



NESNE YÖNELİMLİ PROGRAMLAMA TEMELLERİ

www.turkayurkmez.com

C#.NET ile Nesne Yönelimli Programlama dünyasına
adım atın!

Türkay Ürkmez – Evli ve Şişman Adam'ın
maceraları

İçindekiler

GİRİŞ	3
Nesne Yönelimli Programlamaya Giriş	4
Nesne ve Sınıf kavramları	4
Encapsulation	8
Constructor kavramı ve class tasarımı	13
Inheritance	18
Polymorphism	23
Access Modifiers	28
private	28
public	28
protected	29
internal	30
protected internal	30
Event ve Delegate Methods	31
Abstract Class	35
Interface	40

Sevgilim Derya'ya...

GİRİŞ

Uzun bir süredir, www.turkayurkmez.com adresinde yazılarımı sizlerle paylaşıyorum. Bu yazılardan en çok okunanlar ise Nesne Yönelimli Programlama serisinde yer alıyor.

Bu yazıları yazmaktaki amacım, günümüz yazılım geliştirme dünyasında özellikle ticari uygulamaların olmazsa olmazı Nesne Yönelimli Programlama kavramının tüm detaylarını, olabildiğince açık ve basit bir dille anlatmak oldu. Gelen yorumlardan da görebildiğim kadarıyla bu amacıma ulaştığımı düşünüyorum.

Ancak, gizli bir amacımın daha olduğunu belirtmeliyim... O da, insanları güldürerek-eğlendirerek bir şeyler anlatabilmenin mümkün olduğunu göstermekti. Bir şekilde bu hedefi de tutturduğumu tahmin ediyorum.

Şimdi de hedefim, bu yazıları e-kitap formatında yayınlayarak yazılım severlerle buluşturmak.

Ne diyelim? Hepinize eğlenceli okumalar...

Ne mutlu kod yazabilene ☺

Türkay ÜRKMEZ

www.turkayurkmez.com

Nesne Yönelimli Programlama Temelleri

Nesne Yönelimli Programlamaya Giriş

Nesne ve Sınıf kavramları

Merhaba ey dostlar...

Bu günlerde, günlük hayatımızda yaptığımız birçok “rutin” eylemin önüne anlamsız bir espri gibi yapışan bir cümleyle başlamak istiyorum... Evet, ne felsefi bir giriş oldu değil mi? Ama o başlangıcı henüz yapmadığının da farkında mısınız? Peki, tamam uzatmıyorum... İşte geliyor; “2008’in son makalelerini yazıyorum” (bakınız ne kadar kararsız bir cümle ‘son makaleler’ ne yahu?).

Arkadaşlar, yine uzun zamandan beri yazmayı planladığım bir konuyla karşınızdayım; Nesne Yönelimli Programlama. Yani orijinal ismiyle Object Oriented Programming (iş bu makalede bundan sonra OOP olarak anılacaktır 😊). C#.NET program geliştirmeye henüz başlayan, ya da tanıştığı ilk programlama platformu .NET olan herkes, bu yapının tamamıyla OOP uyumlu olduğunu duymuştur. Ama nedir bu OOP denen teknik ve neden ona ihtiyaç duyalım?

Her şeyden önce şunu belirtelim, OOP programlama tekniği, tamamen insanı taklit eder. Bu açıdan, yüzyılın en dinamik ve verimli programlama tekniği olarak düşünülebilir. Peki, bu taklit nasıl oluyor? Gelin bunu şişman adamca örneklerle görelim.

Bildiğiniz gibi; dünya üzerindeki ilk insanlar, kendi ihtiyaçlarını kendileri karşılamak zorundaydılar. Yani; kendi barınaklarını kendileri inşa ediyor, kendi av silahlarını, kap, kacak gibi gündelik yaşamlarında ihtiyaç duydukları tüm malzemeleri kendileri imal ediyorlardı. Yüzyıllar sonra, toplumlar oluştuğça -ya da yerleşik düzene geçildikçe- insanlar kendilerine özel sorumluluk vermeye başka bir deyişle branşlaşmaya başladılar. Yani berber, kasap, bilim insanı, öğretmen, filozof, doktor, marangoz gibi meslekler ortaya çıktı ve böylece ticaret de ilerlemeye başladı.

Günümüz dünyasında, ihtiyaç duyduğumuz şeylerin hemen hepsini, üreticilerinden elde ediyoruz... Kazmayı küreği alıp evimizi yapmıyoruz ya da arka odadaki elektronik aletlerle kendimize bir LCD televizyon üretmiyoruz. Sorumluluğu vermiş olduğumuz üreticiden gidip satın alıyoruz. İşte teknolojinin bu kadar hızlı gelişmesinin temelinde de bu “branşlaşmanın” yatmakta olduğu çok net bir biçimde gözümüze çarpıyor.

Bu noktaya biraz daha yoğunlaşalım... Bir mucit, bir ürün icat ediyor. Daha sonra bu ürünün seri üretimine başlıyor. Zaman geçtikçe, bu ürüne yeni özellikler ekliyor ve ayrıca, ürünün ilk hallerinde de bulunan işlevleri daha verimli hale getiriyor. Bu ürünün ilk halinde bulunan özelliklere “yeniden kullanılabilir” demek pek de yanlış olmaz sanırım. Ayrıca, ürünün yıllar boyunca geçirdiği değişim de, “geliştirilebilirlik” ilkesini kanıtlamıyor mu?

Şimdi OOP tekniği öncesine bir bakalım... Binlerce satır koddan oluşan yazılım ürünleri... Bu ürünleri geliştiren programcılar; her kodu, her seferinde tek tek yazmak zorunda kaldılar. Muhtemelen bu sebepten dolayı, ürünün versiyonları arasında oldukça uzun bir zaman vardı. Çünkü tasarımdaki en ufak bir değişim (mesela yeni bir menü eklenmesi bile), kodun büyük ölçüde değişmesine neden oluyordu. Bu tam olarak zaman kaybıydı.

Artan yazılım talebi, yeni bir kodlama tekniğini yani OOP tekniğini doğurdu. Tıpkı insanların branşlaştığı gibi, kodları da kendi aralarında görevlerine göre ayırarak bir nevi kod fabrikaları oluşturuldu.

İşte bu fabrikaların adı class (sınıf) olarak biliniyor. Haliyle biz, bu class'lardan nesne üretiyor ve o nesneyi, ihtiyacımız olan her yerde kullanabiliyoruz. İşte dediğim gibi OOP kodlama tekniği, tam olarak insanın şu anki yaşamını taklit eden bir yapı.

Peki bir class'ı nasıl tasarlarız? Neler içereceğine nasıl karar veririz? Bu tamamen üreteceğimiz nesneyi ne amaçla kullanacağımıza bağlıdır. Az önce verdiğim örneklerle düşünürseniz, soruyu bir de şöyle sormamız faydalı olur; Bir fabrikayı nasıl kurarız? Fabrikayı kurma aşamasına geldiyseniz eğer, ne üreteceğinize çoktan karar vermişsiniz demektir değil mi? Öyleyse konu class olduğunda da, öncelikle istediğimiz nesnede hangi özelliklerin olması gerektiğini bilmek durumundayız.

Şimdi, bir proje düşünün. Bu proje, bir ayakkabı satıcısının kullanacağı bir program olsun. Kesinlikle bu projede bir ayakkabı nesnesine ihtiyacımız olacak öyle değil mi? Peki bu ayakkabı nesnesinin özellikleri neler olmalı? Gelin bir liste yapalım...

1. Markası
2. Tipi (Bot, spor, çizme vs).
3. Malzemesi (Deri, süet vs).
4. Numarası
5. Rengi
6. Bağcıklı mı değil mi?

Bu özelliklere karar verdim. İşte class'ımız (yani fabrikamız) üretilecek olan nesnemizin bu özelliklerini belirlemek durumunda. Hadi o zaman işe başlayalım ve uygun tiplerle classımızı yazalım:

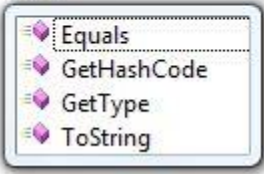
(Aşağıdaki kodlar bir Windows Application projesi içinde yazılmıştır)

Projeye sağ tık, Add New Item, Class diyelim ve aşağıdaki kodları oluşturalım.

```
class Ayakkabi
{
    string markasi;
    string tipi;
    string malzemesi;
    byte numarası;
    string renk;
    bool bagcikliMi;
}
```

Tamam. Şimdi form1'in Form1_Load metodunda, bu class'dan bir nesne üretmeyi deneyelim;

```
private void Form1_Load(object sender, EventArgs e)
{
    Ayakkabi pabuc = new Ayakkabi();
    pabuc.|
}
```



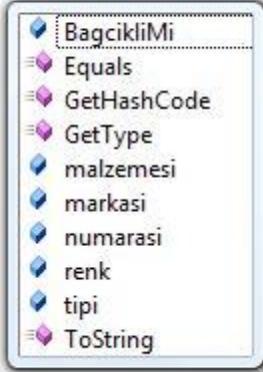
Fakat o da ne? Ayakkabi classımdan ürettiğim pabuc nesnesinde, istediğim özelliklerden hiçbirine erişemiyorum. Çünkü yazdığım özelliklerden hiçbirinin erişim düzenleyicisi (Access Modifier) Public değil! Eğer, class içinde tanımladığınız bir alana class dışından erişmek isterseniz; bu alanı public olarak tanımlamalısınız. Bu durumda, eğer bir proje içinde yazdığınız bir classın başka bir projeden erişilebilmesi için de classın başına public kelimesi getirme zorunluluğumuz ortaya çıkar.

Şimdi, bu değişiklikleri yapalım ve tekrar nesne üretmeye çalışalım:

```
public class Ayakkabi
{
    public string markasi;
    public string tipi;
    public string malzemesi;
    public byte numarası;
    public string renk;
    public bool BagcikliMi;
}
```

Değişiklikler tamam. Peki ya nesnemiz?

```
private void Form1_Load(object sender, EventArgs e)
{
    Ayakkabi pabuc = new Ayakkabi();
    pabuc.
}
```



İşte şimdi, nesnemde aradığım özellikleri görebiliyorum... Ama burada bir sorun var sanki. Bazı özelliklerin nesne tarafında değiştirilebilmesini kesinlikle istemiyorum. Örneğin ben Ayakkabi fabrikamdan 44 numara olarak ürettiğim ayakkabının sonradan 48 numaraya yükseltilmesini istemem. Fakat bütün değerlerim public olarak tanımlandığından bunu şu an yapabiliyorum! Bu hiç de iyi bir şey değil.

Söz, tam bu noktada encapsulation dediğimiz kavrama geliyor. Ama bu, bir sonraki makalemin konusu olacak. Şimdilik burada bırakıyorum.

Bir daha görüşene kadar kendinize iyi davranın ve kendinize birkaç class tasarlayın. Hepinize iyi çalışmalar ve mutlu seneler...

Türkay ÜRKMEZ

Encapsulation

Merhaba sevgili dostlar...

OOP konusundaki eğitsel makalemize encapsulation (kapsülleme) konusuyla devam ediyoruz. Lafı hiç eveleyip gevelemeden konumuza girelim... Bir önceki makalemizde oluşturduğumuz Ayakkabı class'ını hatırlayalım... Bu class'ın içinde bulunan özellikleri public erişim düzenleyicisi (Access Modifiers) ile tanımlamıştım. Ancak makalemin sonunda, böyle yapmanın çok mantıklı olmadığını, bazı özellikleri kısıtlamam (yalnızca okuma ya da yazma gibi) gerekeceğini belirtmiş, hatta “numarası” özelliğini örnek göstermiştim...

Peki bunu nasıl yapmalı? Nasıl yapsak da, class içindeki üyeleri yalnızca okunabilir ya da yazılabilir hale getirsek? Acaba; bu işlem için metodlara başvursak nasıl olur? Deneyelim... Bu deney için, “markasi” özelliğini kullanalım.

