

2005 Yılı Yazıları

- 0 - Derinlemesine Session Kullanımı - 2 - 07 Ocak 2005 Cuma
- 1 - Caching Mekanizmasını Anlamak - 1 - 21 Ocak 2005 Cuma
- 2 - Caching Mekanizmasını Anlamak - 2 - 07 Şubat 2005 Pazartesi
- 3 - RijndaelManaged Vasıtasıyla Encryption(Şifreleme) ve Decryption(Deşifre) - 23 Şubat 2005 Çarşamba
- 4 - Bağlantısız Katmanda Concurrency Violation Durumu - 06 Mart 2005 Pazar
- 5 - Self Referencing Relations ve Recursive(Yinelemeli) Metodlar - 14 Mart 2005 Pazartesi
- 6 - Command Nesnelerine Dikkat! - 27 Mart 2005 Pazar
- 7 - DataReader Nesnelerini Kullanırken... - 04 Nisan 2005 Pazartesi
- 8 - Ado.Net 2.0' da Transaction Kavramı - 17 Nisan 2005 Pazar
- 9 - Ado.Net 2.0 ve Data Provider-Independent Mimari - 24 Nisan 2005 Pazar
- 10 - Kendi İstina Nesnelerimizi Kullanmak (ApplicationException) - 23 Mayıs 2005 Pazartesi
- 11 - Operator Overloading (Operatörlerin Aşırı Yüklenmesi) - 03 Haziran 2005 Cuma
- 12 - C# 2.0 ve Anonymous (İsimsiz) Metodlar - 16 Haziran 2005 Perşembe
- 13 - C# 2.0 ve Static Sınıflar - 20 Haziran 2005 Pazartesi
- 14 - C# 2.0 ve Nullable Değer Tipleri - 22 Haziran 2005 Çarşamba
- 15 - C# 2.0 ile Partial Types (Kısmi Tipler) - 27 Haziran 2005 Pazartesi
- 16 - C# 2.0 Covariance ve Contravariance Delegates - 30 Haziran 2005 Perşembe
- 17 - C# 2.0 İçin İterasyon Yenilikleri - 05 Temmuz 2005 Salı
- 18 - Dizilere(Array) İlişkin Üç Basit Öneri - 14 Temmuz 2005 Perşembe
- 19 - Tip Güvenli (Type Safety) Koleksiyonlar Oluşturmak - 1 - 23 Temmuz 2005 Cumartesi
- 20 - Tip Güvenli (Type Safety) Koleksiyonlar Oluşturmak - 2 - 31 Temmuz 2005 Pazar
- 21 - SortedList ve Hashtable İçin 2 Basit Öneri - 06 Ağustos 2005 Cumartesi

- 22 - Numaralandırıcıları Kullanmak İçin Bir Sebep - 28 Ağustos 2005 Pazar
- 23 - Web Uygulamalarında Custom Paging - 03 Eylül 2005 Cumartesi
- 24 - CallBack Tekniği ile Asenkron Metod Yürütmek - 09 Eylül 2005 Cuma
- 25 - Boxing ve Unboxing Performans Kritiği - 26 Eylül 2005 Pazartesi
- 26 - Constructor Initializers (Yapıcı Metod Başlatıcıları) Deyip Geçmeyin - 03 Ekim 2005 Pazartesi
- 27 - Ado.Net 2.0(Beta 2) - Connection String Security (Bağlantı Katarı için Güvenlik) - 17 Ekim 2005 Pazartesi
- 28 - Kendi Referans Tiplerimizi Klonlamak - 14 Kasım 2005 Pazartesi
- 29 - C# 2.0 ile Generic Delegates - 21 Kasım 2005 Pazartesi
- 30 - Generic Mimaride Kısıtlamaların(Costraints) Kullanımı - 31 Aralık 2005 Cumartesi

Derinlemesine Session Kullanımı - 2 - 07 Ocak 2005 Cuma

asp.net, session,

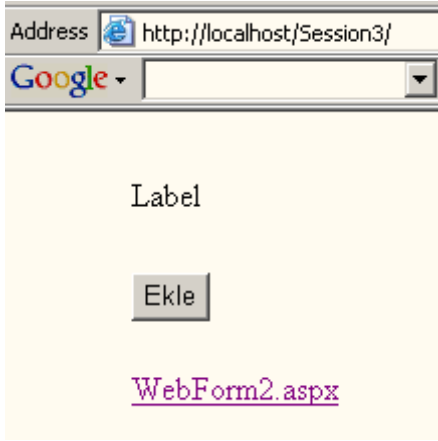
Değerli Okurlarım, Merhabalar.

Bir önceki makalemizde hatırlayacağınız gibi, Session nesnelerinin kullanımını incelemeye başlamıştık. Bu makalemizde ise, Session nesnelerinin nerelerde saklanabildiğine değinmeye çalışacağız. Varsayılan olarak Session nesneleri **In-Proc** (işlem içi) modunda saklanırlar. Yani web sayfasının çalıştığı asp.net work process' in içinde, dolayısıyla bu işlerin çalıştığı web sunucularındaki bellek alanlarında tutulurlar. Bu özellikle Session nesnelerine bilgi yazma ve okumada önemli bir avantajdır.

Nitekim, erişim doğrudan ram üzerindeki bölgelere doğru olduğu için diğer bahsedeceğimiz modlara nazaran göreceli olarak oldukça hızlı bir erişim söz konusudur. Lakin, web sunucusunun başına bir şey gelmesi halinde (örneğin sunucunun bir anda restart olması gibi) bellekte tutulan tüm Session nesneleri bir anda kaybedilir. Bu da çok sayıda kullanıcının açtığı oturumlara ait bilgilerin tamamının kaybolması anlamına gelmektedir. Bunu basit bir örnek ile gösterebiliriz. Aşağıdaki web uygulamasında, Session nesnesine bir değer atanmaktadır. Session' ın time-out süresi varsayılan halinde (Yani 20 dakika olarak) bırakılmıştır.

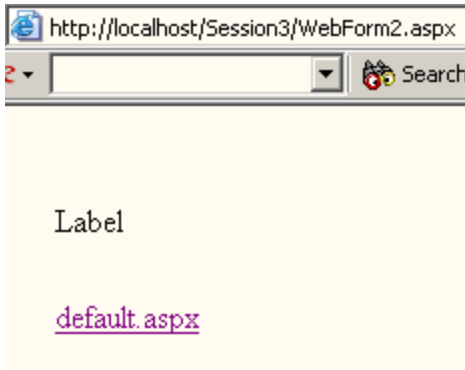
default.aspx kodları ve Form' un ekran görüntüsü;

```
private void Page_Load(object sender, System.EventArgs e)
{
    if(Session["Bilgi"]!=null)
    {
        Label1.Text=Session["Bilgi"].ToString();
    }
}
private void btnEkle_Click(object sender, System.EventArgs e)
{
    Session["Bilgi"]="Deneme";
    Label1.Text=Session["Bilgi"].ToString();
}
```

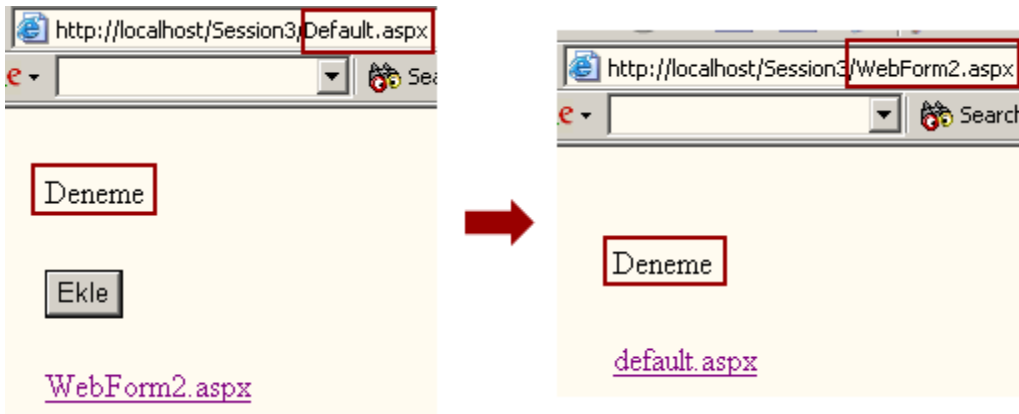


WebForm2.aspx kodları ve Form' un ekran görüntüsü;

```
private void Page_Load(object sender, System.EventArgs e)
{
    if(Session["Bilgi"]!=null)
        Label1.Text=Session["Bilgi"].ToString();
}
```



Bu örneği çalıştırdığımızda, default.aspx sayfasında Ekle butonuna basarsak Session nesnemize Deneme bilgisi eklenecektir. Eğer WebForm2.aspx sayfasına geçerseniz, Session bilgisinin Label kontrolüne yazıldığını görürüz.



Şimdi web uygulamamıza ait **Global.asax** dosyasında herhangi bir değişiklik yapıp uygulamamızı yeniden derleyelim. Ben örnek olarak pek bir anlam ifade

etmeyen bir yorum satırı girdim ve uygulamayı yeniden derledim. Tabi bunu yaparken, web tarayıcımızda sayfalarımız açık halde bulunmalıdır.

```
protected void Application_Start(Object sender, EventArgs e)
{
    //YORUM SATIRIDIR.
}
```

Şimdi linklerimize tıklayarak sayfalar arasında tekrar gezindiğimizde, henüz time-out süresi dolmamış olan Session nesnelerinin kaybedildiğini ve Label kontrollerinde Session nesnesine ait içeriğin yazmadığını görürüz. Kısacası, In-Proc modunda olan (Yani işlem içi - In Process) Session nesneleri kaybedilmiştir. Elbetteki Session nesnelerinin bu şekilde kaybolması dışında da oluşabilecek istisnalar vardır. Örneğin sunucunun istem dışı bir şekilde kapanması gibi.

