde etme imkanı verir.

Serialization, genellikle PHP'nin çekirdeğinde kullanılan bir yöntemdir. Örneğin; Session işlemleri devamlı olarak serialization uygulamaktadır ve bu sonuçları: /tmp dizinine kaydetmektedir. Biz de bu bölümde; Object Serialization yönteminden bahsedeceğiz.

13.1 serialize (value)

serialize (value), value olarak aldığı parametre değerini kaydedilebilir düzeyde String ifadeye çevirir. serialize (value) fonksiyonunu, depolanmasını istediğiniz herhangi bir value değerinde kullanabilirsiniz. Örneğin; array, object, session v.s

Şimdi basit bir class yapısı kuralım ve demek istediklerimizi göstermeye çalışalım. Örneğimiz aşağıdaki gibi olacaktır.

```
<?php

class UserInfo
{
    private $name;
    private $surname;
    private $location;

    public function __construct()
    {
        $this->name = '';
        $this->surname = '';
        $this->location = '';
    }

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

    public function setSurname($surname)
    {
        $this->surname = $surname;
    }

    public function setLocation($location)
    {
        $this->location = $location;
    }

    public function getName()
```

```
    {
        return($this->name);
    }

    public function getSurname()
    {
        return($this->surname);
    }

    public function getLocation()
    {
        return($this->location);
    }
}

$user = new UserInfo();
$userSerialize = serialize($user);
echo($userSerialize);
```

Yukarıdaki örneğimizde; UserInfo class'ından oluşturduğumuz instance'i serialize ettik ve ekrana yazdırdık. Şimdi ekran çıktımıza bir bakalım.

```
O:8:"UserInfo":3:
{s:14:"UserInfoname";s:0:"";s:17:"UserInfosurname";s:0:"";s:18:"UserI
nfolocation";s:0:"";}
```

Görüldüğü gibi; serileştirdiğimiz object'imiz, ait olduğu class'ın değişkenlerini ve class ismini gösteriyor. Artık $userSerialize değişkeni, string yapısında. Yani kolayca saklanabilir.

Burada size farklı bir şey göstermek istiyorum. Aşağıdaki gibi object'imiz ile class'ımızın set fonksiyonlarını kullanalım ve değişkenleri dolduralım. Yani:

```
$user = new UserInfo();
$user->setName("Hazal");
$user->setSurname("ONEY");
$user->setLocation("ANKARA");
$userSerialize = serialize($user);
echo($userSerialize);
```

Şeklinde olsun. Şimdi ekran çıktımıza bakalım.

```
O:8:"UserInfo":3:
{s:14:"UserInfoname";s:5:"Hazal";s:17:"UserInfosurname";s:4:"ONEY";s:
18:"UserInfolocation";s:6:"ANKARA";}
```

Görüldüğü gibi; değişkenlerin sahip olduğu değerler de geldi. Yani oldukça kullanışlı bir yöntem. Bazı durumlarda kullanmanız gerekebilir. Benden söylemesi. Peki, herhangi bir object'i string hale getirdik ve kaydettik. Daha sonra nasıl tekrar eski haline çevireceğiz? Bu sorunun cevabı da bir sonraki alt başlıkta!

13.2 unserialize ($string)

unserialize ($string) ifadesi, serialize (value) ile string hale getirilen ifadenin tekrar eski haline getirilmesini sağlar. Yani; object'i string'e çevirmiş isek; string'i tekrar object haline getirebiliriz.