Öncelikle, “markasi” özelliğinin değerini class dışına döndürmesi, bir başka deyişle değerinin okunması için public Access Modifier'ı ile bir metod tanımlayacağım. Madem ki metodumu public yaptım, öyleyse “markasi” özelliği “private” olabilir. Ayrıca yazdığımız bu metod doğal olarak geriye string tipinde değer döndürecektir... Öyleyse;

```
private string markasi;  
//”markasi” özelliğinin değerini yalnızca okumak için kullanılacak bir metod:  
public string OkuMarkasi()  
{  
    return markasi;  
}
```

Pekala, şimdi sıra geldi deneyin ikinci adımına, yani “markasi” özelliğine değer atayacak olan metodu yazmaya... Bunun için de yine public bir void metoda ihtiyacım var ve haliyle özelliğe atayacağım değeri bu metodun parametresi olarak alacağım:

```
//”markasi” özelliğine yalnızca değer atamak için kullanılacak bir metod:  
public void YazMarkasi(string deger)  
{  
    markasi = deger;  
}
```

Bakalım işe yarayacak mı? Class'ımızdan bir nesne oluşturup görelim:

```
private void Form1_Load(object sender, EventArgs e)
{
    Ayakkabi pabuc = new Ayakkabi();
    pabuc.YazMarkasi("adidas");
    MessageBox.Show(pabuc.OkuMarkasi());
}
```

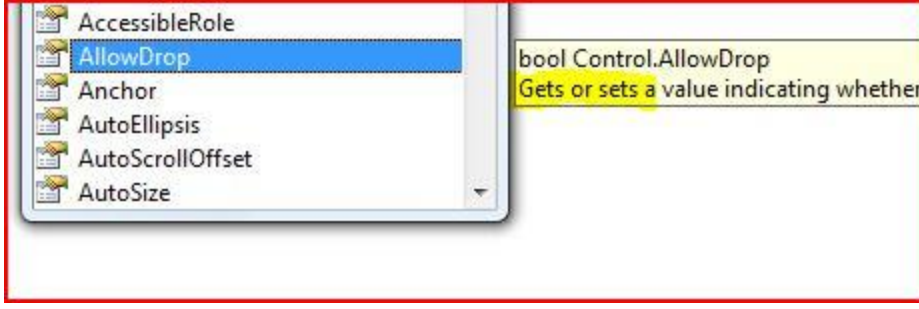
Bu kodu çalıştırdığınızda MessageBox’da “adidas” değerini görebiliriz. Bu, YazMarkasi() metodunun doğru bir biçimde çalışarak “adidas” değerini class içinde tanımlanan “markasi” alanına değer atadığını ve ardından OkuMarkasi() metodunun bu değeri geri döndürdüğünü gösteriyor. Böylece biz, class içinde “markasi” alanıyla ilgili istediğimiz gibi çalışabiliriz. İşte bu yaptığımız, OOP’ nin en temel prensiplerinden biri olan encapsulation işlemidir. Şimdi gelin biraz koddan uzaklaşıp aslında ne yapmak istediğimizi örneklerle açıklayalım...

Gerçek dünyada sık kullandığımız bir örneği ele alalım; televizyon... Standart bir kullanıcı, televizyon izlerken içinde bulunan transducer (anten aracılığı ile aldığı sinyalleri ses ve görüntüye çeviren parça)’i belirli bir frekans aralığında çalışması için ayarlayabilir mi? Cevap, elbette ki hayır. Biz kullanıcılar, televizyon karşısında, ancak kanal değiştirir, sesini ayarlar, renkliğini ya da parlaklığını değiştiririz. Bu bizim için yeterlidir.

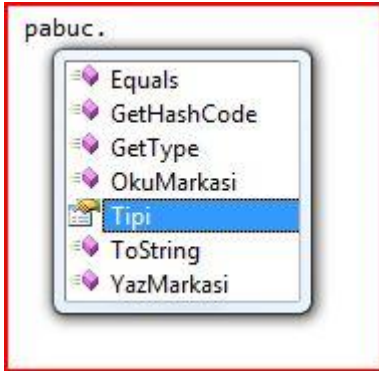
İşte biz de ayakkabi class’ımız için aynı şeyi yaptık... “markasi” alanımızı encapsulate ettik. Peki, bu daha kolay yazılabilir mi? Elbette... Gelin şimdi buna bakalım. İkinci deneğimiz, “tipi” alanı olsun...

```
private string tipi;
//tipi alanının encapsulate edilmesi:
public string Tipi
{
    get { return tipi; }
    set { tipi = value; }
}
```

İşte belki de çok sık gördüğünüz - hatta kullandığınız - ama anlamını bilmediğiniz property (özellik) kavramını görüyorsunuz. Burada gördüğünüz; “get”, aslında az önce yazdığımız OkuMarkasi() metodunun, markasi alanı için yaptığının tam olarak aynısını tipi alanı için yapıyor. Haliyle “set” de YazMarkasi() metodunun görevini üstleniyor. Burada durup size bir şey söylemek istiyorum. Eminim ki birçok nesnenin herhangi bir özelliğinin intellisense sırasında karşınıza çıkan açıklamasını okurken “Get or Set” kelimeleriyle karşılaşmışsınızdır. Aslında bu açıklama bahsi geçen özellikten değer okuyabileceğinizi ya da yazabileceğinizi anlatmaktadır. Bu da özelliğin nasıl yazıldığı hususunda bize bir ipucu vermektedir.



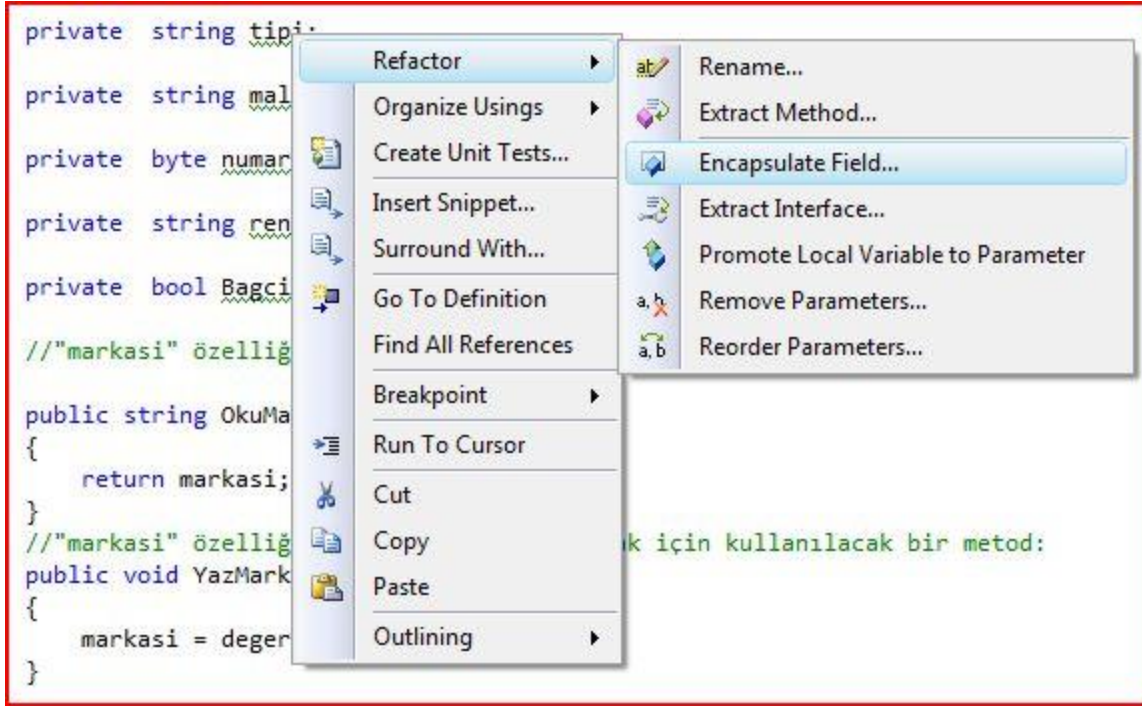
Artık, bizim class’ımızın da “Tipi” adında bir özelliği bulunmaktadır...



Bildiğiniz gibi, Visual Studio’nun biz programcılar için, küçük jestleri, kısayolları mevcuttur. Bir alanın encapsulate edilmesinin de böyle bir yolu var. Şimdi gelin bir de bu yola bakalım, sıradaki alanımız, “numarasi” alanı olsun...

```
private string numarasi;
```

Bu alanımızın üzerine sağ tıklayalım ve ardından açılan menüden “Refactor” seçeneğini, ve alt menüden “Encapsulate Field...” seçeneğini işaretleyelim.



Karşımıza çıkan sihirbaz ekranını tamamladıktan sonra SÜRPRİZ! Property'niz hazır.

```
public byte Numarasi
{
    get { return numarasi; }
    set { numarasi = value; }
}
```

Bu noktada durup, bir önceki makalemde de aktardığım bir durumu hatırlayalım. Ayakkabi classının “Numarasi” özelliğinin dışardan değiştirilmesini istemiyorum. Yani, sadece “okunabilir” olmasını istiyorum. Öyleyse özelliğimizi değiştirelim:

```
public byte Numarasi
{
    get { return numarasi; }
}
```

Böylece, “numarasi” alanı (field) class dışından yalnızca “get” edilebilecek. İşte bu şekilde tanımladığımız özelliklere “read-only” (yalnızca okunabilir) property diyoruz. Hemen söyleyelim; bunun tam tersi de mümkün. Yani yalnızca “set” edilen, “sadece yazılabilir” (write-only) özellik oluşturmak mümkündür.

Bu property kavramındaki son nokta ise, C# 3.0'da bizimle birlikte oluyor. Hiçbir şekilde alan (field) tanımlamadan direkt özellik tanımlama tekniği. Yalnız, Bu tekniği kullanmanız durumunda, yalnızca-okunur ya da yalnızca-yazılabilir özellik oluşturamazsınız. Başka bir deyişle, bu tekniği kullanarak oluşturduğunuz özellik, hem get hem de set yapısını içermek zorundadır. Bunun sebebi, get ya da set kısımlarının kod bloklarının yazılmamasıdır. Öyleyse, üçüncü deneyimiz, “renk” alanı olsun. İşte özelliği geliyor:

```
public string Renk { get; set; }
```

Hemen belirteyim, bu kodun da kısa bir yolu var elbette. “prop” snippet’ini kullanarak (bunu söylemek hoşuma gidiyor; prop tab tab), özelliğinizi oluşturabilirsiniz.

Evet sevgili dostlar.. OOP hakkındaki bu ikinci makalemizin sonuna gelmiş bulunuyoruz. Bir sonraki makalede görüşmek dileğiyle...

Kendinize iyi bakın...

Kodla kalın...

Constructor kavramı ve class tasarımı

Merhaba arkadaşlar...

Yine uzunca bir aradan sonra karşınızdayım. Bu bloğu takip eden ve “birşey yazmıyor bu adam” diye bana sitem eden tüm dostlarımdan da özür diliyorum.

Object Oriented hakkında yazdığımız makale dizimizin bu üçüncü bölümünde constructor kavramından bahsediyor olacağım. Bu konuya girmeden önce bir konuya açıklık getirmek istiyorum. Bazı arkadaşlardan ” çok basit konularda yazıyorsun ” diye eleştiriler geldi. Oysa benim amacım zaten bu. Basit ve anlaşılır makaleler yazmak. Web’ de en sık rastlanan sıkıntılardan biri, aranan konunun ya çok teknik bir dille anlatılması ya da temeline inilmeden nedeni anlatılmadan yüzeysel geçilmesi oluyor. Naçizane, özellikle yeni başlayanlara böyle hizmette bulunmaktır amacım.

Bu kısa mesajı verdikten sonra, object oriented hakkındaki üçüncü makalemize geçebiliriz. Bu makalemize bir soru ile başlayalım; nasıl oluyor da bir class’ tan bir nesne türetebiliyoruz? Bu “türetme” işine kod tarafında baktığımızda en sık karşılaştığımız yöntem, “new” anahtar kelimesi kullanmaktır. Örneğin;

```
Ayakkabi pabuc = new Ayakkabi();
```

Örnekteki “new Ayakkabi()” kısmı, Ayakkabi class’ı içinde bulunan tüm üyeleri, pabuc isimli nesneye aktarır. Biz bu duruma Ayakkabi class’ından nesne türetmek ya da daha teknik bir ifadeyle instance almak diyoruz. Şimdi, işi biraz daha basit almak istiyorum. Aslında, nesne üretmenizi sağlayan şey class’ ınızın içinde bulunan oluşturucu metot (constructor) dediğimiz üyedir.

Constructor, kesinlikle bir metottur. Ama herhangi bir değer döndüren metot değildir. Ya da void metot olarak da düşünülemez. Constructor, yalnızca üyesi bulunduğu class’dan nesne üretimi sırasında çalışacak olan metoddur. Bu durumda şunu söylemek çok yanlış olmayacaktır, classımdan nesne üretilirken; üyelerin varsayılan değerlerini ayarlamak için constructor kullanırım. Bakınız, Windows uygulaması geliştirirken hep gözümüzün önünde bulunan bir constructor’u inceleyelim şimdi...