Asp.Net ile birlikte durum yönetiminde (state management), Session nesnelerinin saklanabilmesi için iki teknik daha geliştirilmiştir. Bu teknikler yardımıyla, durum nesnelerinin yukarıdaki gibi nedenlerden ötürü kaybolmalarının önüne geçilebilmektedir. Bu tekniklerden bir tanesi Session nesnelerinin bir **Sql Veritabanında** tutulduğu **SQLServer** modu, diğeri ise Session nesnelerinin **ASP.NET State Service Windows Servisinde** tutulduğu **StateServer** modudur. İlk olarak SQLServer modunu inceleyeceğiz.

SQLServer modunda, Session nesnesine ait tüm bilgiler bir Sql Sunucusunda bu iş için özel olarak hazırlanmış bir veritabanında tutulmaktadır. Böylece, Session nesnelere ait içerik, istenen süre kadar (aylarca bile olabilir) fiziki bir disk bölgesinde saklanabilmektedir. Bu ayrıca, veritabanının başka bir sunucuda konuşturulmasıyla, Web Çiftliklerinin (**Web Farms**) yapısına uygun bir oluşumda imkan tanır. Böylece, Web Sunucusunda oluşabilecek aksaklıklardan doğacak sorunlar Sql Sunucusunu etkilemeyecek, dolayısıyla Session' lar korunmuş olacaktır. Elbette bu sistemin de dezavantajı vardır.

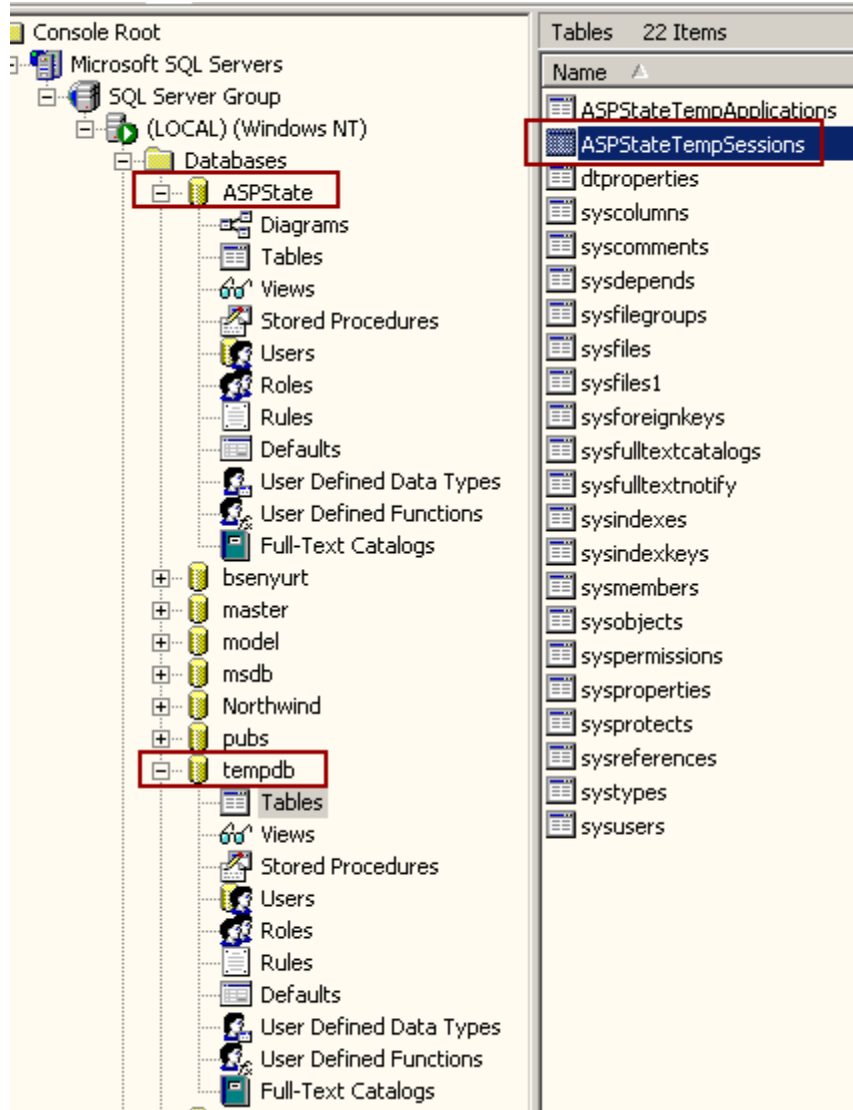


Session nesnelerini ayrı bir sunucudaki veri tabanında tutmak her ne kadar **güvenlik ve tutarlılık** açısından **yüksek performans** sağlasada, bilgilere erişimin In-Proc moda göre **daha yavaş** olmasında neden olur. Bu elbetteki verinin okunması veya yazılması için sürekli veritabanına doğru atılan turların bir sonucudur.

SQLServer modunda kullanılan ASPState isimli veritabanında Session nesnelerinin yazılma, silinme gibi işlemleri için kullanılan stored procedure' ler yer alır. Session nesnelere ait asıl içerik ise tempdb isimli veritabanında yer alan tablolarda tutulmaktadır. Microsoft .NET Framework bu veritabanını ve içeriğini kurmak için gerekli Sql kodlarını içeren script dosyalarını içerir. Windows XP sistemlerinde bu dosyaya (**InstallSqlState.sql**)

D:\WINDOWS\Microsoft.NET\Framework\v1.1.4322 adresinden ulaşabilirsiniz. Bu sql script dosyasını **Sql Query Analyzer** ile çalıştırdığımızda,

Sql Sunucusunda **ASPState** isimli bir veritabanı oluşturulduğunu görürüz. Ayrıca Session nesnelerini tutacak olan tablolarda **tempdb** veritabanı içerisine eklenirler.



Burada bizi asıl ilgilendiren kısım, **tempdb** veritabanında oluşturulan **ASPStateTempSessions** isimli tablodur. Nitekim bu tablo Session verilerini saklamak üzere kullanılmaktadır. Tablonun yapısı aşağıdaki şekilde görüldüğü gibidir.

	Column Name	Data Type	Length	Allow Nulls
	SessionId	char	32	
	Created	datetime	8	
	Expires	datetime	8	
	LockDate	datetime	8	
	LockDateLocal	datetime	8	
	LockCookie	int	4	
	Timeout	int	4	
	Locked	bit	1	
	SessionItemShort	varbinary	7000	✓
	SessionItemLong	image	16	✓

Dikkat edecek olursanız, **SessionId** isimli alan tablonun Primary Key alanıdır. Bu alanda, web sunucu tarafından otomatik olarak oluşturulan **ASP.NET_SessionId** değeri tutulmaktadır. Bunun dışında Session'ın yaratıldığı tarih, oturumun sona ereceği süre bilgisi gibi veriler dışında Session nesnesinin içeriğinin saklanacağı iki önemli alan daha vardır. Bunlar **SessionItemShort** ve **SessionItemLong** alanlarıdır. Her iki alandan hangisinin kullanılacağı, Session nesnesine atanan verinin büyüklüğüne göre belirlenmektedir. İşte bu noktada karşımıza önemli bir sorun çıkar.



Bir DataSet içeriğini Session nesnesine aktardığımızda, SessionItemShort veya SessionItemLong alanlarından hangisi kullanılırsa kullanılsın, Session nesnesinin içerdiği veri nasıl olurda **tek bir alan içerisine** sığdırılabilir?

İşte burada bir önceki makalemizde bahsettiğimiz gibi Session nesnesinin taşıyacağı verinin serileştirilebilir olması gerekliliği ortaya çıkmaktadır. Böylece ister binary olarak ister XML olarak DataSet nesnesinin içeriği serileştirilebilir ve tek bir alan içerisine yazılıp okunabilir. Bu elbetteki Session ile taşımak istediğimiz her nesne örneği için geçerli bir durumdur. (Örneğin kendi yazdığımız bir sınıf için.)



Out-of-Proc (İşlem dışı) modlarda, Session nesnesine atanan nesnelerin **mutlaka serileştirilebilir (Serializable)** olmaları gerekmektedir.

Şimdi basit olarak yukarıda işlediğimiz örneğimizde kullandığımız Session nesnesini, **SQLServer** modunda saklayalım. Varsayılan olarak, Session nesneleri In-Proc modda tutulduklarından **web.config** dosyasında yer alan **sessionState** boğumunun standart içeriği aşağıdaki gibidir.

```
<sessionState
  mode="InProc"
  stateConnectionString="tcpip=127.0.0.1:42424"
  sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
  cookieless="false"
  timeout="20"
/>
```