13.1 serialize (value) alt başlığında yaptığımız örneğinizi kullanalım. String'e çevirdiğimiz object'imizi tekrar eski haline getirelim. Örneğimiz aşağıdaki gibi olacaktır.

```
$user = new UserInfo();
$user->setName("Hazal");
$user->setSurname("ONEY");
$user->setLocation("ANKARA");

// serialize()
$userSerialize = serialize($user);
// echo($userSerialize);

// unserialize()
$userUnserialize = unserialize($userSerialize);
print_r($userUnserialize);
```

Yukarıdaki yapımıza ait olan ekran çıktısına bakalım.

```
UserInfo Object ( [name:UserInfo:private] => Hazal
[surname:UserInfo:private] => ONEY [location:UserInfo:private] =>
ANKARA )
```

Yukarıdaki ekran alıntısında da görüldüğü üzere; unserialize ($string) fonksiyonu ile object'imizi yeniden elde ettik.

BÖLÜM 14: namespace

namespace, Türkçe olarak; isim alanları şeklinde ifade edebiliriz. Peki namespace nedir? Ne işe yarar? Şimdi bu iki soru üzerinde yoğunlaşalım.

namespace'ler, yazılan sınıf, değişken veya fonsiyonlara bir standart getirir. Örneğin; aynı dizin altında aynı isimli iki dosya bulunduramazsınız. Başka bir örnek olarak; aynı PHP dosyası içerisinde aynı isimli değişken, fonksiyon veya class'ta bulunduramazsınız. İşte burada devreye namespace'ler girer. namespace'ler ile, bahsettiğimiz üyeler birbirlerinden isim alanlarıyla ayrılırlar. Yani bunu bir mahlas olarak veya bir package (paket) olarakta düşünebilirsiniz. Buradan hareketle bir PHP dosyası içerisinde; namespace ile ayrılmış aynı isme sahip değişken, fonksiyon ve class'ları bulundurabilirsiniz.

Şimdi bir isim alanı tanımlayalım.

```
<?php
namespace Documents;
```

namespace'ler bir PHP dosyasında, PHP tag'lerinden hemen sonra yani ilk satırda yer almalıdır. namespace üzerinde sadece declare anahtar kelimesi bulunabilir.

Bir veya daha fazla isim alanı yazacaksanız aşağıdaki yöntemi takip etmelisiniz.

```
<?php
namespace Documents\Settings\Music;
```

Yukarıdaki satırda da görüldüğü üzere; sanki bir dizin hiyerarşisiymiş gibi namespace'leri oluşturabilirsiniz.

Peki namespace'in nasıl tanımlandığını gördük. Şimdi de nasıl kullanılacağına bakalım.

Aşağıdaki örneğimizi inceleyelim.

```
<?php
namespace Documents\Settings\Music;
function sayHello()
{
    echo("Hello!");
}

\Documents\Settings\Music\sayHello();
```

Aşağıdaki örneğimizde; namespace'in altında bir fonksiyon tanımlaması yaptık. Artık bu fonksiyon belirttiğimiz namespace'e bağlı. Daha öncede dediğim gibi bu yapıyı bir paket gibi veya bir dizin yapısı gibi düşünebilirsiniz. namespace altındaki fonksiyonu ise; \namespace_full_name\member_name şeklinde çağırıyoruz. Aslında bu temel bir

yapı. Sadece fonksiyon için geçerli değil. Diğer üyeler içinde aynı şey geçerli. Hepsine biraz sonra değineceğiz.

Örneğin tanımladığımız namespace altında değişken tanımlaması da yapabiliriz. Lakin; namespace altında tanımlanacak olan değişkenler: sabit değişkenler olmalıdır. Bunun dışında tanımlanan; yaz/sil değişkenlere izin verilmemektedir.
Aşağıdaki yapımızı inceleyelim.

```
<?php
namespace Documents\Settings\Music;
const NAME = "J.Sawyer";
const NUMBER = 639;
const STATUS = TRUE;

function sayHello()
{
    echo("Hello!");
}

# \Documents\Settings\Music\sayHello();
echo(\Documents\Settings\Music\NAME);
echo(\Documents\Settings\Music\NUMBER);
echo(\Documents\Settings\Music\STATUS);
```

Yukarıdaki örneğimizde; tanımlamış olduğumuz const yani sabit değişkenleri yine aynı şekilde ekrana yazdırabiliriz veya herhangi bir işlemde kullanabiliriz.

