PHP'de Interface ve Abstract Class Kullanımı

[PHP](https://burakozdemir.co.uk/category/php)

Interface birçok şekilde tanımlanabilir, ama şu ana kadar gördüğüm en sade ve net tanımlamayı yapacak olursam, bir objenin(nesnenin) bir başka objenin ne olduğuna bağlı olmaması, ama o objenin yapabildiklerine bağlı olmasını düzenleyen, objelerin karşılıklı anlaşması olarak düşünebilirsiniz. Ancak tabii ki bu tanım da yeterli kalmayabilir, o yüzden biraz daha bu kavramı gerçek hayattan bir örnekle açıklayacağım.  
  
İşe geç kaldınız ve evden en kısa sürede çıkmanız lazım, ancak eşiniz, siz tam çıkarken, akşama ne yemek yapayım sorusunu soruyor, siz de herhangi bir **şey** yap, yeter ki **yiyebilelim** diye geçiştirip çıkıyorsunuz. Yani eşinize ne yapması gerektiğini belirtmeyip, özellik(interface) belirterek eşinizin üreteceği çıktının(objenin), **yenilebilir** özelliğe sahip herhangi bir şey olmasını bekliyorsunuz. Ne yapacağı ise eşinize kalmış, ister önünüze makarna koyar, isterse de ot, çünkü siz sadece **yenilebilir** bir şey yapmasını istediniz. Eğer sorusunu fazla önemsemediğiniz için size kızıp, tabağınıza taş koyarsa, duruma anlam veremediğiniz için(taş yenilenemez, dolayısıyla istediğiniz şartı sağlamıyor) yan sokaktaki pilavcının yolunu tutmak zorunda kalabilirsiniz.  
  
Herhangi bir programlama dilinde de durum aynı, eğer herhangi bir interface'i uygulayan bir class(sınıf), o interface'in yöntemlerini uygulamazsa, compiler(derleyici) hatası oluşur.   
  
Yukarıdaki örneği alttaki gibi uygulayabiliriz. İstediğimiz tek özellik yemeğin yenilebilir olması.

**interface Eatable   
{  
    public function getContents();  
 public function getName();  
}**

Ancak akşam eve gelirken malzemeleri almayı unuttu, size sunabileceği, sizin yiyebileceğiniz hiçbir şey de yok.   
  
Bu durumda Eatable interface'inin sahip olduğu **getContents()**ve**getName()** yöntemleri uygulanmadığı için compiler hatasıyla karşılaşırsınız.

**class Dinner implements Eatable   
{  
    public function serve(){}  
}**

Ancak eğer eşiniz size domates soslu makarna yapmaya karar verdi ve bunun için de, eve gelmeden önce marketten tuz, zeytinyağı, makarna ve domates sosu aldıysa, yemeğiniz siz geldiğinizde hazır olacaktır.

**class Dinner implements Eatable   
{  
 public function serve()  
 {   
 return "Akşam yemeğim: " . $this->getName();   
 }  
  
    public function getName()  
    {  
        return "Domates soslu makarna";  
    }  
  
    public function getContents()  
    {  
         return ["domates sosu", "makarna", "tuz", "zeytinyağı"];  
    }  
}**

Peki neden interface kullanmak önemli diye düşünecek olursanız, en önemli kullanım alanının önemli bir yazılım terminolojisi olan dependency injection için olduğunu söyleyebilirim. Bu terminolojiyi ilk defa duymuş olabilirsiniz ama inanılmaz derecede önemli olduğunu söylebilirim. Ufak bir örnek verip devamında açıklayacağım. Örneğin, bir ziyafet verdiğinizi düşünün, bu ziyafette de yemek servis edeceğinizi, bu yemeğin de pizza olduğunu düşünün. Bunu iki şekilde yapabilirsiniz.  
  
Muhtemelen aklınıza gelen ilk yöntem alttaki gibi olacaktır.

**class Pizza {}  
  
class Feast   
{  
     protected $food;  
  
     public function \_\_construct()  
     {  
         $this->food = new Pizza();  
     }  
}**

Ancak dependency injection ile ziyafet sınıfınızı oluşturacak olsaydınız, alttaki gibi oluştururdunuz.

**class Pizza {}  
  
class Feast   
{  
     protected $food;  
  
     public function \_\_construct(Pizza $pizza)  
     {  
         $this->food = $pizza;  
     }  
}**

Üstteki gerçeklemeyle, ziyafetinizin pizza olmadan gerçekleşemeyeceğini belirtmiş olduğunuz gibi, ziyafetin kendi içindeki akışını pizza dışarıdan servis edildiği için, ayırmış oldunuz. Böylece birbirine bağımlı sınıflar, kendi içlerindeki akışları itibariyle bağımsız oldular.   
  
Ancak bu tasarımla da sadece pizza servis edebiliyorsunuz, ileride fikrinizi değiştirip makarna servis edeceğinizde, problem yaşamanız mümkün. İşte burada interface kullanımıyla bu sorunu aşabilirsiniz.

**interface Eatable   
{  
    public function getContents();  
 public function getName();  
}  
  
class Pizza implements Eatable  
{      
 public function getName()  
    {  
    return "Pizza";  
  }  
  
    public function getContents()  
    {  
    return ["margarita", "mısır", "mantar", "sosis"];  
    }  
}  
  
class Pasta implements Eatable  
{      
 public function getName()  
    {  
 return "Makarna";  
    }  
  
    public function getContents()  
    {  
     return ["domates sosu", "makarna", "tuz", "zeytinyağı"];  
    }  
}  
  
class Feast   
{  
     protected $food;  
  
     public function \_\_construct(Eatable $food)  
     {  
         $this->food = $food;  
     }  
}**

Böylece, dilediğiniz zamanda vereceğiniz herhangi bir ziyafette, yenilebilir herhangi bir yemeği insanlara sunabilirsiniz, çünkü tek bağlılığınız, sunulacak şeyin yenilebilir olması.   
  