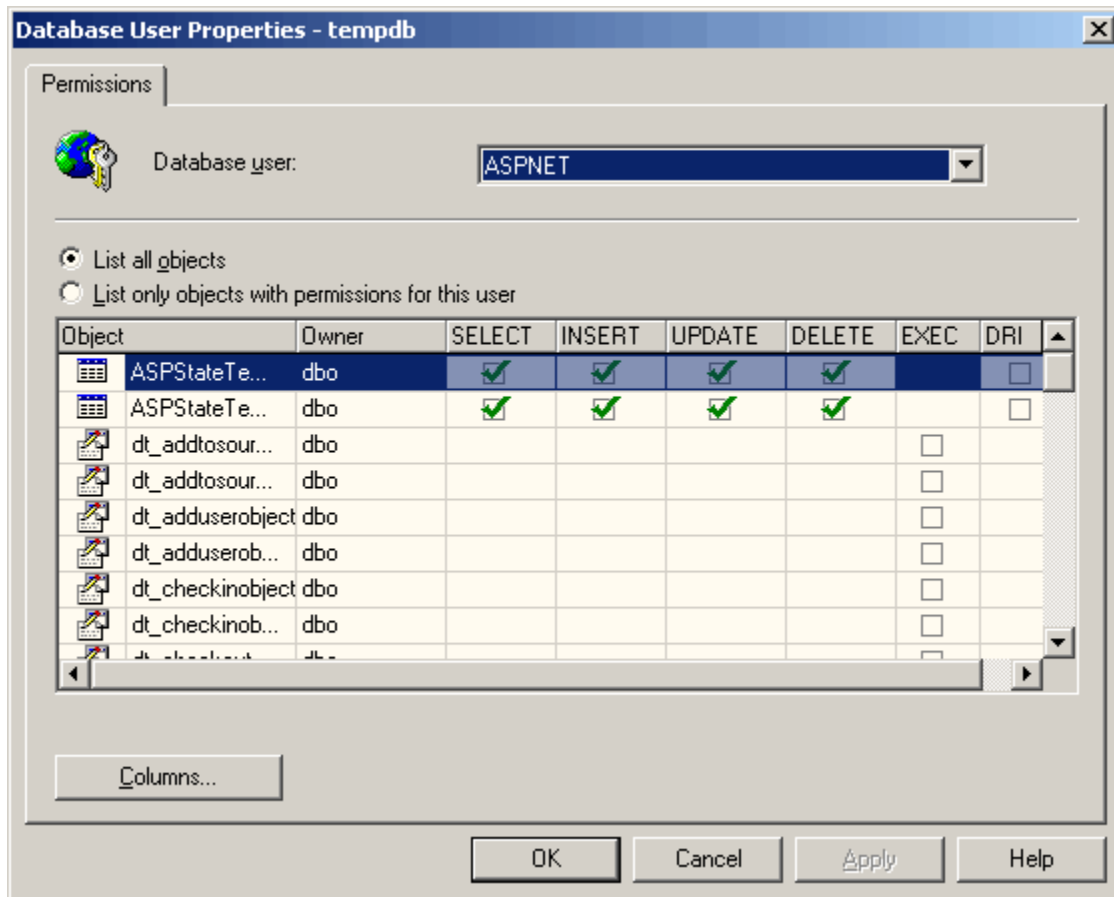
Mode özelliğinin değeri varsayılan olarak InProc' tur. Yani, Session nesneleri işlem içinde tutulmaktadır. Session bilgilerini veritabanına yazabilmek için sessionState boğumunu aşağıdaki gibi düzenlememiz yeterli olacaktır.

```
<sessionState
  mode="SQLServer"
  stateConnectionString="tcpip=127.0.0.1:42424"
```

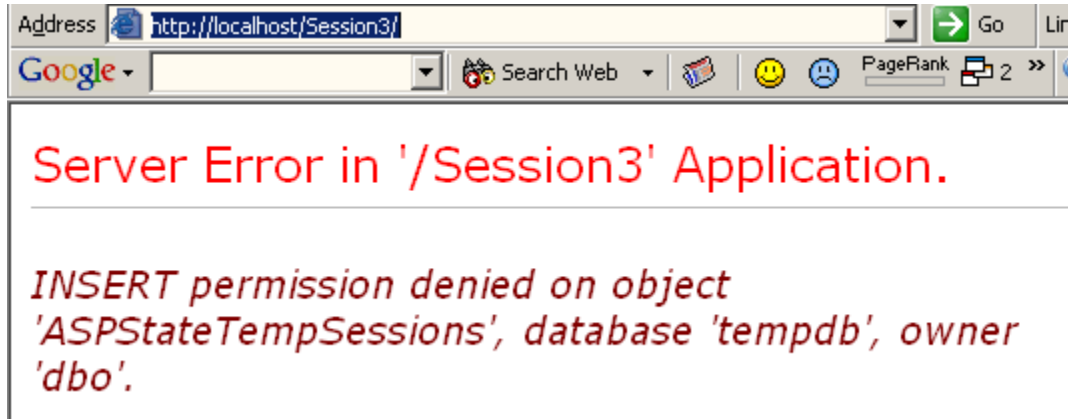
```
sqlConnectionString="data source=127.0.0.1;user id=sa;password=123456"
cookieless="false"
timeout="20"
/>
```

Burada dikkat etmemiz gereken en önemli nokta, **sqlConnectionString** özelliğinin aldığı bağlantı cümlecisidir. Bu özellikte, sql sunucusunun bulunduğu lokasyon **data source** ile belirtilmektedir. Varsayılan olarak Sql sunucusunun localhost üzerinde bulunduğu düşünüldüğünden bu değer 127.0.0.1 ip değerini alır. Ancak **Web Çiftliği (Web-Farm)** gibi sistemlerde eğer Sql Sunucusunun bulunduğu adres farklı ise data source değerini bu adrese göre ayarlamamız gerekecektir. Diğer taraftan, Ado.Net' te olduğu gibi bağlantının yapılacağı veritabanı adının burada belirtilmesine gerek yoktur. Bunun sebebi SQLServer modunda Session nesnelerinin nereye yazılacaklarının zaten belli olmasıdır. Bir diğer husus ise, mutlaka ve mutlaka bağlantıyı belli bir kullanıcı adı ve şifresi üzerinden yapmamızın güvenlik açısından daha sağlıklı olacaktır.

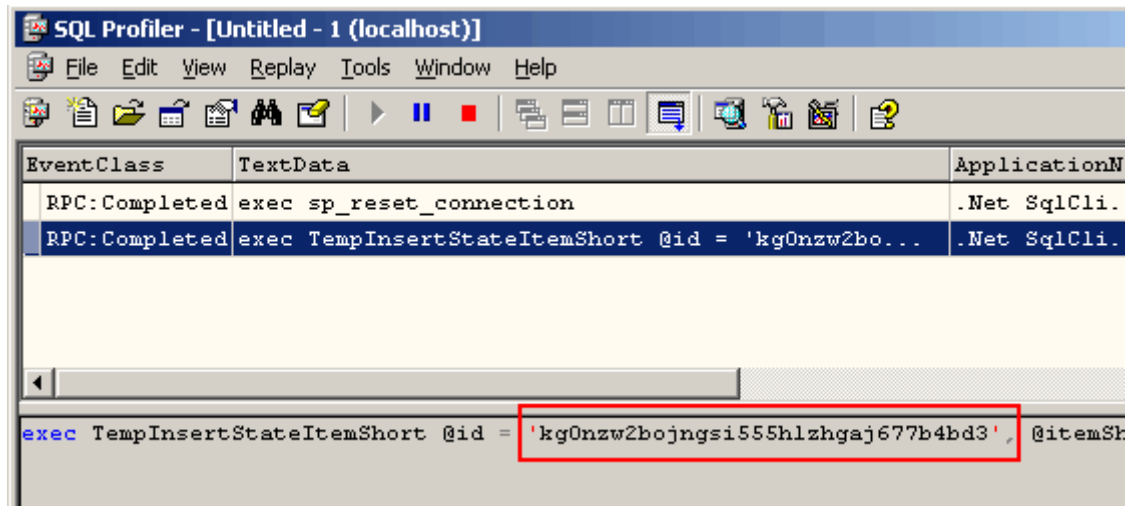
Şunu da hatırlatmakta fayda var. InstallSqlState script' i her ne kadar session yönetimi için gerekli veritabanı düzenlemelerini yapsada, ASPNET kullanıcısına ASPState veritabanındaki stored procedure' leri çalıştırma ve tempdb içindeki ASPStateTempSessions ile ASPStateTempApplications tabloları için gerekli **Select, Insert, Update, Delete** komutlarını yürütebilme izinlerini vermemiz gerekiyor.



Aksi takdirde ASPNET kullancısının bu sp ve sql komutlarını çalıştırma yetkisi olmayacağından web uygulamamızda aşağıdakine benzer türden hata sayfaları ile karşılaşabiliriz.



Gerekli izinleride verdikten sonra artık uygulamamızı çalıştırabiliriz. Uygulamamızı çalıştırdığımız sırada eğer Sql Profiler ile arka planda olanları izlersek hemen bir sp' nin çalıştırıldığını ve sp' ye parametre olarak bir GUID' in aktarıldığını görürüz. Buradaki GUID, web sunucusu tarafından üretilen **ASP.NET_SessionId**' den başka bir şey değildir. Session nesnesi ilk kez yaratıldığından ilgili tabloya **INSERT** işlemini uygulayan bir sp çalışmaktadır.



Tam bu noktada ASPStateTempSessions tablomuza bakacak olursak, yukarıdaki sp'ye aktarılan Id' değerini SessionId alanına almış yeni bir satır oluşturulduğunu görürüz.

SQL Server Enterprise Manager - [Data in Table 'ASPStateTempSessions' in 'tempdb' on '(LOCA...]

SessionId	Created	Expires	LockDate	LockDateLoc
kg0nzw2bojnysi555h1zhgaj677b4bd3	06.01.2005 20:30:14	06.01.2005 20:50:14	06.01.2005 20:30:14	06.01.2005 20:30:14

Elbetteki bu noktada, Session' ımıza henüz bir bilgi aktarmadığımız için tabloda yer alan ilgili alanlara herhangi bir bilgi yazılmamıştır. default.aspx sayfasında ekle butonuna basarsak, başka bir sp' nin bu kez var olan SessionId' li satırı güncellemek üzere çalıştırıldığını ve parametre olarakta encrypt edilmiş Session içeriğinin gönderildiğini görürüz.

```

TextData
exec sp_reset_connection
declare @P1 varbinary(7000) set @P1=0x1400000000000000FF declare @P2 bit set @P2=0 declare @
exec TempUpdateStateItemShort @id = 'kg0nzw2bojnysi555h1zhgaj677b4bd3', @itemShort = 0x140

```

Dolayısıyla Session içeriği tabloda yer alan ilgili satıra (ASP.NET_SessionId değerine sahip olan satır) yazılmış olacaktır. Eğer makalemizin başındaki örneğimizde yaptığımız gibi, Global.asax dosyasında bir değişiklik yapıp uygulamayı yeniden derleyip time-out süresinden önce Session' ları okumak istersek, Session' lara ait değerlerin kaybolmadığını kolayca tespit edebiliriz. Session' ların yaşam süreleri dolduğunda otomatik olarak silindiklerini biliyoruz. InstallSqlState script' i ayrıca zaman aşımına uğramış Session' ların otomatik olarak kaldırılması için gerekli bir **job** nesnesinide sql sunucusuna yükler. (Job nesnesinin çalışabilmesi için **Sql Server Agent** servisinin çalışıyor olması gerekmektedir.)