Gelelim, namespace'lerin en çok kullanıldıkları alana: class'lar! Daha öncede belirttiğimiz gibi; bir PHP dosyasında aynı isme sahip olan üyelerin bulunmasına izin verilmez. Lakin namespace'ler yardımıyla bu engeli kaldırabiliriz.

Örnek bir namespace oluşturalım ve altında bir de class tanımlayalım.

```
<?php
namespace Package\Session;
class SessionControl
{

    private $username;

    function __construct()
    {
        session_start();
        echo("Session is started!<br>");
        $this->username = "";
    }

    function setUsername($username)
    {
        $this->username = $username;
        $this->setSessionName($this->username);
        echo($username . "<br>");
```

```
    }

    function setSessionName($username)
    {
        $_SESSION['username'] = $username;
        echo(session_id() . "<br>");
    }

    function stopSession()
    {
        session_destroy();
        echo("Session is destroyed!<br>");
    }

}
```

Örnek bir namespace oluşturduk ve altında SessionControl isminde basit bir class tanımlaması yaptık. Şimdi bu class'tan bir nesne oluşturalım ve class'ın üye fonksiyonlarını kullanalım.

```
$sessionObject = new \Package\Session\SessionControl();
$sessionObject->setUsername("jsawyer");
$sessionObject->stopSession();
```

Daha önce de belirttiğimiz gibi; namespace altında olanları; \namespace_full_name\member_name şeklinde çağrılabilir. Keza aynı şekilde bir object'te oluşturulabilir.

Yukarıdaki örneğimizin ekran çıktısını alalım.

```
Session is started!
nmkjqar8fmqk0or0na7hfc54t0
jsawyer
Session is destroyed!
```

Yukarıdaki ekran alıntısında da görüldüğü üzere; namespace alanlarını başarı ile uyguladık.

Bir PHP dosyası içerisinde birden fazla namespace alanı olabilir ve her namespace altında çeşitli üyeleri olabilir.

Örneğin:

```
<?php
namespace Admin\Session;
class SessionControl {

}

namespace Admin\Mail;
class SessionControl {
```

```
}
```

Yukarıdaki yapımızda; bir PHP dosyası içerisinde aynı isme sahip olan iki tane class oluşturabildik. Bu class'ları oluşturabilmek için; namespace'leri kullandık. namespace'ler farklı olduğu için bir class içerisinde aynı isme sahip birden fazla üye tanımlanabilir ve kullanılabilir.

Son yazdığımız örneğimizi biraz daha geliştirelim. Son şekli aşağıdaki gibi olsun.

```
<?php
namespace Admin\Session;

    class SessionControl {
        function __construct()
        {
            echo("Hello!");
        }
    }

namespace Admin\Mail;
    class SessionControl {
        function __construct()
        {
            echo("Moikka!");
        }
    }

$objOne = new \Admin\Session\SessionControl();
$objTwo = new \Admin\Mail\SessionControl();
```

Yukarıdaki örneğimizde; aynı isme sahip lakin isim alanları ile ayrılmış olan class'larımızdan ayrı ayrı olarak iki nesne oluşturduk. Yani herhangi bir problem yok. Dilediğimiz şekilde kullanabiliriz. Yeterki kullanmasını bilelim.