Bildiğiniz gibi Windows Form’u aslında bir class. Bakalım bu class’ ın bir constructor’u var mı?

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
}
```

İşte orada! **public Form1()** ifadesi ile başlayan üye bizim constructor’ ımız oluyor. Şimdi gelin bu tanımı biraz inceleyelim. Gördüğümüz gibi constructor’ un erişim düzenleyicisi (Access Modifier)

public. Çoğu constructor'un erişim düzenleyicisi public' tir. Bunun sebebi, elbette constructor'a class dışından erişilebilir olması gerektiğidir. (Aksi halde, `Form1 frm = new Form1 ();` diye bir ifade yazamazdık.) Constructor tanımının bir diğer özelliği ise isminin class ismiyle aynı olmasıdır (Form1()). İşte object oriented modellemenin vaz geçilmezi constructor ile böylece tanışmış olduk. Şimdi, aklınıza bir sorunun takıldığını hissediyorum. Şöyle ki, “geçtiğimiz OOP makalelerinde yazdığımız Ayakkabi class' ında constructor yazmadığımız halde pabuc nesnesini türetebilmişik bu nasıl oldu?” Hemen cevaplayalım; Yazdığımız her class'ta varsayılan olarak bir constructor bulunur. Dolayısıyla, siz yalnızca varsayılan ayarları değiştirmek için constructor tanımlarsınız. Belki bu cevap kafanızda bir soru daha oluşmasına neden olacaktır; ‘madem varsayılan olarak bir constructor zaten var o halde neden `Form1` classı bir constructor içeriyor? ‘ bunun da cevabını hemen verelim; yukardaki örneğe dikkat ederseniz `Form1()` constructor'u içerisinde çağırılan `InitializeComponent();` isimli bir metod var. Bu metod, `Form1.Designer.cs` dosyasında bulunan bir metottur ve görevi form üzerinde kullandığımız kontrolleri ve bu kontrollerin özelliklerini oluşturmaktır.

Şunu tekrar belirtelim; constructor, classdan instance alınırken çalışır ve amacı, class üyelerinizin değerlerini ayarlayarak nesne referansına geçirir.

Şimdi gelin, bizim meşhur `Ayakkabi` class'ımıza bir constructor ekleyelim. Hemen kafamızda bir senaryo oluşturalım. Bir ayakkabıcıya gittiniz. Doğru ayakkabıyı alabilmek için, ayakkabıcının bilmesi gereken şey, kaç numara giydiğinizdir öyle değil mi? Sizin elde edeceğiniz ayakkabı nesnesi belirttiğiniz ayak numarasına bağlıdır öncelikle. Madem öyle, işte `Ayakkabi` class'ımızın constructor'u geliyor:

```
public Ayakkabi(byte ayakkabiNo)
{
    numarası = ayakkabiNo;
}
```

Ne mi yaptık? Artık `Ayakkabi` class'ından nesne üretirken sizden `ayakkabiNo` parametresinin değerini alacak. Ardından bu değeri class üyelerinden “numarası” isimli alana (bkz. [OOP2 Encapsulation](#)) aktaracak. Böylece “Numarası” özelliğinin değeri oluşmuş olacak. Haydi... Nesneyi üretirken görelim:

```
private void Form1_Load(object sender, EventArgs e)
{
    Ayakkabi pabuc = new Ayakkabi(44);
}
```

Görüldüğü gibi pabuc nesnemi üretirken artık numara parametresini girme zorunluluğum var. Şunu da belirtelim ki, constructor'un bir nevi metod olduğunu unutmayın. Bu demek oluyor ki, constructor'lar da overload edilebilirler.

Buradan anlaşılıyor ki, constructor yapısı class tasarımı için vazgeçilmez bir kavram. Aslında söz class tasarımına geldiğinde göre, biraz da bu konu üzerinde duralım.

Class tasarımı dediğimizde, o classı oluşturan üyelerin neler olacağı ve nasıl çalışacağı üzerinde kafa patlatmaktan bahsediyoruz. Doğru tasarlanmış bir class, programcıya çok daha yardımcı olacaktır. Tasarıma ilk olarak, özelliklere ve bu özelliklerin tiplerine karar vererek başlarız. Bir özelliğe karar verirken ele aldığımız kriter ilk olarak o özellik ne yapacağıdır. Örneğin bizim class'ımızda malzemesi, markası ve tipi özellikleri arama yapmak için kullanılabilir mi? Evet kullanılabilir. O halde bu özelliklerin tipleri ne olmalıdır? Aklınıza "string" geldiğini hisseder gibiyim. O zaman izninizle biraz kafanızı karıştırmak istiyorum.

Markası özelliğinin tipini "string" yapmaya karar verdiniz. Sonra kodlama aşamasına geçtiniz ve "Adidas", "Nike" ile "Camel" markalarından olmak üzere üç farklı nesne ürettiniz. Buraya kadar tamam. Şimdi de bir MarkayaGoreAra metodu yazdınız ve doğal olarak parametresi de string tipinde. Bu metodu test ediyorsunuz; MarkayaGoreAra("adidas"); yazdığınızda sonuç bulunur mu?

Efendim? Bulunur mu dediniz? Üzgünüm yanlış cevap. Sonuç bulunamaz. Ben "Adidas" markalı bir ayakkabı oluşturdum, "adidas" değil. Unutmayın... case sensitivity öldürür!!

Peki o zaman ne yapacağız? Markası özelliğinin tipi string olmazsa ne olacak? Acaba kendi tipimizi yapsak nasıl olur? Öyle bir tip yazalım ki, sattığımız ayakkabı markalarını üye olarak içersin. Bu cümleyi okuyan bazı arkadaşlarımız enum'dan bahsettiğimi hemen anlayacaklardır. Eğer, belli değerler arasında sınırları olan (Örneğimizde; yalnızca "Adidas", "Nike" ile "Camel" markaları seçilebilecektir.) tipler oluşturmak istiyorsak enum olarak isimlendirdiğimiz yapıları kullanıyoruz. Hemen bunu örneğimiz üzerinde uygulayarak görelim.

Projemize sağ click / Add / Class diyelim ve "Enums" isimli bir fiziksel dosya ekleyelim. Ardından açılan dosyada namespace alanı içindeki tüm kodları temizleyelim... Ops! Bunu neden mi yaptık? Elbette oluşturacağımız enum tipini farklı fiziksel dosyada tutmak için... Şimdi buraya enum tipimizi yazalım:

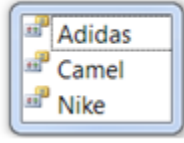
```
public enum MarkaIsimleri
{
    Adidas,
    Nike,
    Camel
}
```

İşte kendi value-type' ımızı oluşturduk. Şimdi Ayakkabı class'ı içinde bulunan Markası özelliğini kendi tipimizle tekrar oluşturalım:

```
public MarkaIsimleri Markasi { get; set; }
```

Böylece artık Markası özelliğine değer aktarırken, belirttiğimiz alanlardan değer seçme özgürlüğüne sahip olacağız. Haydi bunun da fotoğrafını çekelim:


```
pabuc.Markasi= MarkaIsimleri.
```



Ya arkadaş, bir programcı daha ne ister? Resmen MarkaIsimleri tipim beni yönlendiriyor: “kardeşim senin sattığın markalar bunlar, seç birini”. Şimdi yukarıda oluşturduğumuz ayyakkabı arama senaryosuna geri dönelim. Artık MarkayaGoreAra metodunuzun parametresi string değil MarkaIsimleri tipinde olacaktır ve doğal olarak siz bu metodu şöyle çağırabileceksiniz; MarkayaGoreAra(MarkaIsimleri.Adidas); . Bakınız güzel bir class tasarlayarak, kendinize iyilik yaptınız. Şimdi class’ imizdaki diğer bazı özellikler için de birer enum yazalım ve ardından Markasi özelliğinde yaptığımız gibi, söz konusu özellikleri yeniden oluşturalım.

Enums.cs :

```
public enum AyakkabiTipleri
{
    Bot,
    Çizme,
    SporAyakkabi,
    Sandalet
}
public enum MalzemeTipleri
{
    Nubuk,
    Deri,
    Suet
}
```

Ayakkabi.cs:

```
public AyakkabiTipleri AyakkabiTipi { get; set; }
public MalzemeTipleri Malzemesi { get; set; }
```

Böylece, basitçe class modelleme konusuna değinmiş olduk. Modelleme konusu elbette bu kadarla sınırlı kalmıyor. Ama konuyu başka makalelere bırakıyorum.

Burada isterseniz constructor konusuna geri dönelim... En son, constructor yapısının da overload edilebileceğini belirtmiştim. O zaman yapalım!

Bir müşteri ayakkabıcıya girdiğinde, “42 numara spor ayakkabı istiyorum” ya da “42 numara deri çizme istiyorum” diyebilir değil mi? İşte bu parametreleri değerlendirerek Ayakkabi class’inin constructor metodunu overload edelim.

```
public Ayakkabi(byte ayakkabiNo, MarkaIsimleri marka)
{
    numarasi = ayakkabiNo;
    Markasi = marka;
}
public Ayakkabi(byte ayakkabiNo, MarkaIsimleri marka, MalzemeTiplerimalzeme)
{
    numarasi = ayakkabiNo;
    Markasi = marka;
    Malzemesi = malzeme;
}
```

Eh artık bunun da fotoğrafını çekmek bir zaruret:

```
Ayakkabi pabuc = new Ayakkabi(
    ▲ 3 of 3 ▼ Ayakkabi.Ayakkabi (byte ayakkabiNo, MarkaIsimleri marka, MalzemeT
```

İşte dostlarım, constructor metodumuzu kullanarak, nesne üretimini daha verimli bir hale getirmiş bulunuyoruz. Umarım bu konuyu yeterince aydınlatabilmişimdir.

Böylece bir makalemizin de sonuna gelmiş bulunmaktayız sevgili dostlar...

Bir başka makalemizde görüşmek dileğiyle...

C# kardeşliği adına...

Kodla kalın...

Inheritance

Merhaba can dostlarım. Öncelikle sizleri, altı ay kadar beklettiğim için özür diliyorum...

Object Oriented makalelerimize (başlıktan da anladığınız üzere) devam ediyoruz. Bu makalemizde, OOP'nin altın kurallarından olan miras kavramından bahsediyor olacağız. Hemen bir ipucu ile başlayalım... Bu miras, tam olarak genetik miras anlamındadır. Yani ebeveyninden çocuğa geçen göz rengi gibi. Biz de class' lar arasında bu tarz bir bağ kurarak bir class içindeki özellik ve metodları, başka bir class'a aktarabiliyoruz. Teknik olarak, miras veren class'a base (temel) class, miras alan class'a ise derived (türetilmiş) class diyoruz.

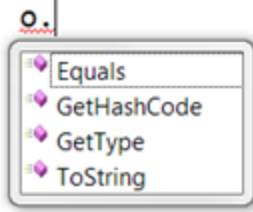
Inheritance, basitçe bir class'dan başka class'lar türetmektir. Peki, bu bizim ne işimize yarar? Neden bir class'dan başka class'lar türetelim ki? Bu sorunun cevabı, tahmin edeceğimiz gibi günlük hayatta yer alıyor sevgili yazılım aşıkları. Şöyle bir düşünün; insanlar, günlük hayatında kullandıkları nesnelere kategorize eder ve geneller. Ne demek istediğimi şöyle anlatayım; öğlen yemeğinizi bitirdiniz ve iş yerinize dönüyorsunuz. O sırada bir arkadaşınızla karşılaştınız. Arkadaşınız, nereden geldiğinizi sorduğunda ona “kuru-pilav yemekten” demek yerine “öğlen yemeğinden” dersiniz. İşte genellediniz! Veya başka bir örnek şöyle olabilir. Bir bilim-kurgu filmi almaya DVD mağazasına girdiniz. Bu mağaza sizin için ürünleri kategorilemiştir. Mağazadan içeri girdiğinizde Müzik ve Film kategorisi arasında seçiminizi yaparsınız önce. Daha sonra Film kategorisi içinden Bilim-Kurgu kategorisini bulur, oradan da aradığınız filme ulaşırsınız. Bir de bilimsel bir örnek verelim, bilim adamlarının bir çoğu, uzman oldukları alanlarda, genelleme ve özelleştirme yaparlar (Bu noktada inheritance konusunu bilen/duyan bazı arkadaşlar, biyologların canlı-hayvan-memeli-maymun kategorisi gibi bir örnek vereceğimi sanıyorlar ama yanılıyorlar). Örneğin dil bilimciler (yaaa demedim mi ben?), dünya dillerini üç kategoriye ayırırlar (Tek heceliler, sondan eklemeli diller ve çekimli diller). Yani genellerler. Ama bir genelleme aynı zamanda özelleştirmedir de, Gidiş yönünüze göre değişir.

Biz de; programlarımızda, bazı nesnelere kategorize etmek isteyebiliriz değil mi? İşte o zaman inheritance'in nimetlerinden faydalanıyoruz. Peki teknik olarak inheritance bize ne katıyor? Gelelim o konuya...

Class'lar arası inheritance uygulamamızın en makul sebebi, “kod maliyetini azaltmak” (yani daha az kod yazmak) olacaktır. Biraz daha açmak gerekirse; birbirleriyle benzer classları ayrı ayrı yazmak yerine, ortak üyeleri belirleyerek bir base class oluşturmak ve geri kalanlarını bu base'den türetmek çok daha pratik bir çözüm olacaktır.

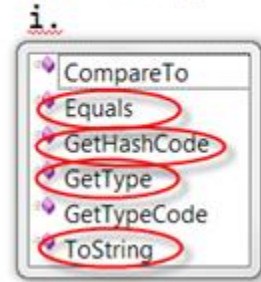
Biliyorsunuz ki .NET'de tüm tipler object'den **türer**. Burada bahsedilen türeme de, inheritance konusuna işaret etmektedir. Aşağıdaki resimde object tipinde bir değişkenin metodlarını görmekteyiz:

```
object o = 0;
```



Peki, inheritance konusunda söylediğim doğruysa ve tüm tipler bu object'den türüyorsa, herhangi bir .NET tipinde de, object metodlarını bulabilmeliyim değil mi?

```
int i = 9;
```



İşte kanıt! Dilerseniz, siz de başka tipler ile bu testi yapabilirsiniz.

Tamam, .NET kütüphanesinde tüm tiplerin, object tipinden türediğini gözlerimizle gördük. Şimdi sıra kendi tiplerimiz arasında miras uygulamaya geldi.

Aslında bunu yapmadan önce türeten - türetilen ilişkisinin (base-derived) nasıl modellenebileceği konusunda biraz konuşmak istiyorum. Kimin base kimin derived olduğuna nasıl karar vereceğiz? Cevap çok basit! Nesnelere arasında, bir tanımlama cümlesi kurabiliyorsanız, aralarında inheritance ilişkisi var demektir. Örneğin:

“Pırasa bir sebzedir” cümlesi, bir tanımlama cümlesidir. Burada iki nesne bulunuyor, “pırasa” ve “sebze”. Demek ki sebze base class, pırasa ise derived class olacaktır. Bir başka bir örnek daha, “Kamyonet bir arabadır”. Dayanamayacağım bir örnek daha vereceğim, “Yazılım eğitmeni bir eğitimidir” (Bu yaklaşım, İngilizce “... is a ...” relationship olarak bilinir. Türkçesi biraz komik duruyor: ... bir 'dir.).

Biz modellememizde, bir restoranın kullanacağı bir yazılımda yer alacak olan class'lar arasında inheritance uygulayacağız. Bir müşteri, restorana gittiğinde masasında hangi çeşitler olur? Genelleyelim; Yemek, içecek ve tatlı. Şimdi bunları özelleştirelim. Yemek; Et yemekleri, ara sıcaklar, sulu yemekler. İçecek; alkollü, alkolsüz. Formülümüzü uygulayalım, “Et yemeği (derived) bir yemektir(base)”.

Ohh... Anlattım rahatladım. Hadi yazalım şunu!

```
public class Yemek
{
    public double Fiyat { get; set; }
    public string Adi { get; set; }

    public string SunumSekli()
    {
        return "Mevsim yeşillikleri, yanında pilav ile";
    }
}
```

İşte yemek class'ım hazır. Şimdi, derived class oluşturalım:

```
public class EtYemekleri : Yemek
{
    public enum EtSunumTipleri
    {
        Izgara,
        Haslama,
        Kavurma,
        Sote
    }
    public EtSunumTipleri EtinSunumu { get; set; }

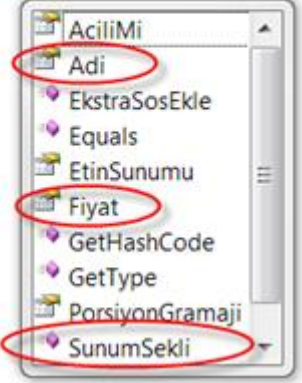
    public int PorsiyonGramaji { get; set; }

    public bool AciliMi { get; set; }

    public void EkstraSosEkle(string sos)
    {
        //sos ekle...
    }
}
```

Şişman adam'dan ancak bu kod beklenirdi zaten! Nasıl miras aldığıma dikkat edin; class isminin yanında ':' işareti ve ardından base class'ımın adı. Peki ne elde ettik? Görelim:

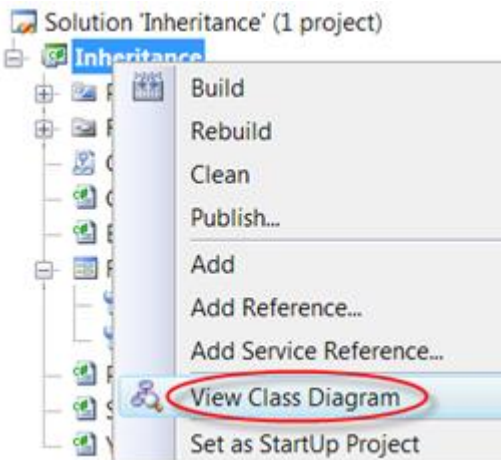
```
EtYemekleri doner = new EtYemekleri ();  
doner.  
}
```



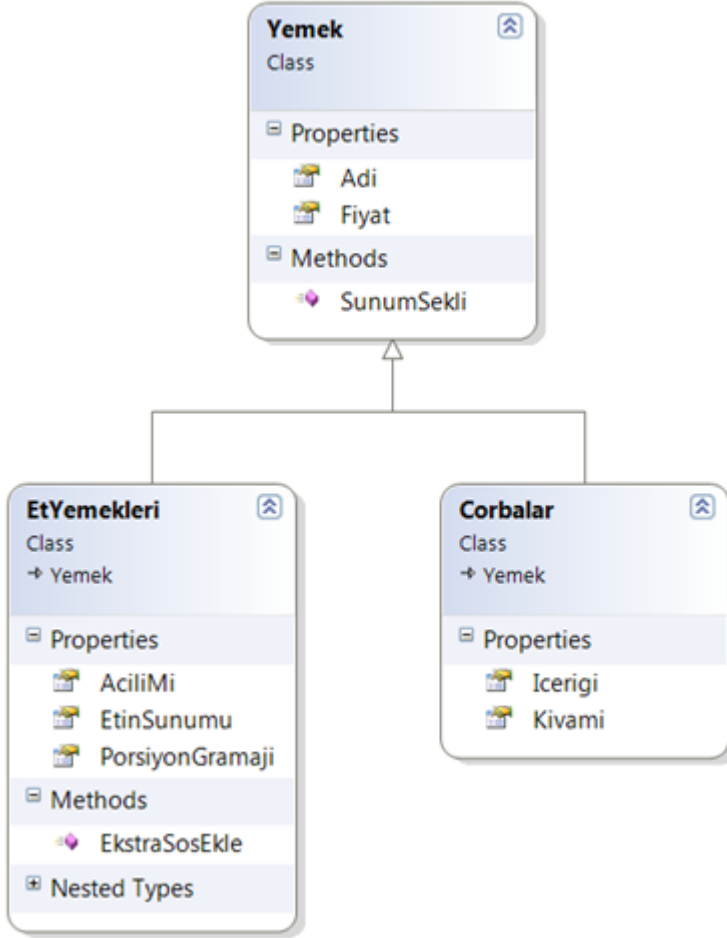
İşte bu! Yemek class'ı içinde yazmış olduğum metod ve özelliklerin tamamı EtYemekleri class'ı na aktarıldı. Böylece ortak olan alanları, genelleme yoluyla ayırabilir ve yemek class'ından istediğim kadar class türetebilirim...

Gördüğümüz gibi dostlarım, nesnelar arası ortak olan metodları, özellikleri ve olayları bir base class altında toplayarak; düzenli bir modelleme oluşturdum. Şimdi kavramı daha da iyi anlayabilmek adına, Windows Application platformundan bir örnek vermek istiyorum. Biliyorsunuz ki, Form nesnesi üzerinde bir çok kontrol (TextBox,Button,Label vs.) kullanabilirim. Bu kontrollerin tümünde varolan bazı özellikler var. Mesela, BackgroundColor, Text, ForeColor bunlardan sadece bir kaçını. Bakın Microsoft ne yapmış? Bu ortak özelliklerin tümünü, Control diye bir class içinde toplamış. Sonra Button, Label, TextBox gibi kontrollerin tümünü bu Control classından türetmiş. İşte, bizim de yaptığımız tam olarak böyle bir modelleme...

Söz modellemeye gelmişken, küçük ve güzel bir ipucu vermek istiyorum. Bazen nesneları görerek tasarlamak işimizi çok kolaylaştırır. Bakın, Visual Studio'da bizim için çok kullanışlı bir class diagram alanı var. Gelin, bir göz atalım:



Solution Explorer penceresinden Projeye sağ tıklatıp açılan menüden View Class Diagram seçeneğini işaretlerseniz, projedeki class'larınızın görsel tasarımını görürsünüz. Ben biraz temizlik yapıp aşağıdaki görüntüyü elde ettim:



Bu şekilde, modellemeyi görerek çalışmak, bazen çok daha eğlenceli/anlaşılır/pratik oluyor sevgili arkadaşlar. Kesinlikle tavsiye edilir.

Şimdi, dikkatinizi bir yere çekmek istiyorum... Eminim ki, birçok okurum, Yemek isimli base class içinde yer alan SunumSekli metoduna dikkat etmişlerdir. İçeriğinde “mevsim yeşillikleri, yanında pilav ile” yazıyor. Peki, menüdeki tüm yemekler aynı şekilde mi sunulacak? Elbette ki hayır. Yani, bazı metodlar ve özellikler; base class'da yazıldıkları gibi kullanılmak zorunda değiller. Peki bu durumu nasıl düzelteceğiz? Her yemeğin bir SunumSekli() vardır ama bunların her birinin içeriği aynı olamaz. Base class'da bulunan Metodun ya da özelliğin, derived class içinde yeniden düzenlenmesi gerekebilir. İşte bu da, polymorphism (çok biçimlilik) konusuna giriyor. Onu da bir sonraki makaleye bırakalım.

Kendinize iyi bakın. Bol nesne modellemeli günler diliyorum.

Polymorphism

Merhaba sevgili yazılım âşıkları! Nesne yönelimli programlama konulu eğitsemakale serimize devam ediyoruz. Bu yazımda ele alacağım konu, “çok biçimlilik” olarak türkçeye çevrilebilecek olan ve yine, nesne yönelimli programlama kavramının temellerinden olan polymorphism olacak.

İnanın böyle ciddi cümleler kurmak beni biraz sıkıyor dostlarım. O nedenle, izin verirsiniz rahat rahat, geyik yapa yapa size bu kavramı sunmaya çalışacağım (Gerçi, izin vermeseniz de böyle olacak ama neyse).

Sevgili dostlar, çok biçimlilik kavramını iyi anlayabilmek için miras kavramı hakkında kafamızda hiçbir soru işareti kalmamalı. Eğer o konuda sorun varsa sizi oraya [alayım](#). Şimdi çok biçimlilik kavramını anlamak üzere yine bir nesne modelleyelim isterseniz. Bu kez bir e-ticaret sitesinde, ürün satışı yapmak için kolları sıvıyoruz. Bu e-ticaret sitemiz, sanal market olsun. Yani hem gıda hem de elektronik ürünleri satabilecek bir portal. Site kullanıcısı, satın almak istediği ürünü sepete ekler. Siparişi vermeden önce, sepetteki tüm ürünlerin fiyatları toplanır ve ödenmesi gereken miktar belirlenir. Ha unutmadan, doğal olarak bu ürünlerin KDV oranları da farklı (haydi hayırlısı) olacak ve elbette fiyat ona göre hesaplanacak.

Haydi bakalım. Öncelikle dilerseñiz Urun isminde bir sınıf oluşturalım. Site üzerinden satışını gerçekleştirdiğimiz tüm ürünlerin ortak alanları burada olsun.