Jobs 1 Item

Name	Category
ASPState_Job_DeleteExpiredSessions	[Un

Şimdi aşağıdaki örneği inceleyelim. Bu örneğimizde, Session nesnesine bizim tanımladığımız bir nesne örneğini aktarıyoruz.

Personel isimli sınıfımız;

```
public class Personel
{
    private string mAd;
    private string mSoyad;

    public string Ad
    {
        get
        {
            return mAd;
        }
        set
        {
            mAd=value;
        }
    }

    public string Soyad
    {
        get
        {
            return mSoyad;
        }
        set
        {
            mSoyad=value;
        }
    }

    public Personel(string ad,string soyad)
    {
        mAd=ad;
        mSoyad=soyad;
    }

    public Personel()
    {
    }
}
```

default.aspx;

```

private Personel pOku;

private void Page_Load(object sender, System.EventArgs e)
{
    if(Session["Eleman"]!=null)
    {
        pOku=new Personel();
        pOku=(Personel)Session["Eleman"];
        Label1.Text=pOku.Ad+" "+pOku.Soyad;
    }
}

private void btnEkle_Click(object sender, System.EventArgs e)
{
    Personel p1=new Personel("Burak Selim","Şenyurt");
    Session["Eleman"]=p1;
}

```

WebForm2.aspx

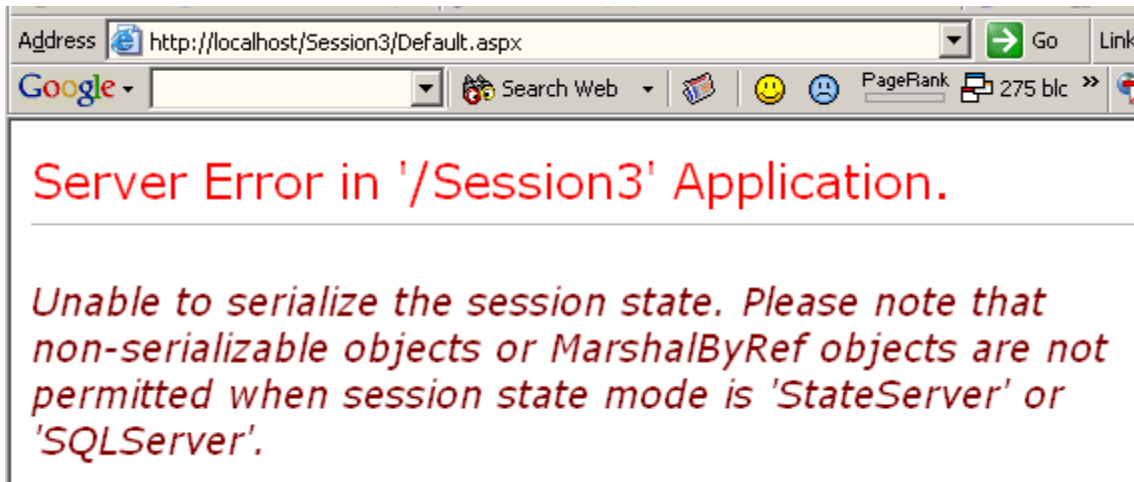
```

private Personel pOku;

private void Page_Load(object sender, System.EventArgs e)
{
    if(Session["Eleman"]!=null)
    {
        pOku=new Personel();
        pOku=(Personel)Session["Eleman"];
        Label1.Text=pOku.Ad+" "+pOku.Soyad;
    }
}

```

default.aspx sayfasında Session nesnemize Personel sınıfından p1 isimli nesne örneğimizi aktarıyoruz. Her iki sayfada da Session nesnesinin içeriğini okurken, Personel sınıfından bir nesne örneğine açıkça bir dönüştürme işlemi yaptığımıza dikkat edelim. Çünkü Session nesnesi, kendisine atanan verileri object tipinde taşımaktadır. Şimdi default.aspx sayfamızı tarayıcı penceresinde açar ve Ekle butonuna basarsak aşağıdaki hata mesajını alırız.

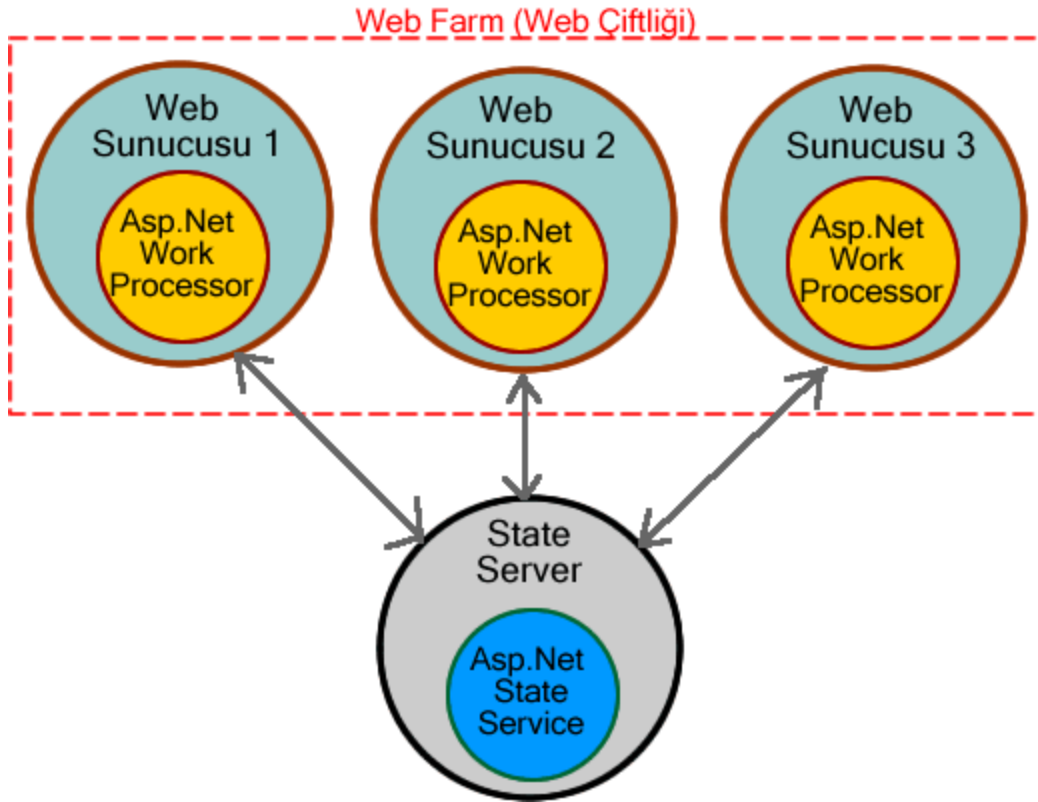


Sorun gayet açıktır. Personel sınıfımızın serileştirilebilir bir nesne olması gerekmektedir. Bu nedenle Personel sınıfımıza **Serializable** niteliğini (**Attribute**) eklememiz gerekiyor.

```
[Serializable]
public class Personel
{
    .
    .
    .
}
```

Şimdi Ekle butonuna tekrardan basarsak ve sayfalar arasında gezersek, Personel sınıfına ait nesne örneğinin başarılı bir şekilde taşındığını görürüz.

Session nesnelerini işlem dışından saklayabileceğimiz bir diğer seçenekte **ASP.NET State Service** isimli windows servisinin kullanılmasıdır. Bu kullanımda çoğunlukla, State Service başka bir sunucu üzerinde çalıştırılır ve diğer web sunucuları tarafından ortaklaşa kullanılır. Dolayısıyla, çalışan **Asp.Net work processor'** dan ayrı process' ler söz konusudur. Bu ayrı process' ler **State Server** üzerinde konuşlandırılır.



Session nesnelerini ASP.NET State Service ile kontrol edebilmek için öncelikle bu servisin çalıştırılması gerekir.

Application Layer Gateway Service	Started	Automatic	Local Service
Application Management		Manual	Local System
ASP.NET State Service	Started	Manual	Network S...
Automatic Updates	Started	Automatic	Local System
Background Intelligent Transfer Serv...	Started	Manual	Local System

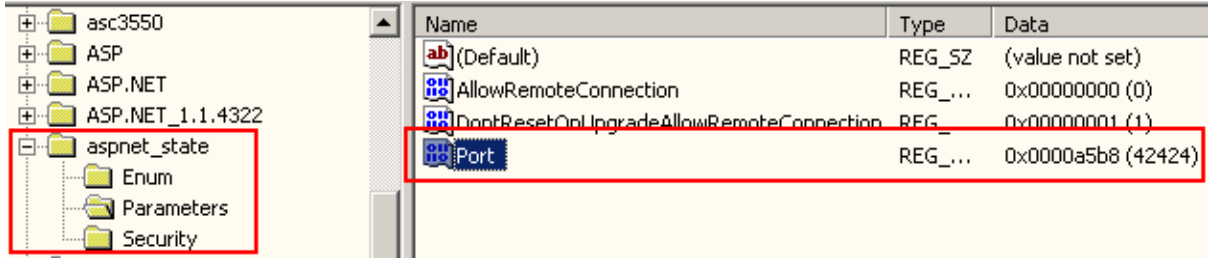
Servisin çalıştırılmasının ardından Web.config dosyasında da **sessionState** bölümünün özelliklerini aşağıdaki gibi değiştirmeliyiz.

```

<sessionState
  mode="StateServer"
  stateConnectionString="tcpip=127.0.0.1:42424"
  sqlConnectionString="data source=127.0.0.1;integrated security=SSPI"
  cookieless="false"
  timeout="20"
/>
  
```

StateServer modunda, State Server olarak kullanılacak sunucunun **tcpip adresi** ve ilgili **port numarası stateConnectionString** özelliğinde belirlenir. Biz burada local makineyi kullanıyoruz. Buradaki **42424** port numarası, ASP.NET State Service servisinin kullandığı varsayılan port numarasıdır. Dilersek bu numarayı değiştirmemiz mümkün. Bunun için, sistemdeki registry ayarlarına inmemiz gerekiyor. **Hot Key Local Machine** sekmesinde

\System\CurrentControlSet\Services\aspnet_state\Parameters altındaki **Port** elemanının deęerini deęiřtirmemiz yeterlidir.