Buna benzer örneklerde gördüğüm en büyük hata, Feast gibi, aslında yemek olmayan, ancak yemek sunulan bir sınıfı, yemekten inherit ederek(kalıtımla) oluşturmak oluyor ki bunu yaptığınız takdirde, tasarımınız tasarım olmaktan çıkıyor.  
  
Bunun dışında aynı interface'i dilediğiniz gibi kullanabileceğiniz gibi, aynı zamanda da birden fazla interface'i bir class üzerinde uygulayabilirsiniz. Örneğin yemeğinizin yenilebilir olmasının yanında, bir başka özelliğinin daha bulunmasını istediniz. Alttaki gibi bu interface'i de uygulayabilirsiniz.

**interface Something{}  
  
class Dinner implements Eatable, Something {}**

Son olarak ufak bir not, interface içerisindeki yöntemlerin hepsi public olmalıdır, protected veya private yöntemler tanımlarsanız derleyici hatası ile karşılaşırsınız.  
  
Abstract Class ise, en kaba tabirle, kendisinden obje üretilemeyen class'tır. Genellikle verilen klişe örnek ise meyveler ya da ağaçlar üzerinedir. Sizin Tree adında abstract bir class'ınız olur, daha sonrasında da bu class üzerinden limon, elma vb. yeni class üretirsiniz. Limonun da elmanın da ağacı vardır, ikisinin birbirinden farklı noktaları olduğu gibi aynı oldukları noktalar da olur. Örneğin her ikisi de yapraklarını döker, ama her ikisinin de ortalama boyu farklıdır.

**abstract class Tree   
{  
 public function dropLeaves()  
 {  
 echo "Yapraklar dökülüyor.\n";  
 }  
  
 abstract public function getHeight();  
}  
  
class Lemon extends Tree   
{   
 public function getHeight()  
 {  
 return 17;  
 }  
}  
  
class Apple extends Tree   
{  
 public function getHeight()  
 {  
 return 48;  
 }  
}**

Eğer siz elmadan da limondan da yeni bir nesne üretseniz, dropLeaves fonksiyonuyla her ikisinden de "Yapraklar dökülüyor." çıktısını alabilirsiniz. 

**$lemon\_tree = new Lemon();  
echo $lemon\_tree->dropLeaves(); *// Yapraklar dökülüyor.*  
$apple\_tree = new Apple();  
echo $apple\_tree->dropLeaves(); *// Yapraklar dökülüyor.***

Abstract class tanımını unutmayın, direkt olarak Tree class'ından obje üretmeye çalışırsanız hata alırsınız.

**$tree = new Tree(); *// Hata***

Üstteki örneğimizi tekrar ele alalım ve öğün abstract class'ından inheritance(kalıtım) ile üretilen, akşam yemeği ve kahvaltı class'larını oluşturalım.

**abstract class Meal   
{  
    abstract public function serve();  
}  
  
class Dinner extends Meal   
{  
    public function serve()  
   {   
    return "Akşam yemeğim: Domates soslu makarna";  
   }  
}  
  
class Breakfast extends Meal   
{  
    public function serve()  
    {  
    return "Kahvaltım: Sucuklu yumurta";  
    }  
}**

Tekrardan üstteki ziyafet sınıfı ile ilgili bir örnek gerçekleştirecek olursak, onu da alttaki gibi gerçekleyebilirsiniz, böylece ziyafetinizde bir öğün servis edeceğinizi belirtiyorsunuz, ancak bu öğünün akşam yemeği mi kahvaltı mı olacağını belirtmiyorsunuz, dolayısıyla herhangi bir ziyafetinizde akşam yemeği, bir başka ziyafetinizde de kahvaltı servis edebilirsiniz.

**class Dinner extends Meal   
{  
    public function serve()  
   {   
    return "Akşam yemeğim: Domates soslu makarna";  
   }  
}  
  
class Breakfast extends Meal   
{  
    public function serve()  
    {  
    return "Kahvaltım: Sucuklu yumurta";  
    }  
}  
  
class Feast   
{  
     protected $food;  
  
     public function \_\_construct(Meal $food)  
     {  
         $this->food = $food;  
     }  
}**

Burada üstte interface ile yaptıklarımızı, tekrar yapmadık mı, ne farkı var diye düşünebilirsiniz, ama en baştaki tanımı unutmayın, abstract class kullanarak yaptığımız şey, kahvaltı ve akşam yemeğinin birer öğün olduğunu(**ne olduğunu**) belirterek onları bir tanıma oturtarak kısıtlamakken, interface ile belirttiğimiz şey, kahvaltı ve akşam yemeğinin yenilebilir özelliğe(**yapabildikleri**) sahip olduğunu belirtmek oluyor. Ayrıca, birden fazla interface kullanabilirken, sadece bir tane abstract class'tan yeni class'ınızı inherit edebilirsiniz.   
  
Sonuç olarak:

* Abstract class kullanarak, o class'tan inheritance ile üretilen class'lardaki ortak yöntemleri sıfırdan yazmak zorunda kalmazsınız, ancak üretilen class'ın ne olduğunu belirterek bir nevi kısıtlarsınız.
* Interface ile, o class'ın neler yapabildiğini belirtebilir, birden fazla interface kullanabilir, ancak her bir interface yöntemini farklı class'larda sıfırdan oluşturursunuz.