14.1 use ... as ...

use ... as ... anahtar kelimesi namespace'lerin isimlerini kısaltmak için kullanılır. Örneğin; çok basit bir class yapısı oluşturalım ve bu class yapısını ayrı bir dosya olarak kaydedelim.

```
<?php
/**
* file name: classes.php
*/
namespace com\say\hello;
class SayHello {
    function __construct() {
        echo("Hello!");
    }
}
```

Dosyamızı kaydettik. Şimdi başka bir dosya açalım ve bu dosyamızı include ile ekleyelim.

```
<?php
include('classes.php');
use com\say\hello\SayHello as Hello;

$newObject = new Hello();
```

Yukarıdaki kullanımda da gördüğünüz üzere; namespace içeren herhangi bir yapı kaydedilebilir ve daha sonra da include veya diğer herhangi anahtar kelimeler ile istenilen class'a eklenebilir. use ... as ... anahtar kelimesi, bize uzun olan namespace ismini kısaltmamızı sağlar. Aslında use ... as ... anahtar kelimesi sadece bu amaçla kullanılmaz. Genel bir kullanımı olmakla birlikte diğer herhangi bir alanda da kullanabilirsiniz. Örneğin; class veya fonksiyon isimleri v.s gibi.

BÖLÜM 15: trait

PHP 5.4.0 ile hayatımıza yeni bir kavram girmiştir. trait kelime anlamı olarak; özellik manasına gelmektedir. Peki trait ne işe yarar?

Bildiğiniz gibi; PHP, yalnız bir super class'tan inheritance (kalıtım) yapılmasına olanak tanır. Bu da oldukça sıkıntılı bir durumdur. İşte trait'ler burada devreye girerler. Class bağımlılıklarını ortadan kaldırmak ve sıklıkla kullanılan fonksiyonların yeniden kullanılmasını sağlamak amacıyla geliştirilmişlerdir. trait'ler için; bir class'ın gölgesi ya da izdüşümü desek yanlış olmaz, herhalde. Çünkü tanımlanma şekilleri ve kullanımları, normal class'lardan herhangi bir fark içermemekte.

Bir trait aşağıdaki şekildeki gibi tanımlanır.

```
trait TraitName {

}
```

Görüldüğü gibi; bir trait, trait anahtar kelimesi ile tanımlanır. Bir trait içerisinde değişken tanımlamaları da yapılabilir.

```
trait TraitName {
    public $val1;
    protected $val2 = 639;
    private $val3 = TRUE;
}
```

Burada çok önemli bir nokta var: bir trait içerisinde constant yani sabit değişken tanımlaması yapılmaz. trait'ler yapısı gereği sabit değişkenleri içeremez. Buna dikkat edelim.

Tabiki trait içerisinde fonksiyon tanımlamaları da gerçekleştirilebilir.

```
trait TraitName {

    public $val1;
    protected $val2 = 639;
    private $val3 = TRUE;

    public function funcName1(){ }
    protected function funcName2(){ }
    private function funcName3(){ }
    function funcName4(){ }

}
```

Yani bir trait içerisinde, class içerisinde yaptıklarınızın büyük bir bölümünü yapabilirsiniz. Burada herhangi bir kısıtlama yok.

Aşağı yukarı trait'leri anladık sayılır. Peki bir trait'i nasıl kullanacağız? Bunun cevabı da bir sonraki başlıkta!

15.1 use

Bir trait bir class içerisinde kullanılmak için tasarlanmıştır. Bölümümüzün girişinde de bahsettiğimiz gibi; inheritace (kalıtım) kısıtlamasından dolayı trait'ler geliştirilmiştir. Dolayısıyla bir trait'i kullanacağımız class içerisine; use anahtar kelimesi ile çağırıp kullanmalıyız. Yani anlatmak istediğimiz şu şekilde;

```
class ClassName {
    use TraitName;
}
```

Yukarıdaki yapımızda da görüldüğü gibi; trait'i çağırdık. Burada çok önemli bir şey var. Bir class içerisinde birden fazla trait eklenebilir. Yani:

```
class ClassName {
    use TraitName1, TraitName2, TraitName3...;
}
// veya
class ClassName {
    use TraitName1;
    use TraitName2;
    use TraitName3;
    use ...
}
```