Bu deęiřikliklerden sonra Session nesnelerini ASP.NET State Service' in kontrolü altında tutulmak üzere kullanabiliriz. Bu servis yardımıyla tuttuęumuz Session nesneleri için de **serileřtirilebilme řartı** aranmaktadır, burada hatırlatalım.

Böylece geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüřmek dileęiyle hepinize mutlu günler dilerim.

Caching Mekanizmasını Anlamak - 1 - 21 Ocak 2005 Cuma

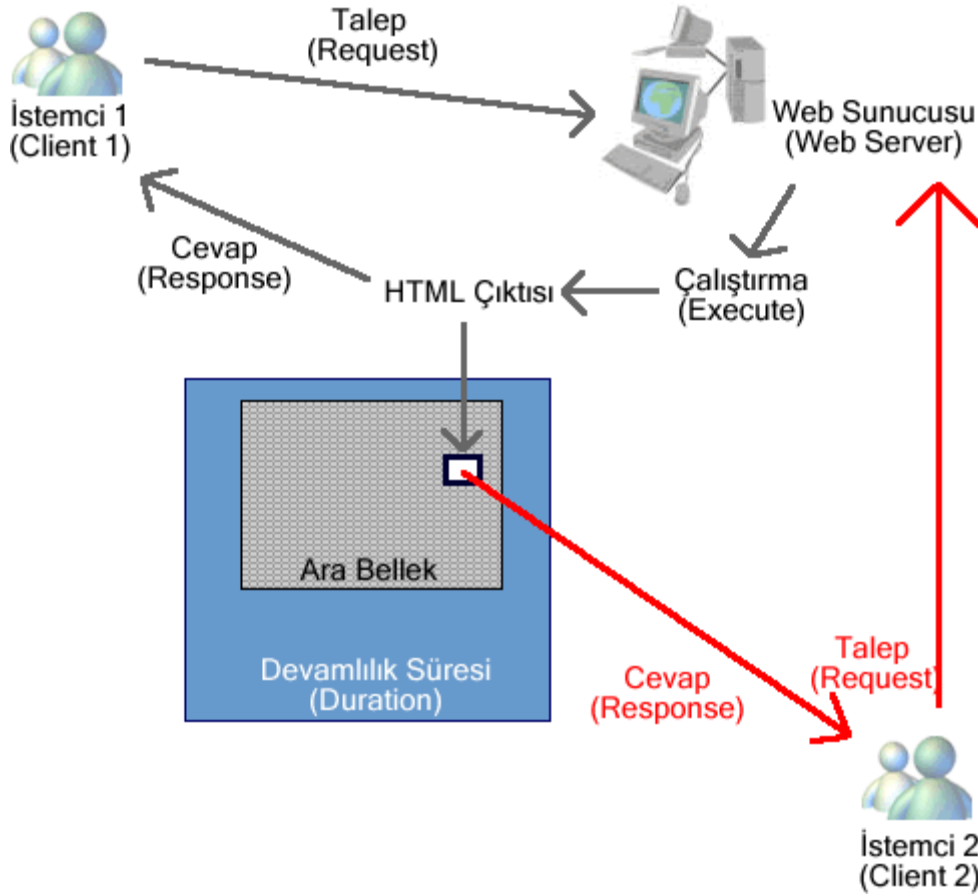
asp.net, caching,

Değerli Okurlarım, Merhabalar.

Bu makalemiz ile birlikte, web sayfalarının istemcilere daha hızlı bir şekilde ulaştırılmasında kullanılan tekniklerden birisi olan Caching (Ara Belleğe Alma) mekanizmasını incelemeye başlayacağız. Akıllıca kullanıldığı takdirde web uygulamalarında istemcilere nazaran göreceli olarak performans artışına neden olan Caching (Ara Belleğe Alma) mekanizması, teorik olarak bir web sayfasının tamamının ya da bir parçasının ara belleğe alınarak belli bir süre boyunca burada tutulması prensibini temel alarak çalışır. Asp.Net uygulamaları söz konusu olduğunda bir sayfanın tamamını, belli bir veri kümesini veya sayfa üzerindeki herhangi bir kontrolü ara belleğe alabiliriz. Buna göre Asp.Net uygulamalarındaki Caching (Ara Belleğe Alma) mekanizması aşağıdaki üç farklı tekniği destekler.

Asp.Net Uygulamaları İçin Caching (Ara Belleğe Alma) Teknikleri
1 - Output Caching (Çıktının Ara Belleğe Alınması)
2 - Data Caching (Verilerin Ara Belleğe Alınması)
3 - Fragment Caching (Parçaların Ara Belleğe Alınması)

Output Caching tekniğinde, bir aspx sayfasının tüm içeriği ara belleğe alınır ve belirli bir süre boyunca burada tutulur. Bu mekanizmanın çalışma biçimini aşağıdaki şekil ile daha kolay irdeleyebiliriz. Bu çalışma sistemi temel olarak diğer Caching mekanizmalarında da aynı şekilde işlemektedir. Değişen sadece ara belleğe alınan HTML görüntüsünün içeriği olacaktır.



İlk olarak, birinci istemci web sunucusundan bir aspx sayfasını talep eder. Web sunucusu sayfanın ilişkili kodlarını çalıştırarak bir çıktı üretir ve bu sayfa için Output Caching (Çıktının Ara Belleğe Alınması) aktif ise üretilen HTML sonuçlarını istemciye gönderir. Hemen ardından gönderilen HTML içeriğini belirtilen süre boyunca(duration) tutmak üzere sunucu üzerindeki ara belleğe alır.

Şimdi ikinci bir istemcinin devamlılık süresi (Duration) içerisindeyken aynı sayfayı sunucudan talep ettiğini düşünelim. Bu durumda web sunucusu, talep edilen sayfanın Cache (Ara Bellek) üzerinde olup olmadığına bakar. Eğer ikinci kullanıcı talebini, Devamlılık Süresi (Duration) içerisindeki bir zaman diliminde iletmış ise, web sunucusu istemciye sayfanın ara bellekteki hazır halini gönderir. Dolayısıyla sayfanın üretilmesinde çalıştırılan arka plan kodlarının hiç biri tekrardan yürütülmez. Bu elbetteki ikinci kullanıcının talep ettiği sayfa için daha kısa sürede cevap almasını sağlar.

Output Caching mekanizmasını bir aspx sayfasına uygulayabilmek için tek yapılması gereken OutputCache direktifinin sayfanın aspx kodlarının olduğu kısma eklemek yeterlidir.

```
<%@ OutputCache Duration="300" VaryByParam="None"%>
```

Burada, Duration özelliği ile sayfanın ilk talep edilışinden sonra 300 saniye (5 dakika) süre ile ara bellekte tutulacağını bildirmiş oluyoruz. VaryByName parametresini daha sonra inceleyeceğiz.

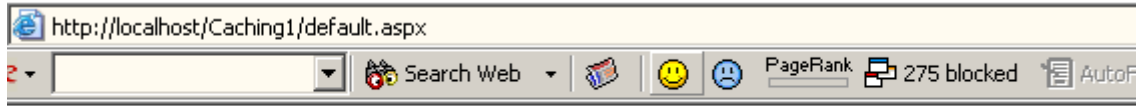
Şimdi dilerseniz, kısa bir örnek ile Output Caching mekanizmasının etkilerini incelemeye çalışalım. Basit bir web uygulaması geliştireceğiz. Bu uygulamada, Sql Sunucusunda yer alan Pubs isimli veri tabanındaki Authors isimli tabloya ait verileri bir DataGrid kontrolü içerisinde aktaracağız.

default.aspx sayfamızın kodları;

```
private void Baglan()
{
    con=new SqlConnection("data source=localhost;initial catalog=pubs;integrated
security=SSPI");
    cmd=new SqlCommand("SELECT * FROM authors",con);
    con.Open();
}
private void Doldur()
{
    dr=cmd.ExecuteReader(CommandBehavior.CloseConnection);
    dgVeriler.DataSource=dr;
    dgVeriler.DataBind();
    dr.Close();
}

private void Page_Load(object sender, System.EventArgs e)
{
    Baglan();
    Doldur();
}
```

Uygulama çalıştığında ve default.aspx sayfasının çıktısı üretildiğinde bu sayfanın son halinin bir kopyasında ara belleğe alınır. Yeni koypa ara bellekte 300 saniye (5 dakika) boyunca kalacaktır. Aynı sayfayı başka bir tarayıcı' da açtığımızda ya da aynı tarayıcı penceresinde iken ileri geri gittiğimizde sayfanın ara bellekte bulunan halini elde ederiz. Ancak bu örnekte dikkat edilmesi gereken bir nokta vardır. Bu durumu simule etmek için sayfayı refresh edelim. Bu durumda sayfanın içeriği aşağıdakine benzer olacaktır.



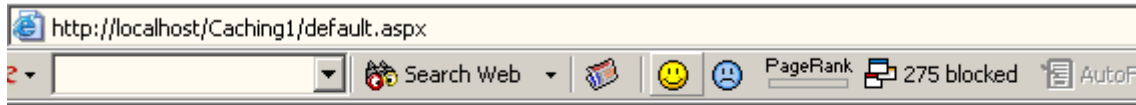
au_id	au_lname	au_fname	phone	address	city	state	zip	contract
172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd.	Menlo Park	CA	94025	True
213-46-8915	Green	Marjorie	415 986-7020	309 63rd St. #411	Oakland	CA	94618	True

Daha sonra Duration süresi dolmadan önce ilk satırın değerini Sql Sunucusu üzerinden değiştirelim. Bu durumda sayfada Output Caching uygulanmamış olsaydı, sayfayı bir sonraki talep edişimizde Load metodu çalışacak ve tablonun en güncel hali kullanıcıya sunulacaktı. Oysaki şimdi sayfayı talep ettiğimizde, yapılan değişikliğin görünmediğini farkederez.