```
public class Urun
{
    public string UrunAdi { get; set; }
    public double Fiyat { get; set; }
    public double KDVUygula()
    {
        return Fiyat * 1.08;
    }

    public Urun()
    { }
    public Urun(string ad,double fiyat)
    {
        UrunAdi = ad;
        Fiyat = fiyat;
    }
}
```

Sınıf üyelerim gayet basit gördüğünüz gibi. Buradaki amacım, size çok biçimlilik kavramını anlatabilmek olduğu için, diğer sınıflarımın da basit olmasına özen göstereceğim (bazen böyle açıklamaları neden yaptığımı hiç anlayamıyorum. Boşverin takılmayın siz.). Bu sınıfta KDVUygula metoduna dikkat!! Varsayılan olarak, ürün fiyatına %8 oranında KDV uyguluyorum.

Peki gelelim diğer sınıflara. Urun sınıfından Tekstil, CepTelefonu ve kuru gıda ürünlerini temsil edecek sınıflarımı türetiyorum. Evet.. e dedim ya polymorphism'i anlamak için miras kavramını bilmek gerekir diye... Her neyse.

Tekstil:

```
public class Tekstil : Urun
{
    public string KumasTuru { get; set; }
    public int Beden { get; set; }
    public string UreticiFirma { get; set; }

    public Tekstil(string ad, double fiyat,string kumasTuru,int beden)
    {
        UrunAdi = ad;
        Fiyat = fiyat;
        KumasTuru = kumasTuru;
        Beden = beden;
    }
}
```

Cep telefonu:

```
public class CepTelefonu : Urun
{
    public string Ozellikler { get; set; }
    public string Marka { get; set; }

    public CepTelefonu(string ad, double fiyat, string marka)
    {
        UrunAdi = ad;
        Fiyat = fiyat;
        Marka = marka;
    }
}
```

Ve... Ekmek:

```
public class Ekmek:Urun
{
    public string EkmekTuru { get; set; }
    public int Gramaj { get; set; }

    public Ekmek(string urunAdi, double fiyat, string ekmekTuru, int gramaj)
    {
        UrunAdi = urunAdi;
        Fiyat = fiyat;
        EkmekTuru = ekmekTuru;
        Gramaj = gramaj;
    }
}
```

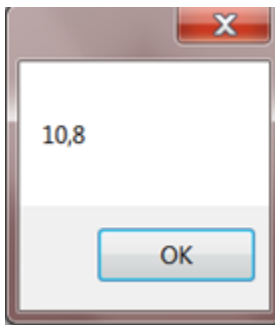
Peki, şimdi de Sepet sınıfımızı yazalım. Bu sınıf, ürünlerimizi taşıyacak ve “ödenmesi gereken tutar” ı bizim için hesaplayacak.

```
public class Sepet
{
    private List<Urun> urunler = new List<Urun>();
    public double ToplamTutar()
    {
        double toplamFiyat = 0;
        foreach (Urun item in urunler)
        {
            toplamFiyat += item.KDVUygula();
        }
        return toplamFiyat;
    }
    public void Ekle(Urun yeniUrun)
    {
        urunler.Add(yeniUrun);
    }
}
```

Şimdi, bu makaleyi yazarken, doğru KDV oranlarını bulmak için [bu siteye](#) baktım durdum. Ekmeğin KDV oranı %1, cep telefonu %18, tekstil ürünlerinin ise %8. Şöyle kolay hesaplayacağımız örnek fiyatlar vererek, elde etmemiz gereken KDV dahil fiyatları bir görelim. Eğer ekmeğin kdv hariç fiyatı 10 kuruş olsaydı, KDV dahil fiyatı $10 \times 1.01 = 10,1$ kuruş olacaktı. Peki bakalım bizim şu sınıflar, bu fiyatı hesaplayabilecek mi? Görelim:

```
Sepet sepet = new Sepet();
Ekmek ekmek = new Ekmek("Uno", 10, "kepekli", 100);
sepet.Ekle(ekmek);
MessageBox.Show(sepet.ToplamTutar().ToString());
```

..ve sonuç:



Hmm... İşte sorun! Ama bu sorunu, eminim ki teşhis etmişsinizdir. İçinizden diyorsunuz ki, “eh be şişman adam, Urun sınıfındaki KDVUygula metodu varsayılan olarak %8 vergi uygularsa böyle olur”. Peki tamam. Nasıl çözeceğiz o zaman bu sorunu? KDVUygula metodunu Urun sınıfından kaldırmamı beklemeyin. Çünkü tüm ürünlerimde (tekstil, cep telefonu ve ekmeği) bu metod var. O zaman şöyle diyebilir miyiz, evet tüm ürünlerde KDVUygula olmalı ama, hepsinde FARKLI çalışmalı! Evet işte şu

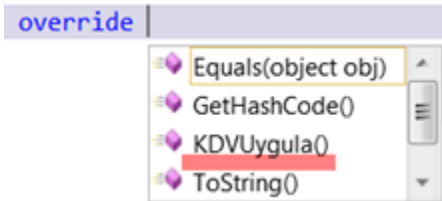
an da çok biçimli olması gereken bir KDVUygula metoduna ihtiyaç duyuyorsunuz! Hazır bu ihtiyacı duymuşken, şu polymorphism'in tanımını bir yapalım. Temel sınıftan, türetilmiş sınıfa kalıtım yoluyla aktarılan, fakat türetilmiş sınıfta farklı bir şekilde kullanılabilmesine izin verilen sınıf üyesi, çok biçimli bir üyedir. Buraya kadar bu yazıyı okumuşsanız, hala sıkılmamışsınız demektir. Öyleyse, bir şişman adam örneği hakettiniz!

Çok değil bundan bir elli atmış yıl önce, babalarımız dedelerimiz bizden çok daha doğal besleniyorlardı öyle değil mi? Fakat biz, hazır yiyeceklerle biraz daha haşır neşiriz. Peki bu beslenme dediğimiz olay canlıların tümünden bize kalıtımla aktarılan birşey değil mi? Evet ama kalıtımdan kalıtıma bu beslenme şekli oldukça değişmiş! İşte size, atalarımızdan aldığımız çok biçimli bir metod... Beslenme metodu!

Peki, bu çok biçimliliği kendi sınıfımıza uygulama zamanı geldi. Bakın Urun sınıfındaki KDVUygula() metoduma ne yapıyorum:

```
public virtual double KDVUygula()
{
    return Fiyat * 1.08;
}
```

“virtual” anahtar kelimesini ekleyerek metodumun çok biçimli bir metod olduğunu gösterdim. Şimdi de, bu metodun yapısını, türemiş sınıflarımda nasıl değiştireceğime bakalım. Hemen Ekmek sınıfına gidelim ve override kelime sini yazdıktan sonra neler geldiğine bir bakalım:



İşte! KDVUygula orada! Demek ki sadece virtual imzalı sınıf üyeleri override edilebilirler (Bu arada override; çiğnemek,ezmek anlamına gelir). Biz KDVUygula metodunu seçerek yola devam edelim ve metodun gövde kodunu değiştirelim:

```
public override double KDVUygula()
{
    return Fiyat * 1.01;
}
```

Şimdi az önce yanlış hesapladığımı gördüğümüz kodu tekrar çalıştırıyorum. Bakalım sonuç ne olacak:



İşte bu! Çok biçimlilik bir kez daha günü kurtardı. Teşekkürler nesne yönelimli programlama, teşekkürler çok biçimlilik.

Buraya kadar, aklınızda bir soru işareti kalmış olabilir.. Yukarıda override kelimesini “ezmek” olarak tanımlamıştım. Peki kim neyi eziyor? Biraz buraya odaklanalım. Unutmayın ki, bu örnekte tüm ürünlerimde KDV uygulanması gerektiğinden dolayı Urun sınıfına KDVUygula metodunu ekledim. Sepet nesnesinde ödenecek tutarı hesaplarken de urun nesnesinin KDVUygula metodunu kullandım. Yine örnekte sepet.Ekle(Urun urun) metodunu kullanarak bir Ekmek nesnesini, sepete ekledim. Aradaki miras ilişkisi sayesinde bunu yapabildim. Sonuç olarak, Ekmek sınıfı içinde yer alan KDVUygula metodu, Urun içindeki KDVHesapla metodunu geçersiz kıldı yani ezdi. Böylece, ekmek için %1 KDV uygulanması gerektiğini belirtebildim.

İşte sevgili dostlarım. Nesne Yönelimli Programlama dünyasının bu karışıkmiş gibi görünen polymorphism konusunun da aslında ne kadar basit olduğunu göstermiş bulunuyoruz. Unutmayın; karmaşık olan şeyler basit geldiğinde, kendimizi geliştirmişiz demektir.

Sağlıcakla kalın.

Access Modifiers

Merhaba dostlarım...

Nesne yönelimli programlama serimize devam ediyoruz. Bu yazımda size, erişim düzenleyicilerden (Access modifiers) bahsedeceğim.

Aslına bakarsanız, bu yazıyı yazarken içimden bir ses; “şişman adam bak bu makale çok kısa olacak, hem zaten her yerden öğrenilebilecek bir konu, ayrıca çok kolay niye yazıyorsun ki?” diye soruyor. Benim düşüncem ise şöyle yanıt veriyor; “evet basit bir konu olabilir ama, nesne yönelimli programlamanın en temel noktalarından biri. Ayrıca, ben bu nesne yönelimli programlama serisinde, kavramın tüm ayrıntılarını anlatmak istiyorum”. Eğer, şu an bu yazıyı okuyorsanız, bu söz düellosunu ben kazanmışım demektir.

İşte ben, böyle şizofrenik bir durumdayken (gecenin 02.53'ünde normal tabii), farkında olmadan makaleyi biraz daha uzatmış oluyorum (kim demiş, makale ciddi olmalı diye?)...

Öncelikle erişim düzenleyici dediğimiz anahtar kelimelerin ne işe yaradığı hakkında genel bir tanım yapalım. Bir tipin kendisine veya o tipe ait üyelere (metod, özellik ya da olay) nasıl erişileceğini daha doğrusu, nereden (hangi kod bloğundan) erişilebileceğini belirleyen kelimelere “erişim düzenleyiciler” diyoruz. İşte şimdi bu kelimeleri, tek tek anlatmaya sıra geldi.

private

En minik erişim düzenleyicisidir. Tipin üyelerinde kullanılır. Üyenin (bu üye metod veya global değişken -alan- olabilir), yalnızca o tipin içerisinden erişilmesine izin verir. Yani üyenin o tipe “özel” olmasını sağlar. Adı üzerinde! Başka bir deyişle, private anahtar kelimesi ile tanımlanmış bir üyeye, tip dışından ulaşamazsınız. Ha unutmadan: private üyeler miras yoluyla türetilmiş sınıflara aktarılamazlar. Bir şey daha, eğer bir üyenin önüne hiçbir erişim düzenleyici anahtar kelimesi belirtmezseniz, o üye derleyici tarafından private olarak algılanır. Yani private, üyeler için varsayılan erişim düzenleyicisidir.

public

En genel ve sınırsız erişim düzenleyicisidir. Hem tip için hem de, tip üyeleri için kullanılabilir. Elbette her ikisi için de farklı anlamları vardır. Önce tipler için konuşalım. Örneğin, bir sınıfın erişim düzenleyicisi public ise bu, o sınıfın, bulunduğu assembly dışından (referans olarak alındığı başka bir assembly'den) her türlü erişilebileceği anlamına gelir. Peki, tipin üyeleri public ise ne olur? Yine adı üzerinde. public, “genel” demektir. Yani, o üyeye her yerden erişilebilir. Doğal olarak kalıtım yolu ile türetilmiş sınıflara aktarılırlar.

Şimdiye dek anlattığım erişim düzenleyicileri hakkında zaten az çok birşeyler duymuşsunuzdur. Sınıflarımda bu konuyu anlatırken, öğrencilerimin yüz ifadelerinin en çok değiştiği erişim düzenleyicilere geldi sıra (bu ifade değişimi genellikle şöyle yorumlanır; “ee nerede kullanacağız lan bunu?” ya da “anlamadım ki!”).

protected

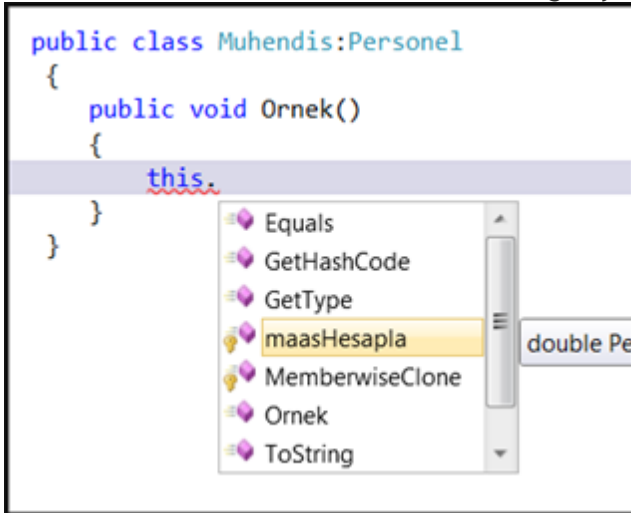
“Korunan” anlamına geldiği aşikar. Peki ama kim kimden korunuyor (“şşt!! Değiştirmeyin yüz ifadenizi hemen”)? Peki, sadece tipin üyelerinde (metod veya alan) kullanılabilir (aynı private gibi). Bu üyeye yalnızca ait olduğu tip içinden ulaşılabilir (aa! Bu da private gibi). E o zaman, private'den farkı nedir? İşte tek fark: protected olarak tanımlanmış alan ya da metodlar, miras olarak aktarılabilirler. Yani, “bu üyeye kesinlikle dışardan ulaşılmasın, ama miras bırakılabilirsin” diyorsanız, o üye protected olmalı.

Peki ne zaman, nasıl bir modelde buna ihtiyaç duyarsınız? Şöyle hayal edelim; “personel” isminde bir sınıfınız var ve bu sınıfın da “maasHesapla” isminde bir metodu var. Metod basitçe, mesai saati ve saat ücretlerini alıp çarpıyor. Şimdi, bu sınıfın temel sınıf olmasına karar verdiniz. Muhendis ve Muhasebeci sınıflarını buradan türeteceksiniz. E onlarda da maasHesapla metodu olacak (yani kalıtım yoluyla aktarılacak - private olamaz). Ama siz, bu sınıfların dışından (yani sınıfın örneğinden) maasHesapla metoduna erişmek istemiyorsunuz (o zaman public olma şansı yok). İşte bu durumda maasHesapla metodu, protected olmalı.

Örnekle bakalım:

```
public class Personel
{
    protected double maasHesapla(int mesaiSaati, double saatUcreti )
    {
        return mesaiSaati * saatUcreti;
    }
}
```

Yukarıda mevzu bahis Personel isimli sınıfımı görüyorsunuz.



Burada ise, türetilmiş sınıfım olan Muhendis içinde, “Ornek” isimli metodda, this anahtar kelimesiyle; Personel sınıfından kalıtım aldığım “maasHesapla” metoduna ulaşabiliyorum. Burada bir hatırlatma yapalım; this anahtar sözcüğü, yalnızca o sınıfın (örneğinizde Muhendis sınıfı) üyelerine ulaşmak için kullanılır. Böylece, kabaca “miras bırakılabilen, fakat dışarıdan ulaşılabilen alanlar protected erişim düzenleyicisi ile belirlenir” diyebilir miyiz? Bence deriz güzel de olur!

internal

“Dahili” anlamına gelmektedir. Yalnızca bulunduğu assembly'den erişilebilir. Burada assembly ifadesinden kasıt, projenin kendisidir. Yani, bir kütüphane (.dll) oluşturuyorsanız, internal bir sınıfa sadece o kütüphaneden ulaşabilirsiniz. Bu erişim düzenleyicisi, sınıf üyelerinde de kullanılabilir. Onda da etkisi aynıdır. Bir sınıfın erişim düzenleyicisi belirtilmezse, varsayılan olarak internal kabul edilir.

protected internal

Yalnızca sınıf üyelerine uygulanır. Kelimelerin arasına “ya da” konulunca anlaşılması daha kolay olacaktır. Erişim yapılan yere göre “internal” ya da “protected” davranır. Doğal olarak assembly dışından erişilmeye çalışıldığında internal, aynı assembly'den erişilmeye çalışıldığında ise protected davranır.

İşte sevgili dostlarım. Bu noktadan bakıldığında bile, bu nesne yönelimli programlama kavramında, “nesne modellemenin” ne kadar büyük bir okyanus olduğunu gösteriyor. Bu okyanus ki, bir sorunun sonsuz çözümünü ihtiva ediyor.

Bu konuda bir yazımızın daha sonuna geliyor ve yazılım isimli puzzle'ın bir parçasını daha yerine koymuş oluyoruz.

Sağlıcakla kalın.

Event ve Delegate Methods

Merhaba ey yazılıma gönül verenler... Nesne yönelimli programlama yazı dizimize kaldığımız yerden devam ediyoruz. Bu yazımda sizlere, OOP'nin anlaşılması belki de en zor konularından biri olan event (olay) ve delegate (delege) metodlardan bahsedeceğim.

Öncelikle, sınıf üyelerinden biri (diğerlerini biliyorsunuz zaten -özellik ve metodlar) olan olayları yazmaya neden ihtiyaç duyarız buradan başlayalım... Olay üyesinin tanımını şöyle yapalım; “nesnenin başına gelebilecek herşey bir olaydır”. Örneğin, Button nesnesinin tıklanması (click), sanırım bu nesnenin başına en çok gelecek şeylerden biridir. Siz bir programcı olarak, button nesnesinin click olayını veya bir başka olayı, “canınız isterse” ya da daha doğrusu, “işinize yarayacaksa” kullanırsınız öyle değil mi?

Tam olarak burada, bir şişman adam örneği gelsin... Normal bir günde yağmur yağması bir olaydır sevgili dostlarım. Siz, bu olaya nasıl yanıt verirsiniz? Bazılarımız şemsiyesini açıp ıslanmaktan korunurken, bazılarımız ise romantik bir yürüyüş yapar. O zaman şöyle diyebilir miyiz; “her olay bir metodu çağırır”. Bu metodun ne yapacağına ise siz karar verirsiniz. Ama bir dakika! Bu metodun, bazı parametrelere ihtiyacı yok mu sizce? Örneğin; yağmurun şiddeti, o an nerede olduğunuz gibi bazı değerleri dikkate almak istemez misiniz?

O halde, olay herhangi bir metodu değil; önceden belirlenmiş, bu olayı yakalamak için yetkilendirilmiş (delege edilmiş) bir kalıba uygun bir metodu çağıracaktır.

Sanırım, hala “neden olay yazarım” sorusuna net bir yanıt vermedim. Şöyle ifade edelim; yazdığınız bir nesnenin, gerçekleştireceği bir eylemin sonucunda, başka bir metodu tetiklemesini ve bu metodun sizin (ya da nesnenizi kullanan başka bir programcının) o anda istediğiniz işlemi yapmasını isteyebilirsiniz. İşte bu isteğinizi gerçekleştirmek için, o nesneye olay yazarsınız...

Yukarıdaki soyut cümleyi, bir örnekle açıklayalım. Veritabanı ile çalışan ve içerisinde kabaca ekleme, silme ve güncelleme işlemlerini yaptığımız bir nesne yarattığımızı ele alalım. Bu nesne; ekleme işlemini gerçekleştirdiğinde siz, kullanıcıyı bilgilendirmek, veritabanında başka bir sorgu çalıştırmak, yapılan işlemi bir dosyaya kaydetmek veya başka herhangi birşey yapmak istiyorsunuz. O halde bir, “eklendi” olayı yazarsınız, bu olayın bir metodu çağırmasını sağlar ve o metoda canınız ne yaptırmak istiyorsa yaptırırsınız.

Şimdi, yukarıdaki yağmur örneğinde belirttiğim, “delege edilmiş bir kalıp” a bir kez daha dikkat çekmek istiyorum. Az önce ihtiyaç duyduğumuz “eklendi” olayının çağıracağı metot, sizin belirlediğiniz bir kalıba uygun olmalı. Çünkü, çalışacak olan metot, yine sizin belirlediğiniz parametrelere ihtiyaç duyabilir. Mesela, hangi datayı eklediniz, saat kaçta eklediniz, eklenen datanın oluşan son id değeri nedir gibi... Madem bunları istiyorsunuz, o halde bunu yapabilmeyin sırrını vermemin zamanı geldi; bir olay yazmak istiyorsanız, önce o olayın çağıracağı metodun kalıbını, yani delege metodunu yazmak zorundasınız.

Peki, mademki delege metodumuz bazı değerlere ihtiyaç duyuyor (yukarıda belirttiğim ihtiyaçlar), o zaman bu değerleri tutan bir sınıf tanımlarız ve bu sınıfı da delege metodun parametresi olarak kullanırız. Böylece bu kalıba uygun metot da bu parametreyi içermek zorunda kalır. İşte bu amaç için yazılan sınıflar, olay argümanları anlamına gelen EventArgs olarak adlandırılır ve aynı isimli sınıftan kalıtım yoluyla türetilirler. Yine geleneksel olarak, bu argümanlara “e” ismi verilir. Birazdan kod örneklerine geçeceğim ama ondan önce, birşeyin altını çizmek istiyorum, delegate methods, .NET Framework'ün en önemli yapı taşlarından birisidir ve sadece olay mimarisi için kullanılmaz, çok farklı konseptlerde de karşımıza çıkabilir ama gelin bunu başka bir makaleye bırakalım...

Bir delegate metod, kesinlikle bir sınıf üyesi olmak zorunda değildir. Bu nedenle genellikle namespace alanında tanımlanır. Ancak ben -konu event olduğu için- classın içinde tanımlayacağım. Senaryomuz, az önceki örnekte olduğu gibi, veritabanına kayıt ekleme üzerine olsun... Buyrun o zaman;

Önce, delege metodumuzun kullanacağı EventArgs sınıfımızı yazalım.

```
public class UrunEklendiEventArgs : EventArgs
{
    public DateTime EklenmeAni { get; set; }

    public string EklenenUrunAdi { get; set; }

    public UrunEklendiEventArgs()
    {
        EklenmeAni = DateTime.Now;
    }
    public UrunEklendiEventArgs(string urunAdi)
    {
        EklenenUrunAdi = urunAdi;
    }
}
```

Konunun özünden uzaklaşmamak için oldukça sade bir EventArgs sınıfı oluşturdum. Şimdi, olay yazacağımız sınıfımıza geçelim.

```
public class UrunIslemleri
{
    /*geleneksel olarak, delege metodumun adı olay yakalayıcısı anlamına
    * gelen "EventHandler" kelimesiyle bitiyor:*/
    public delegate void UrunEklendiEventHandler(UrunEklendiEventArgs e);
    /*ve işte olayımı yazıyorum:
    *dikkat ederseniz, olay ve delege metod, et ve tırnak gibi..
    *Aşağıdaki kodun türkçesi şu,
    *"UrunEklendi olayı, UrunEklendiEventHandler kalıbına uyan bir metodu
    *tetikler"*/
    public event UrunEklendiEventHandler UrunEklendi;
}
```

İşte o kadar dil döküp, anlattığımız şeyin koda yansması bu kadar 😊 İşin esprisi bu. Peki, ama bu olay nasıl ve ne zaman fırlayacak? Bunun en net cevabı şudur, “siz istediğiniz zaman ve istediğiniz şekilde”. Buradaki sevimli olayımız, ürün ekleme işlemi bittiğinde fırlayacak. Bunu birazdan yazacağım ama burada çok önemli bir noktanın üzerinde durmak istiyorum. Bir soru sorarak duruma parmak basalım; “yakalamadığınız bir olay tetiklenir mi?” Cevap hayır. Yani, button nesnesinin Click olayını yakalamazsanız, o olay tetiklenmez. Bunun sebebi şu; eğer bir olay, delegate metoda (kalıba) uygun bir metod ile ilişkilendirilmezse o olay fırlatılmaz. Bunu kontrol edebilmek için biz de fırlatıcı bir metod yazarız. Bu metod, elbette UrunIslemleri sınıfına özel olmalıdır.

```
//yine geleneksel olarak fırlatıcı metodun adı "On" ile başlıyor:  
private void OnUrunEklendi(UrunEklendiEventArgs e)  
{  
    if (UrunEklendi!=null )  
    {  
        UrunEklendi(e);  
    }  
}
```

Artık fırlatıcımız da hazır olduğuna göre, fırlatalım gitsin...