Peki o halde basit bir trait tanımlayalım ve çalışmasını birlikte inceleyelim. traiti'imiz aşağıdaki gibi olsun.

```
trait LoginControl {
    private $username = "jsawyer";
    private $password = 123;

    function doControl($username, $password)
    {
        if($this->username === $username){
            if($this->password === $password){
                echo("Login is successful!");
            } else {
                echo("Login failed!");
            }
        } else {
            echo("Login failed!");
        }
    }
}
```

trait'imiz basit olarak static (statik, durağan) olarak; login (giriş) kontrolü yapıyor. Şimdi bir de class yazalım ve bu class'a trait'imizi ekleyelim. Class'ımız aşağıdaki gibi olacaktır.

```
class SystemClass {
    use LoginControl;
}
```

Evet, yanlış görmedeniz! Yazdığımız class bu kadar! Çünkü; daha önce yazdığımız trait'imizi class'ımıza ekledik. Artık; trait içerisinde olan herşey, SystemClass'ına da ait. Geriye sadece class'ımızdan bir instance oluşturmak kalıyor. Hemen oluşturalım ve doControl( ) fonksiyonunu çağıralım.

```
$user = new SystemClass();
$user->doControl("jsawyer", 123);

Yazdığımız kodlarımızın tamamı aşağıdaki gibi olacaktır.
<?php
trait LoginControl {
    private $username = "jsawyer";
    private $password = 123;

    function doControl($username, $password)
    {
        if($this->username === $username){
            if($this->password === $password){
                echo("Login is successful!");
            } else {
                echo("Login failed!");
            }
        } else {
            echo("Login failed!");
        }
    }
}

class SystemClass {
    use LoginControl;
}

$user = new SystemClass();
$user->doControl("jsawyer", 123);
?>
```

Ekran çıktımızda aşağıdaki gibi olacaktır.

```
Login is successful!
```

Görüldüğü gibi trait'i başarıyla uyguladık. Doğru kullanıldığı taktirde; trait'ler oldukça kullanışlı yapılardır.

trait'ler hakkında konuşmaya devam edelim. Çünkü trait'ler hakkında bahsedecek çok şeyimiz var.

Bir trait içerisinde tanımlanan değişkenin değeri değiştirilebilir veya değer ataması yapılabilir. Bunun dışında; trait içerisindeki fonksiyonlar override (ezme) edilebilirler. Yine örneğimiz için konuşursak:

```
class SystemClass {
    use LoginControl;

    function doControl($username, $password)
    {
        $username = str_shuffle($username);
        $password = str_shuffle($password);
        echo($username . " " . $password);
    }
}
```

SystemClass içerisinde; doControl () fonksiyonunu override ettik. Artık trait içerisinde aynı isimle bulunan diğer fonksiyon çalışamayacak, class'ta revize ettiğimiz fonksiyon çalışacaktır.

15.2 insteadof

Bazı trait'ler içerisinde aynı isimli fonksiyonlar tanımlanmış olabilir. İnsanlık hali. İşte böyle durumlarda; sistem hangi fonksiyonu çalıştıracağını bilemez. Dolayısıyla bu sorunun ortadan kaldırılması lazım. Sistem ciddi şekilde hata verebilir, işleyişini durdurabilir. insteadof anahtar kelimesi böyle durumlarda, karışıklığı çözmek için kullanılır.

Aşağıdaki gibi bir yapımız olduğu varsayalım.

```
trait LoginControl {
    private $username = "jsawyer";
    private $password = 123;

    function doControl($username, $password)
    {
        if($this->username === $username){
            if($this->password === $password){
                echo("Login is successful!");
            } else {
                echo("Login failed!");
            }
        } else {
            echo("Login failed!");
        }
    }
}

trait SessionControl {
    function doControl($username, $password)
```

```
    {
        session_start();
        $_SESSION['username'] = $username;
        $_SESSION['password'] = $password;
    }
}

class UserClass {
    use LoginControl;
    use SessionControl;
}
```