Değişiklik;

au_id	au_lname	au_fname	phone	address
172-32-1176	White White	Johnson	408 496-7223	10932 Bigge Rd.
213-46-8915	Green	Marjorie	415 986-7020	309 63rd St. #411
228-05-7766	Green	Cheryl	415 548-7722	590 Davis Ln.

Yeni bir tarayıcı ile aynı sayfayı talep ettiğimizde;



au_id	au_lname	au_fname	phone	address	city	state	zip	contract
172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd.	Menlo Park	CA	94025	True
213-46-8915	Green	Marjorie	415 986-7020	309 63rd St. #411	Oakland	CA	94618	True

Bu işleyiş bazen büyük bir risk olabilir. Örneğin, sayfanın daha dinamik olduğu durumlarda ve hatta olay metodlarına cevap vermesi gerektiği durumlarda sayfanın tamamının ara belleğe alınması, aslında istemcilere sayfanın en güncel halinin gönderilmesi demek değildir. Bu gibi durumlarda çoğunlukla ya veri kümelerinin ya da sayfa üzerindeki belirli kısımların (kontrollerin) ara belleğe alınması tekniği tercih edilir. Bu teknikleri bir sonraki makalemizde inceleyeceğiz.

Bir sayfanın tamamını ara belleğe aldığımızda, sayfanın arabellekte yer alacak birden fazla kopyasına ihtiyaç duyduğumuz durumlar söz konusu olabilir. Çoğunlukla QueryString kullanımı sırasında başka sayfalara çeşitli parametreleri ve değerlerini göndeririz. İşte VaryByParam özelliği sayesinde sayfanın, gönderdiğimiz her parametre için ayrı ayrı veya sadece belirli parametreler için ayrı ayrı kopyalarını ara bellekte tutabiliriz.

Bu durumu daha yakından inceleyebilmek için pubs veritabanındaki titles tablosunu kullanacağımız bir örnek geliştirelim. Uygulamanın default.aspx sayfasında bu kez titles tablosunda yer alan her bir satır için title alanının değerlerini göstereceğiz. Kullanıcı her hangibir başlığa tıkladığında bu kitap ile ilgili detaylı bilgilerin olduğu başka bir sayfaya(detay.aspx) gidecek. Detay sayfası kitabın Primary Key değerini (title_id) QueryString parametresi olarak alacak. Bu durumda, detay sayfası için ara belleğe alma işlemini gerçekleştirebiliriz. Konuyu daha iyi anlayabilmek için örnek üzerinden adım adım gidelim. İlk olarak ana sayfamızın (default.aspx) aspx içeriğini aşağıdaki gibi değiştirmeliyiz.

```
<asp:DataGrid id="dgVeriler" style="Z-INDEX: 101; LEFT: 56px; POSITION: absolute; TOP: 88px" runat="server" AutoGenerateColumns="False" Height="160px" Width="264px" BorderColor="#DEBA84" BorderStyle="None" BorderWidth="1px" CellSpacing="2" BackColor="#DEBA84" CellPadding="3">
<FooterStyle ForeColor="#8C4510" BackColor="#F7DFB5"></FooterStyle>
<SelectedItemStyle Font-Bold="True" ForeColor="White" BackColor="#738A9C"></SelectedItemStyle>
<ItemStyle ForeColor="#8C4510" BackColor="#FFF7E7"></ItemStyle>
<HeaderStyle Font-Bold="True" ForeColor="White" BackColor="#A55129"></HeaderStyle>
<PagerStyle HorizontalAlign="Center" ForeColor="#8C4510" Mode="NumericPages"></PagerStyle>
<Columns>
<asp:HyperLinkColumn DataNavigateUrlField="title_id" DataNavigateUrlFormatString="Detay.aspx?title_id={0}" DataTextField="title" HeaderText="Başlık"></asp:HyperLinkColumn>
</Columns>
</asp:DataGrid>
```

Bu değişiklik sayesinde, dataGrid kontrolümüzden Detay.aspx sayfasına title_id alanının değerini parametre olarak gönderebileceğiz. Şimdi Detay.aspx sayfamızın kodlarını aşağıdaki gibi geliştirelim.

```
private void Baglan(string Id)
{
    con=new SqlConnection("data source=localhost;initial catalog=pubs;integrated security=SSPI");
    cmd=new SqlCommand("SELECT title_id,title,price,pubdate FROM titles WHERE title_id='"+Id+"' ",con);
    con.Open();
}
```

```

}
private void Doldur()
{
    dr=cmd.ExecuteReader(CommandBehavior.CloseConnection);
    dgDetaylar.DataSource=dr;
    dgDetaylar.DataBind();
    dr.Close();
}

private void Page_Load(object sender, System.EventArgs e)
{
    string id=Request.Params["title_id"].ToString();
    Baglan(id);
    Doldur();
}

```

Burada ara belleğe alma işlemini Detay.aspx sayfası üzerinde uyguluyoruz. Çünkü detay.aspx, dinamik olarak içeriği gelen parametre değerine göre değişen bir sayfadır. Biz gelen parametre değerine göre üretilen sayfaların html çıktılarını ara belleğe almak istiyoruz. İşte bunun için yine aspx kodlarının başına OutputCache direktifini aşağıdaki gibi eklememiz gerekiyor.

```
<%@ OutputCache Duration="300" VaryByParam="title_id"%>
```

Böylece detay.aspx sayfası her çalıştırıldığında gelen title_id değeri baz alınarak ara bellekte 300 saniye süreyle duracak olan html çıktıların oluşturulmasına imkan sağlıyoruz. Olayı şu şekilde irdelersek daha anlaşılır olacaktır;

Parametre tabanlı OutputCache işlemi	
İstemci title_id değeri BU1032 olan satırı detay.aspx sayfasından ister. Bu durumda detay.aspx sayfasının içeriği ara belleğe alınır. Bu sayfayı A olarak düşünelim.	detay.aspx in A koypası oluşturulur.
Başka bir İstemci title_id değeri MC2222 olan sayfayı talep eder. Bu durumda detay.aspx sayfasının yeni halinin kopyası ara belleğe alınır.	detay.aspx in B koypası oluşturulur.
Başka bir İstemci title_id değeri MC2222 olan sayfayı talep eder. Eğer duration süresi dolmadıysa talep edilen bu sayfanın çıktısı ara bellekte zaten olduğundan istemciye direkt olarak bu çıktı gönderilir.	detay.aspx in var olan B kopyası döner.

Başka bir İstemci title_id değeri BU1032 olan sayfayı talep eder. Eğer duration süresi dolmadıysa talep edilen bu sayfanın çıktısı ara bellekte zaten olduğundan istemciye direkt olarak bu çıktı gönderilir.	detay.aspx in var olan A kopyası döner.
<i>* Duration süreleri dolduğunda ise gelen talebe göre ara bellekteki sayfa çıktıları tekrardan oluşturulur.</i>	

Görüldüğü gibi VaryByParam özelliği ile, bir sayfanın kendisine QueryString ile gelen parametrelerinin değerine göre farklı ara bellek görüntülerini elde edebilmekteyiz. Bazı durumlarda talep edilen sayfaya birden fazla parametre gönderildiği veya hiç parametre gönderilmeden çağrıldığında olur. Bu durumu karşılamak için VaryByParam özelliğine * değerini atayabiliriz.

```
<%@ OutputCache Duration="300" VaryByParam="*" %>
```

Her ne kadar bir sayfanın Output Cache tekniği ile ara belleğe alınması avantajlı görünsede, özellikle olay kodlamalı sayfaların işleyişinde bu kullanım sorunlara yol açabilir. Her şeyden önce sayfa içinde postback' e neden olan kodlamalar var ise, sayfa ilk çağrıldıktan sonra ara belleğe alınacağından bu kod satırları duration süresi sonlanana kadar yürütülmeyecektir. Diğer yandan zaman zaman, sayfalarımızın içeriği dinamik olarak değişmek zorunda kalabilir. Bu gibi durumlarda da sayfanın tamamının ara belleğe alınması iyi bir yöntem değildir. Çözüm, sayfanın belirli parçalarının veya sayfadaki herhangi bir veri kümesinin(kümelerinin) ara belleğe alınmasıdır. Data Caching ve Fragment Caching tekniklerini bir sonraki makalemizde inceleyeceğiz. Tekrardan görüşünceye dek hepinize mutlu günler dilerim.

[Örnek uygulama için tıklayın.](#)

Caching Mekanizmasını Anlamak - 2 - 07

Şubat 2005 Pazartesi

asp.net, caching,

Değerli Okurlarım, Merhabalar

Hatırlayacağınız gibi bir önceki makalemizde, web uygulamalarında caching mekanizmasını incelemeye başlamış ve ara belleğe alma tekniklerinden Output Cache yapısını incelemiştik. Output Cache tekniğinde bir sayfanın tamamının HTML içeriği ara belleğe alınmaktaydı. Oysa çoğu zaman sayfamızda yer alan belirli veri kümelerinin ara bellekte tutulmasını isteyebiliriz. Örneğin, bir alışveriş sitesinin pek çok kısmı dinamik olarak değişebilirken satışı yapılan ürünlerin yer aldığı kategori listeleri çok sık değişmez. Hatta uzun süreler boyunca aynı kalabilirler. İşte böyle bir durumda sayfanın tamamını ara belleğe almak yerine sadece kategori listesini sunan veri kümesini ara belleğe almak daha mantıklıdır. Data Caching olarak adlandırılan bu teknikte çoğunlukla veri kümeleri ara belleğe alınır. Data Caching tekniğinde verileri ara belleğe almak için **System.Web.Caching** isim alanında yer alan **Cache** sınıfı ve üyeleri kullanılmaktadır.



Cache sınıfı **sealed** tipinde olup kendisinden türetme yapılmasına izin vermez. Bununla birlikte, her bir **Application Domain** için yalnız **bir Cache** nesne örneği oluşturulur ve kullanılır.