```
public void UrunEkle(string urunAdi)  
{  
    //ADO.NET kodlarınızı yazdığınızı varsayın...  
    //(üşendiğimden değil, konudan uzaklaşmayalım diye)  
    UrunEklendiEventArgs e = new UrunEklendiEventArgs(urunAdi);  
    OnUrunEklendi(e);  
}
```

İşte şimdi, yazılarımın vazgeçilmez anı, kodu test etme zamanı... Bunu yaparken form uygulaması kullandım sevgili dostlarım.

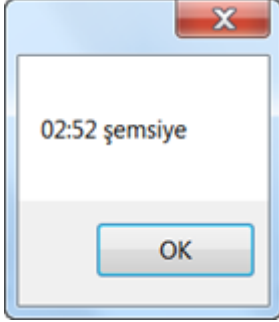
```
private void Form1_Load(object sender, EventArgs e)  
{  
    UrunIslemleri islem = new UrunIslemleri();  
    islem.UrunEklendi += new UrunIslemleri.UrunEklendiEventHandler(islem_UrunEklendi);  
}  
  
void islem_UrunEklendi(UrunEklendiEventArgs e)  
{  
    MessageBox.Show(e.EklenmeAni.ToShortTimeString() + " " + e.EklenenUrunAdi);  
}
```

Bakın “islem” nesnesinin UrunEklendi olayı ile, “islem_UrunEklendi” metodunu, “+=” operatörü kullanarak birleştiriyorum. Bunu yaparken, “UrunEklendiEventHandler” kalıbı ile “islem_UrunEklendi” metodunun uyumlu olması gerektiğine de dikkat ediniz.

Hemen bir ipucu vererek sizi rahatlatalım... Bir nesnenin olayına eriştikten sonra += operatörünü yazdıktan sonra iki kez “tab” tuşuna basarsanız, bu kodu sizin yerinize yazacak ve hatta metodu da oluşturarak, size büyük bir kolaylık sağlayacaktır...

Ve şimdi ürünümüzü ekleyip sonucuna bakalım:

```
islem.UrunEkle("şemsiye");
```



İşte sevgili dostlarım. Nesne Yönelimli Programlamanın önemli parçalarından birinin daha sonuna geldik... Umarım bir nebze faydam olabilmıştır.

Görüşmek üzere

Abstract Class

Merhaba yazılım dostları. Nesne Yönelimli Programlama makale serisine (keşke tüm hızıyla diyebilseydim ama, kaplumbağa yavaşlığıyla) devam ediyoruz.

Tecrübeyle sabittir ki; nesne yönelimli programlama tekniği öğrenilirken zihinde canlandırılması en güç kavramlardan biri, bu abstraction kavramıdır. Ama biliyorsunuz; yazılımda varolan herşey, günlük hayattan alıntıdır. O zaman abstract class kavramına da bu açıdan yaklaşacağız demektir. Hemen bu noktada basit bir örnekle ortamı şenlendirelim; bir markete girdiniz ve kasiyere yaklaştınız ve şöyle dediniz: “merhaba, bir ürün alabilir miyim lütfen?”. O anda kasadaki kişinin yüz halini gözünüzün önünde canlandırın lütfen. Sonra da size ne tepki vereceğini... “Hö? Ne ürün?” Bu tepki çok normal, çünkü siz bir market için gayet ‘soyut’ bir şey istediniz. Ürün, markette satılan malzemelerin tümüne verilen genel addır ancak elinizde tutacağınız sabit bir nesne değildir. En nihayetinde siz, gazete ürünü, sakız ürünü alabilirsiniz fakat “ürün” isminde fiziksel bir nesnenin varlığı söz konusu olamaz. İşte size, abstract (soyut) sınıf kavramını anlatmış bulunuyorum; kendisinden nesne üretilmeyecek, yalnızca miras verebilecek sınıflar; abstract sınıflardır.

İşte o bomba soruyu duyar gibiyim şimdi; “iyi de ben nerede ihtiyaç duyarım soyut sınıfa?” Beyler bayanlar, aşkı yazılım olanlar; benim amacım da bunu anlatmak zaten!

Şöyle düşünün sevgili dostlarım; bir sınıf yazıyorsunuz ve bu sınıfın özelliklerinden (property) biri, başka bir nesne döndürüyor. Bu nesne de, davranışsal olarak bir kaç türe ayrılabilir. Bu durumda hepsini ortak bir noktada buluşturmanız şart demektir.

Biliyorum biliyorum. Bu cümle biraz karışık geldi. En iyisi örneklendirmek. Bir müzik simülatörü yazmaya karar verdiniz. Bu uygulamada kullanıcı, çeşitli müzik enstrumanları kullanan müzisyenler oluşturularak bir nevi orkestra yaratır ve belirli müzikleri çalmalarını sağlar.

Böyle bir uygulamanın en önemli sınıfı, elbette Müzisyen sınıfı olacaktır. Kodlayalım öyleyse (bu örnekte modellemenin ön plana çıkması için Console Application kullandım):

```
public class Müzisyen
{
    public string Ad { get; set; }
    public string Soyad { get; set; }
    public MüzikAleti CaldigiEnstruman { get; set; }
}
```

Dikkat edecek olursanız, Müzisyen sınıfımızda bulunan “CaldigiEnstruman” özelliği MüzikAleti sınıfının bir nesnesini döndürüyor. Bunun nedeni, söz konusu nesnenin işlevsel açıdan uygulamanızda önemli bir yere sahip olması. Örneğin; söz konusu simülasyonda müzisyenimiz, çaldığı enstrumana göre sahnede yerini alacak. Bu durumda, burada bir miras kavramının (inheritance) kokusunu alıyor olmalısınız. Çünkü; bateri çalan müzisyen sahnenin bir yerinde, elektro gitar çalan müzisyenimiz başka bir yerinde olacağına göre, bu enstrumanların hepsi bir base class'tan türemiş olmalıdır. Haliyle, burada base class'ımız da MüzikAleti oluyor. Fakat dikkat edin; müzisyenlerinizden hiçbiri “müzik aleti” isminde bir alet çalamaz. Demek ki yazımın başında belirttiğim “market-ürün” örneğiyle karşı karşıyayız; MüzikAleti sınıfından başka sınıflara miras verilebilir ama kesinlikle nesne üretilmez. Kısacası MüzikAleti sınıfı kesinlikle abstract bir sınıf olacaktır.

Gelelim MüzikAleti sınıfının detaylarına;

```
public abstract class MuzikAleti
{
    public string Marka { get; set; }
    public string Aciklama { get; set; }
}
```

Şimdi bir düşünelim; bir orkestrada olabilecek tüm müzik aletlerini gözünüzün önüne getirin. Tümünde varolan ortak metodları düşünün lütfen... Mesela Çal metodu hepsinde ortaktır öyle değil mi? Madem öyle; bu metodun, MuzikAleti sınıfının bir üyesi olması gerekiyor..

Peki Cal() metodunuzu MuzikAleti sınıfına yazıyorsunuz. Ama metod gövdesinde ne yazacaksınız? Yani MuzikAleti sınıfından sadece miras alınacağına göre; Cal() isimli metodunuz da tüm türetilen sınıflara aktarılacağına göre hangi müzik aletinin çalınma şeklini anlatacaksınız bu metodda? Diyelim ki piyanonun çalma şeklini yazdınız metod gövdesine. O zaman MuzikAleti sınıfından türeyen Gitar sınıfının da Cal() metodu piyano gibi çalınacak!

Bu sitenin takipçileri veya object oriented kavramlarını bilen arkadaşlar burada polymorphic bir metoddan bahsettiğimi anlamışlardır. Ancak, buradaki Cal() metodu virtual anahtar kelimesi ile imzalanamaz. Çünkü virtual dersiniz türetilmiş sınıfta override demek ZORUNDA DEĞİLSİNİZ. Oysa bizim örneğimizde Cal() metodu hemen hemen her türetilmiş sınıfta farklı olmak zorundadır ve mecburen Override edilmesi gerekmektedir. Öyleyse bu Cal() metodu da abstract bir metoddur:

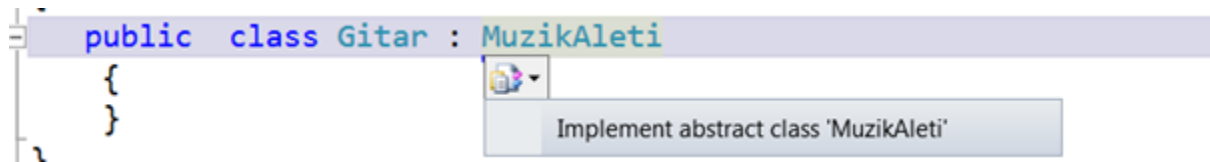
```
public abstract class MuzikAleti
{
    public string Marka { get; set; }
    public string Aciklama { get; set; }

    public abstract string Cal();
}
```

Hemen tekrar edeyim: virtual anahtar kelimesi, “bu metod ezilebilir” derken, abstract kelimesi “bu metod ezilmek zorundadır” der. Dikkat ederseniz, abstract Cal() metodunun bir gövde kodu olmadığını görürsünüz. Anlamı çok basittir: “Cal() metodunun nasıl çalışacağını, MuzikAleti sınıfından türetilmiş sınıf belirler. ”

Aslında bu noktada msdn'den bir alıntı yapmak uygun olabilir; “abstract metodlar; gövde kodları türetilmiş sınıflar tarafından tamamlanacak olan eksik bırakılmış metodlardır”.

Bakalım Visual Studio IDE'si bir abstract sınıftan bir sınıf türetirken bize nasıl yardımcı oluyor:



“Implement edim mi abü?” şeklinde türkçeye çevirebileceğimiz bu yardımsever teklife sıcak baktığımızı ifade etmek için seçeneğe tıkladığımızda bakın ne oluyor:

```
public class Gitar : MuzikAleti
{
    public override string Cal()
    {
        throw new NotImplementedException();
    }
}
```

İşte! Ne demiştim, abstract metodlar; türetilmiş sınıflar tarafından override edilmek zorundadırlar. Şimdi siz, gitara özel cal() metodunu yazabilirsiniz:

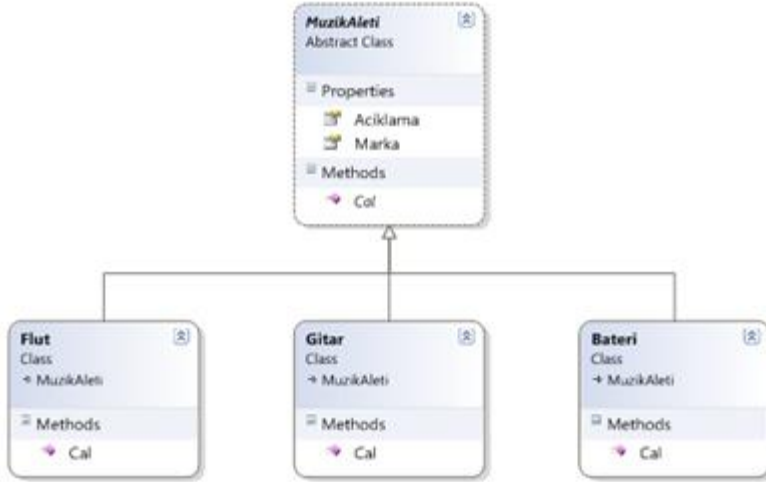
```
public class Gitar : MuzikAleti
{
    public override string Cal()
    {
        return "Dın dırırır dırırın";
    }
}
```

Ben biraz daha örneklendireyim:

```
public class Bateri:MuzikAleti
{
    public override string Cal()
    {
        return "dıp tıs dıp dıp tıs";
    }
}

public class Flut : MuzikAleti
{
    public override string Cal()
    {
        return "dut duru duut";
    }
}
```

Unutmadan belirtelim; eğer class diyagramdan bakacak olursanız, abstract sınıflar, çizgili çerçeve içinde gösterilirler.



Pekala; gelin şimdi, senaryomuza göre programımıza son şeklini verelim.

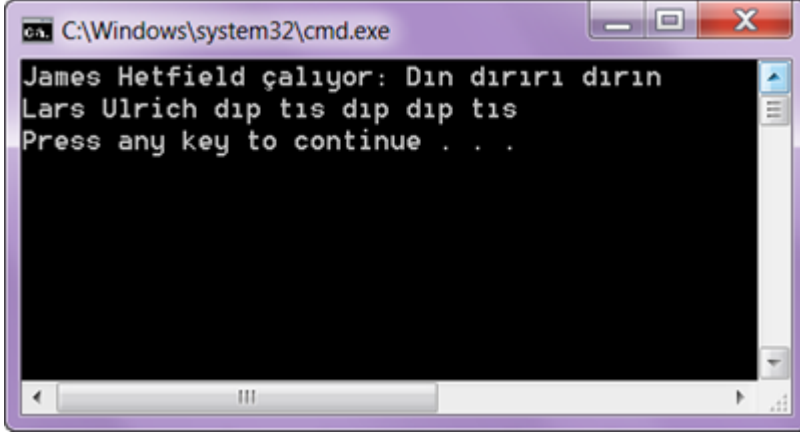
```
static void Main(string[] args)
{
    //MuzikAleti sınıfından türetilmiş iki sınıftan birer nesne ürettim
    //(gitar ve davul)
    Gitar gitar = new Gitar();
    gitar.Marka = "Fender";