Yukarıdaki yapımızda; LoginControl ve SessionControl trait'leri aynı fonksiyona sahipler. İki trait içerisinde; fonskiyonlar aynı ama iki fonksiyon da değişik işler yapıyorlar. Şimdi size ekran çıktısını göstermek istiyorum. Ekran çıktımız aşağıdaki gibi olacaktır.

```
Fatal error: Trait method doControl has not been applied, because
there are collisions with other trait methods on UserClass in
/opt/lampp/htdocs/untitled.php on line 29
```

Görüldüğü gibi hata aldık. Çünkü iki trait içerisinde de aynı işim fonksiyon var. Bu bizim için bir sorun teşkil ediyor. Dolayısıyla bu sorunu çözelim. Giriş bölümünde de belirttiğimiz gibi bu sorunu; insteadof anahtar kelimesi ile çözeceğiz. O halde hadi çözelim.

```
class UserClass {
    use LoginControl, SessionControl {
        LoginControl::doControl insteadof SessionControl;
        // SessionControl::doControl insteadof LoginControl;
    }
}
```

Yukarıda farklı bir şey yaptık. İki trait'i de ekledik lakin; LoginControl trait'inin sahip olduğu doControl ($username, $password) fonksiyonunu kullanacağımızı söyledik. Peki bunu nasıl söyledi? İşte bu şekilde: LoginControl::doControl insteadof SessionControl. Burada aslında; insteadof kelimesi yerine manasını taşıyor. Diğer ihtimal de kullanmak isteseydik o zamanda açıklama satırları ile kapattığımız satırı aktifleştirmemiz gerekirdi.

15.3 as

trait'ler için as özel bir kelime. Bir kaç farklı yerde kullanılıyor. Bunlara değineceğiz. as kelimesinin en önemli kullanıldığı alan: fonksiyonların görünürlüklerini ve isimlerini değiştirme. Örneğin; trait içerisinde tanımlı olan herhangi bir fonksiyon private ise bunu public veya protected yapabilirsiniz. Daha da ötesi bu fonksiyonun ismini değiştirebilirsiniz.
O halde örnek bir trait yazalım. trait'imiz aşağıdaki gibi olsun.

```
trait ExampleTrait {
    private function setName($name)
```

```
    {
        $this->name = $name;
    }
}
```

Şimdi bu trait'imizi uygulayacak bir class yazalım.

```
class ExampleClass {
    use ExampleTrait{
        setName as public;
    }
}
```

Class'ımız içerisinde; use anahtar kelimesinin scope (etki alanı) içerisinde; private olan özelliğini public olarak değiştirdik.

Aşağıdaki yapı ile fonksiyonun adını da değiştirebiliriz.

```
class ExampleClass {
    use ExampleTrait{
        setName as set;
    }
}
```

Lakin fonksiyonumuz private olduğu için kullanamayacağız. Dolayısıyla hem fonksiyonun adını değiştirelim hemde erişim belirleyicisini değiştirelim.

```
class ExampleClass {
    use ExampleTrait{
        setName as public set;
    }
}
```

Görüldüğü trait içerisinde değişik işlemler gerçekleştirebiliriz.

KAYNAKLAR

[1]. http://php.net/manual/en/language.oop5.php

[2]. http://www.killerphp.com/tutorials/object-oriented-php/

[3]. http://phpenthusiast.com/object-oriented-php-tutorials

[4]. http://jaskokoyn.com/php-oop-tutorial-series/

[5]. http://www.w3programmers.com/php-object-oriented-programming-part-1/

[6]. http://www.w3programmers.com/php-object-oriented-programming-part-2/

[7]. http://www.w3programmers.com/php-object-oriented-programming-part-3/

[8]. http://www.w3programmers.com/php-object-oriented-programming-part-4/

[9]. http://www.w3programmers.com/php-object-oriented-programming-part-5/

[10]. http://www.w3programmers.com/php-object-oriented-programming-part-6/