Cache sınıfına bir veri kümesini eklemek bu veriyi ara belleğe almak demektir. Bunun için aşağıdaki 4 aşırı yüklenmiş prototipe sahip olan ve bize pek çok imkan sağlayan **Insert** metodunu kullanabiliriz.

```
public void Insert(string key,object value);
```

```
public void Insert(string key, object value,CacheDependency dependencies);
```

```
public void Insert(string key,object value,CacheDependency dependencies,DateTime absoluteExpiration,TimeSpan slidingExpiration);
```

```
public void Insert(string key, object value, CacheDependency dependencies, DateTime absoluteExpiration,TimeSpan slidingExpiration, CacheItemPriority priority, CacheItemRemovedCallback onRemoveCallback);
```

Insert metodu ara belleğe alınan veri kümesinin durumuna ilişkin olarak çeşitli imkanlar sunar. Örneğin veri kümesinin ara bellekte ne kadar süre ile tutulacağı veya ne kadar süre bu veriye erişilmez ise ara bellekten silineceğinin belirlenmesi vb...Şimdi bu imkanları test etmeden önce basit olarak bir veri kümesini Cache nesnesine nasıl ekleyeceğimizi inceleyeceğiz.

```
private SqlConnection con;
private SqlCommand cmd;
private SqlDataAdapter da;
private DataTable dt;

/*Categories tablosundan CategoryName alanının değerlerini alıyoruz ve bir DataTable nesnesine aktarıyoruz.*/
private void KategorileriAl()
{
    con=new SqlConnection("data source=BURKI;initial catalog=Northwind;integrated security=SSPI");
    cmd=new SqlCommand("SELECT DISTINCT CategoryName FROM Categories",con);
    da=new SqlDataAdapter(cmd);
    dt=new DataTable();
    da.Fill(dt);
}

private void Page_Load(object sender, System.EventArgs e)
{
    //Güncel saat bilgisini label kontrolüne yazdırıyoruz.
    Label1.Text=DateTime.Now.ToLongTimeString();
    /*Eğer ara bellekte kategori isimli Cache nesnesinin içeriği null ise bu durumda verileri çekiyoruz ve DataTable içeriğini ara belleğe Cache sınıfının Insert metodu ile alıyoruz.*/
    if(Cache["kategori"]==null)
    {
        KategorileriAl();

        Cache.Insert("kategori",dt,null,DateTime.Now.AddMinutes(5),Cache.NoSlidingExpiration); /*
        Veriler 5 dakika süreyle ara bellekte saklanacak daha sonra ise silinecektir.*/
    }
    /*DataGrid kontrolüne veri kaynağı olarak ara bellekteki kategori isimli Cache nesnesinin içeriğini veriyoruz. Bunu yaparken uygun türe dönüştürme işlemini yapıyoruz.*/
    DataGrid1.DataSource=(DataTable)Cache["kategori"];
    DataGrid1.DataBind();
}
```

Uygulamamızda, Northwind veritabanında yer alan Categories isimli tablodan CategoryName değerlerini çekiyoruz ve elde ettiğimiz DataTable nesnesini 5 dakika süreyle ara bellekte tutumak üzere Insert metodu ile Cache nesnesine ekliyoruz. Uygulamayı ilk çalıştırışımızda aşağıdaki ekran görüntüsünü elde ederiz.

Address http://localhost/Caching2/default.aspx

23:38:35

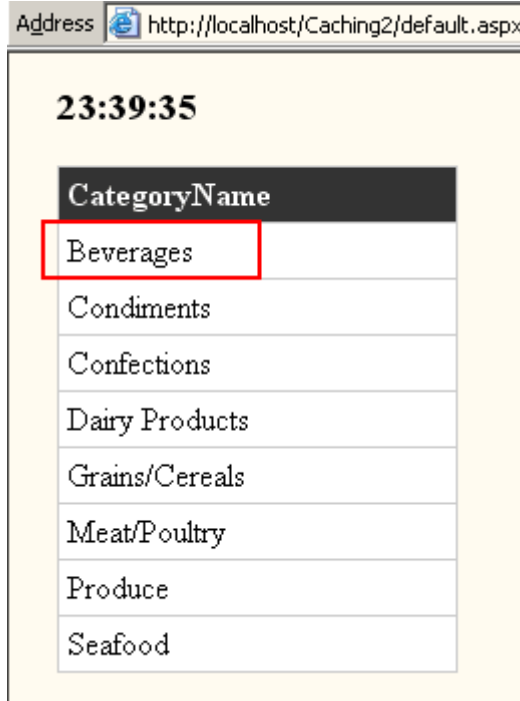
CategoryName
Beverages
Condiments
Confections
Dairy Products
Grains/Cereals
Meat/Poultry
Produce
Seafood

Sayfayı yenilediğimizde veya başka bir tarayıcı penceresinde tekrardan talep ettiğimizde sürenin düzenli olarak değiştiğini görürüz. Şimdi 5 dakika dolmadan tablodaki verilerde, örneğin Beverages alanının değerini [Beverages] olarak değiştirdiğimizi düşünelim.

Start Page | default.aspx | default.aspx.cs | Object Browse

CategoryID	CategoryName	Description
1	[Beverages]	Soft drinks, coffee
2	Condiments	Sweet and savory
3	Confections	Desserts, candies,
4	Dairy Products	Cheeses
5	Grains/Cereals	Breads, crackers, p
6	Meat/Poultry	Prepared meats
7	Produce	Dried fruit and bea
8	Seafood	Seaweed and fish

Daha sonra sayfayı tekrar çağıralım.



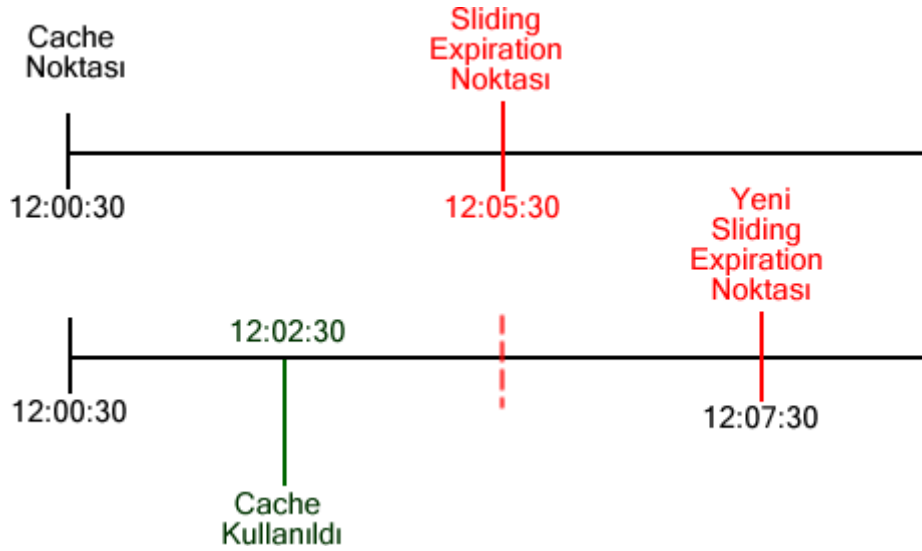
Görüldüğü gibi Beverages değerini değiştirmemize rağmen yapılan değişiklikler uygulamamıza yansımamıştır. Bu, verinin ara bellekten Cache nesnesi vasıtasıyla çekildiğinin bir ispatıdır. Veri ara belleğe alındıktan 5 dakika sonra aynı sayfa tekrar talep edilir ise bu kez verinin güncel hali ekrana gelecektir. Burada veriyi Cache nesnesine alırken kullandığımız Insert metodunda **Absolute Expiration (Tam Süre Sonu)** zamanı belirlenmiştir. Bu süre, verinin ara bellekten kesin olarak ne zaman atılacağını söylemektedir. Bununla birlikte dilersek ara bellekte bulunan veri kümesine olan erişim sıklığına göre bir **Sliding Expiration (Kayan Süre Sonu)** süreside belirleyebiliriz. Buna göre,

Ara bellekteki verilere belirtilen **Sliding Expiration** ile belirtilen süre zarfında erişilmez ise bu süre sonunda bellekten atılırlar. Eğer süre zarfı içinde ara bellekteki verilere sürekli erişiliyorsa Sliding Expiration süresi geçse dahi veriler ara bellekten atılmaz ve durumlarını korurlar.

Sürekli bellekte kalmak deyimi tabiki sistem kaynakları azalıp ara bellek verileri otomatik olarak atılınca veya web sunucusu herhangi bir neden ile restart olunca geçerli değildir. Şimdi dilerseniz Sliding Expiration durumunu incelemeye çalışalım. Bunun için tek yapmamız gereken örneğimizdeki Insert metodunu aşağıdaki ile değiştirmek olacaktır.

```
Cache.Insert("kategori",dt,null,Cache.NoAbsoluteExpiration,TimeSpan.FromMinutes(3));
```

Olayı daha iyi anlayabilmek için aşağıdaki şekli inceleyebiliriz.



Sayfa ilk talep edilip veriler ara belleğe alındığında, Sliding Expiration süresinin 5 dakika ilerisini gösterecek şekilde ayarlandığını düşünelim. Bu süre dolmadan önce ara bellekteki veri tekrardan talep edilirse, Cache nesnesinin içeriğinin boşaltılma süresi şekilden de görüldüğü gibi ilerki bir zamana (o anki andan 5 dakika sonrasına) sarkacaktır.

Insert metodunun parametrelerine dikkat edecek olursanız, ara bellekteki verilerin durumlarının **CacheDependency** sınıfına ait nesne örnekleri ile başka bir nesneye bağlı olabileceğini görürsünüz. Bu bağımlılıkta çoğunlukla fiziki dosyalar göz önüne alınır. Örneğin, bir XML dosyasındaki veriyi ara belleğe alarak kullandığımızı düşünelim. Bu veriler pekala güncel haber başlıklarını veya bir alışveriş sitesindeki ürünlerin kategorilerini gösterebilir. XML dosyasında meydana gelecek olan güncellemeleri anında ara belleğe yansıtmak için, Cache nesnesini bu XML dosyasına bağımlı hale getirebiliriz. Şimdi bunun nasıl yapılabileceğini incelemeye çalışalım. Yukarıda geliştirdiğimiz örneğimizdeki kodlarımızı aşağıdaki gibi değiştirelim.