    Bateri davul = new Bateri();
    davul.Marka = "Yamaha";

    //..ve simüle etmek için iki müzisyen nesnesi:
    Muzisyen gitarist = new Muzisyen();
    gitarist.Ad = "James";
    gitarist.Soyad = "Hetfield";
    //bu satıra dikkat!
    gitarist.CaldigiEnstruman = gitar;

    Muzisyen davulcu = new Muzisyen();
    davulcu.Ad = "Lars";
    davulcu.Soyad = "Ulrich";
    davulcu.CaldigiEnstruman = davul;
    //ve işte... Muzisyenin marifetini gördüğümüz an:
    Console.WriteLine(gitarist.Ad + " " + gitarist.Soyad + " çalıyor: " + gitarist.CaldigiEnstruman.Cal());
    Console.WriteLine(davulcu.Ad + " " + davulcu.Soyad + " " + davulcu.CaldigiEnstruman.Cal());
}
}
```

Ve sonuç:



```
C:\Windows\system32\cmd.exe
James Hetfield alıyor: Dın dırırđ dırđn
Lars Ulrich dđp tđs dđp dđp tđs
Press any key to continue . . .
```

Evet sevgili dostlarđm. Nesne Yönelimli Programlama konusunun bu 8. Makalesini de burada bitirmiř olduk. Bir sonraki makalede görüşmek dileđiyle.

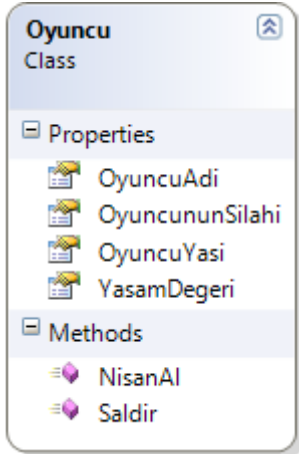
Interface

Merhaba; yazılım dünyasının merdivenlerini tırmanmaktan vazgeçmeyen dostlarım. Başlıktan da anladığınız üzere; Muhteşem Yüzyıl dizisinden daha heyecanlı olan (ya da bana öyle geliyor 😊) yazı dizimize Interface konusu ile devam ediyoruz.

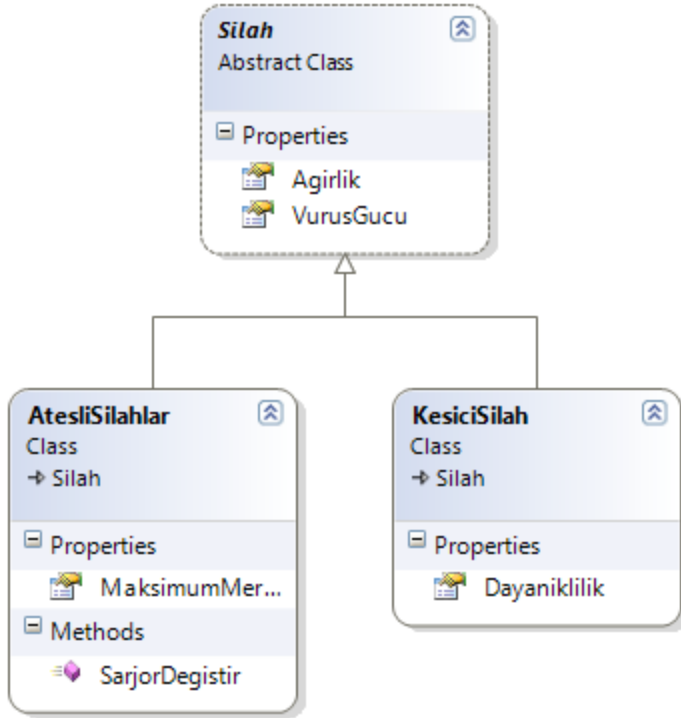
Interface'in kelime anlamı olan “arayüz”e takılırsanız işin içinden çıkamazsınız baştan söyleyeyim :). İçinizden “o zaman adını neden interface koymuşlar” diyor olabilirsiniz. Elbette bunun mantıklı bir sebebi var ve birzandan bu sebebi anlayacaksınız da. Gelin önce basit bir tanımla başlayalım; Interface'ler, geliştirdikleri (implemente ettikleri) sınıflara yetenek kazandıran yapılardır. Yani “tip” oldukları söylenemez. Peki bir interface'e ne zaman ihtiyaç duyarız? İşte bu sorunun yanıtını verebilmek için, sınıf üyelerinin (metod, olay ve özellik) bir yeteneği temsil edip etmediğini nasıl anlarız sorusuna cevap vermemiz gerekiyor. Haydi bakalım .NET kasabasının meşhur Object Oriented Lokantasına bir uğrayalım.

Söz konusu lokantamızda elbette bir çok nesne mevcut. Ama ben, bu lokantadaki insan nesnesi üzerinde duracağım. Efendim lokantamızdaki insanlar başlangıç olarak ikiye ayrılıyor; çalışanlar ve müşteriler. Davranışsal olarak birbirlerinden farklı olan bu iki nesne için çözümü görebiliyorsunuz sanırım: abstract bir insan sınıfı. Hem çalışan hem de müşteri sınıflarına miras veren bir temel sınıf. Peki, siz bu lokantaya müşteri olarak girdiniz. Masanıza oturdunuz ve kararınızı verdiniz. Şimdi sipariş verme zamanı. Peki... Kime sipariş vereceksiniz? Ya da daha doğrusu, garsonu nasıl tanıyacaksınız? Yakasındaki isimliğinden veya giydiği kıyafetten öyle değil mi? İşte interface! O kıyafet, abstract insan sınıfından türemiş olan çalışan sınıfına garsonluk “yeteneği” kazandırıyor. Ya da başka bir deyişle garson o kıyafeti giyerek; “sipariş al” metodunu implemente ediyor.

Bu teorik örnek sanıyorum; Interface'e ne zaman ihtiyaç duyacağımız sorusuna cevap veriyor. Ama elbette, kod örneği yapmadan şuradan şuraya göndermiyorum sizi... Örneğimiz, Counter Strike tarzı bir oyun üzerine olacak (aslında Diablo III'e özel bir örnek yapmak niyetindeydim ancak, oradaki silahlar daha karışık). Önce nesne modelimizi oluşturalım:



İşte, Oyuncu sınıfımızın basitçe bir görüntüsü. Bu sınıfın “OyuncununSilahi” isimli property’si, geriye Silah sınıfından bir nesne döndürüyor. Silah sınıfı da tahmin edersiniz ki abstract bir sınıf. Çünkü, oyunumuzda hem ateşli silahlar hem de kesici silahlar mevcut ve her ikisi de bambaşka özellikleri olan sınıflar. Öyleyse durum aşağıdaki gibi:



Peki, oyuncu sınıfında yer alan “NisanAl” metoduna odaklanalım. Bu metod, oyuncununSilahi özelliğinden dönen nesnenin türüne göre bir davranış sergileyecek. Yani, eğer oyuncunun dürbünlü bir silahı varsa; hedef yakınlştırılacak, dürbünsüz ise; göz gez arpacık görünümüne geçecek ya da oyuncuda bıçak varsa, bıçağı dik konuma getirecek. Bu noktada, bunu sağlamak için ne yapacağız? Bir an için abstract silah sınıfına “yakınlaştır” isimli bir abstract metod yazdığımı düşünelim. Ama o zaman KesiciSilah sınıfında da “yakınlaştır” metodu oluyor. Burada bariz bir mi mari hata var. Biz bu duruma kısaca WTF (What the folk 😄) diyoruz.

İşte o zaman anlıyoruz ki, bir silahın dürbünlü olması, o silahın “yeteneğı”dir. İşte şimdi gerçekten bir interface’e ihtiyaç duyuyorsunuz. Haydi kolları sıvayalım

```
public interface IYakinlastirilabilir
{
    void Yakinlastir();
}
```

Interface’in bir sözleşme olduğunu düşünelim. Bu sözleşmeyi bir sınıfın kabul etmesi demek, sözleşme içerisinde yer alan tüm üyeleri bulunduracağını taahhüt etmek demektir. Bu açıklamaya göre, Interface içerisindeki hiçbir üye access modifier içermemelidir. Çünkü bu üyeler, implemente eden sınıfta public olmak zorundadır. Ayrıca interface üyeleri gövde kodu da içeremezler. Amaçları sadece kalıp oluşturmaktır.

Şimdi dürbünlü bir ateşli silah ekleyelim ve IYakinlastirilabilir interface’ini de imlemente etmesini sağlayalım:

```
public class M51 : AtesliSilahlar, IYakinlastirilabilir
{
    public void Yakınlastir()
    {
        Console.WriteLine("hedef yakınlaştı");
    }
}
```

M51 isimli silah, yakınlaştırma “yeteneğine” sahip dürbünlü bir silah oldu gördüğünüz gibi. İşte şimdi, oyuncu sınıfında yer alan “NisanAl” metoduma geri dönebilirim.

```
public void NisanAl()
{
    if (OyuncununSilahi is IYakinlastirilabilir)
    {
        IYakinlastirilabilir zoom = (IYakinlastirilabilir)OyuncununSilahi;
        zoom.Yakinlastir();
    }
}
```

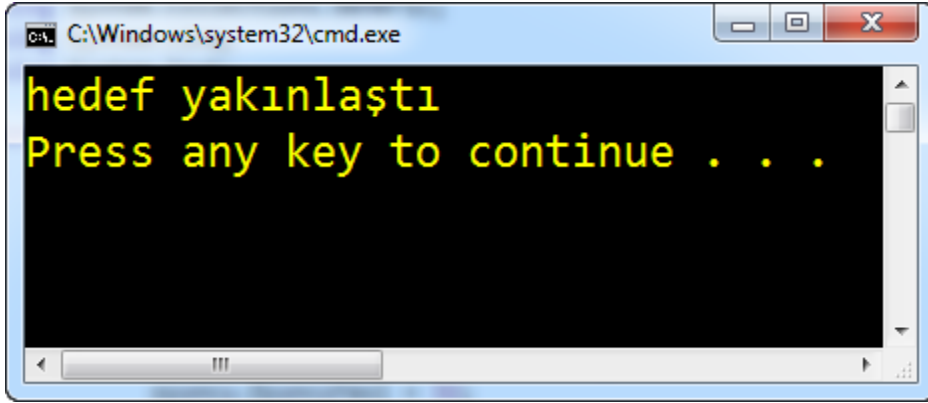
Burada gördüğünüz gibi önce OyuncununSilahi özelliğinin Interface’imi implemente edip etmediğini soruyorum. Eğer etmişse Interface’den geçen üyeleri zoom isimli nesneye atıyorum. Ardından da Yakınlastir() metodunu çağırıyorum.

... Ve final:

```
class Program
{
    static void Main(string[] args)
    {
        Oyuncu oyuncu = new Oyuncu();
        oyuncu.OyuncuAdi = "Şişman Adam";
        oyuncu.OyuncuYasi = 32;
        oyuncu.OyuncununSilahi = new M51();

        oyuncu.NisanAl();
    }
}
```

Çıktı:



```
C:\Windows\system32\cmd.exe
hedef yakınlaştı
Press any key to continue . . .
```

İşte sevgili dostlarım; interface'in kullanımına dair minik bir örnek size... Aman yanlış anlaşılmasın; Interface'in buradaki kullanımı sadece bir pattern. Karşınıza interface'in karşınıza çıkabileceği birçok senaryo var. Artık onları araştırmak da size düşüyor.

O zaman; bir makalemizin daha sonuna gelmiş bulunuyoruz sevgili dostlarım. Sağlıcakla kalın.