Kategoriler.xml dosyasının içeriği;

```
<?xml version="1.0" encoding="utf-8" ?>
<Kategoriler>
  <Tipi>Muzik CD</Tipi>
  <Tipi>Kitap</Tipi>
  <Tipi>Film DVD</Tipi>
  <Tipi>Muzik DVD</Tipi>
  <Tipi>Elektronik Eşyalar</Tipi>
  <Tipi>Bilgisayar</Tipi>
  <Tipi>Laptop</Tipi>
</Kategoriler>
```

default.aspx

```
private SqlConnection con;
```

```

private SqlCommand cmd;
private SqlDataAdapter da;
private DataTable dt;
private DataSet ds;

/*KategoriAIXML metodu Kategoriler.xml dosyası içinden verileri alır ve DataSet' e yükler.*/
private void KategoriAIXML()
{
    ds=new DataSet();
    ds.ReadXml(Server.MapPath("Kategoriler.xml"));
}

private void Page_Load(object sender, System.EventArgs e)
{
    //Güncel saat bilgisini label kontrolüne yazdırıyoruz.
    Label1.Text=DateTime.Now.ToLongTimeString();
    if(Cache["kategori"]==null)
    {
        KategoriAIXML();
        /* Cache nesnesine DataSet içerisindeki xml dosyasından okunan içeriğe ait veri
        kümesi yüklenir. Bu yükleme işlemi yapılırken bir CacheDependency nesnesi ile Cache
        nesnesinin içeriğinin güncelliği belirtilen XML dosyasına bağlanır.*/
        Cache.Insert("kategori",ds.Tables[0],new
        CacheDependency(Server.MapPath("Kategoriler.xml")));
    }
    /*DataGrid kontrolüne veri kaynagi olarak ara bellekteki kategori isimli Cache nesnesinin
    içeriğini veriyoruz. Bunu yaparken uygun türe dönüştürme işlemini yapıyoruz.*/
    DataGrid1.DataSource=(DataTable)Cache["kategori"];
    DataGrid1.DataBind();
}

```

Burada, Cache nesnesinin içeriğinin durumunu xml dosyamıza bağlayabilmek için CacheDependency sınıfına ait bir nesne örneği oluşturulmuştur. Burada CacheDependency sınıfına ait nesne örneği oluşturulurken yapıcı metoda xml dosyasının sanal adresi atanmıştır.

```
new CacheDependency(Server.MapPath("Kategoriler.xml"))
```

Uygulamamızı çalıştırdığımızda ilk olarak aşağıdaki ekran görüntüsünü elde ederiz.

Address  http://localhost/Caching2/default.aspx

01:13:05

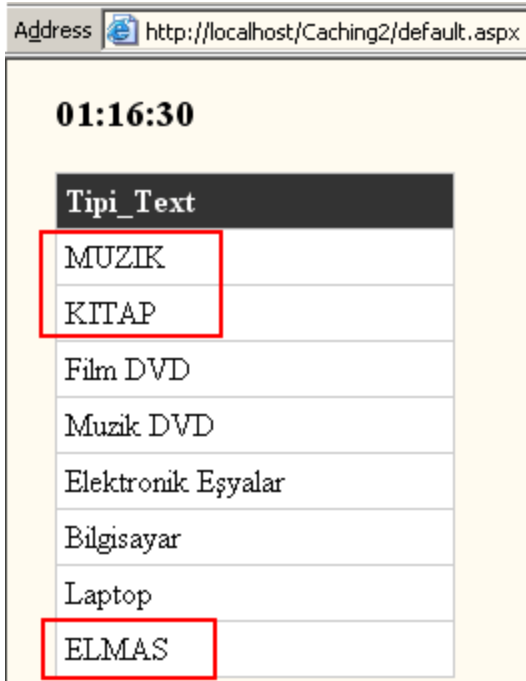
Tipi_Text
Muzik CD
Kitap
Film DVD
Muzik DVD
Elektronik Eşyalar
Bilgisayar
Laptop

Şimdi Kategoriler.xml dosyasının içeriğinde değişiklik yapalım ve bu değişiklikleri kaydedelim. Örneğin aşağıdaki gibi bir kaç elemanın bilgisini değiştirip yeni bir eleman ekleyelim.

ject Browser | Start Page | default.aspx | default.aspx.cs | **Kategoriler.xml**

```
<?xml version="1.0" encoding="utf-8" ?>
<Kategoriler>
  <Tipi>MUZIK</Tipi>
  <Tipi>KITAP</Tipi>
  <Tipi>Film DVD</Tipi>
  <Tipi>Muzik DVD</Tipi>
  <Tipi>Elektronik Eşyalar</Tipi>
  <Tipi>Bilgisayar</Tipi>
  <Tipi>Laptop</Tipi>
  <Tipi>ELMAS</Tipi>
</Kategoriler>
```

Bu değişikliklerden sonra sayfayı tekrardan talep edersek, Cache nesnesinin içerdiği verilerin son yapılan güncellemelere göre yenilendiğini görürüz. Cache nesnesindeki veriyi bir dosyaya yukarıdaki gibi bağımlı hale getirdiğimizde, sayfaya yapılan her talepte dosyanın içeriğinin değişip değişmediği kontrol edilir. Eğer bir değişiklik var ise, Cache nesnesinin içerdiği veri ara bellekten atılır. Biz uygulamamızda Cache nesnesini null olup olmadığına göre yükleme yaptığımız için dosyadaki güncelleme sonucu verinin bu son halini ara belleğe almış ve DataGrid içeriğini yenilemiş oluruz.



Asp.Net 2.0’ da bir Cache nesnesinin doğrudan Sql Server üzerindeki bir tabloya bağımlı hale getirilebilmesi ve dolayısıyla veritabanı içindeki bir tabloda meydana gelecek değişikliklerin Cache nesnesine anında yansıtılabilmesi içinde SqlCacheDependency sınıfına ait nesne örneklerinin kullanılabileceği öngörülmektedir. (* Bu durumu gelecek görsel derslerimizden birisinde incelemeye çalışacağız.)

Cache nesnelerinin bellekten atılması zamana, dosyaya bağlanabileceği gibi, sistem kaynaklarının azalması durumunda da gerçekleşen bir olaydır. Sistem kaynaklarının azalması ve ara bellekteki nesnelerin atılması gerektiği durumlarda Cache nesnelerinin sahip olduğu öncelikler göz önüne alınır. Her ne sebep ile olursa olsun bir Cache nesnesinin içeriği ara bellekten atıldığında otomatik olarak çalışmasını istediğimiz metodlar bildirebiliriz. Yani Callback metod tekniğini Cache nesneleri içinde kullanabiliriz. Örneğin ara bellekte tutulan bir nesnenin zaman aşımına uğraması nedeni ile silindiğinde otomatik olarak callback metodu devreye girerek güncel halinin tekrardan ara belleğe alınması sağlanabilir. Ya da Cache nesnesi Remove metodu ile açıkça ara bellekten atıldığı durumlarda Callback metodlarını çalıştırabiliriz. Burada bir Callback metodunun çağırılabilmesi için, **CacheItemRemovedReason** temsilcisi (delegate) tipinden bir nesne örneğinden faydalanılır. CacheItemRemovedReason temsilcisi aşağıdaki prototipe uyan metodlar işaret edebilir.

```
public delegate void CacheItemRemovedCallback(string anahtar,object deger,  
CacheItemRemovedReason sebep);
```

anahtar parametresi Cache nesnesinin key değerine karşılık gelir. deger parametresi ise ilgili Cache nesnesinin taşıdığı veriye sahiptir. Son parametre ise **CacheItemRemovedReason** numaralandırıcısı (enum sabiti) tipinden bir değerdir ve Callback metodunun çağırılma

nedenini bir başka deyişle Cache nesnesinin hangi sebepten dolayı ara bellekten atıldığının belirlenmesinde kullanılır. **CacheItemRemovedReason** numaralandırıcısının sahip olduğu değerler aşağıdaki tabloda belirtilmektedir.

CacheItemRemovedReason Numaralandırıcı Değeri	Açıklama
DependencyChanged	Herhangibir bağımlılık nedeni ile Cache nesnesinin içeriği ara bellekten atılmıştır. Örneğin Cache nesnesine bağladığımız XML dosyasında yapılan bir değişiklik buna neden olabilir.
Expired	Cache nesnesinin içeriği zaman aşımaları nedeni ile ara bellekten atılmıştır.
Removed	Cache nesnesinin içeriği Remove metodu ile açıkça ara bellekten atılmıştır. Yani programatik olarak Cache nesnesinin Remove metodu ilgili öğeye uygulanmıştır.
Underused	Sistem kaynaklarının azalması sonucunda web sunucusu, Cache nesnelerinin içeriğini sahip oldukları önem sıralarına göre ara bellekten atmaya başlamıştır.

Şimdi Callback tekniğinin nasıl uygulandığını basit bir örnek ile incelemeye çalışalım.

```
private CacheItemRemovedCallback delCallback=null;
private SqlConnection con;
private SqlCommand cmd;
private SqlDataAdapter da;
private DataTable dt;
private static string m_Sebe;
private static bool m_Durum=false;

private void KategorileriAl()
{
    con=new SqlConnection("data source=BURKI;initial catalog=Northwind;integrated
security=SSPI");
    cmd=new SqlCommand("SELECT DISTINCT CategoryName FROM Categories",con);
```

```

        da=new SqlDataAdapter(cmd);
        dt=new DataTable();
        da.Fill(dt);
    }

    private void GeriBildirimMetodu(string anahtar,object deger,CacheItemRemovedReason
    sebep)
    {
        m_Sebepe=sebep.ToString();
        m_Durum=true;
    }
    private void btnRemove_Click(object sender, System.EventArgs e)
    {
        if(Cache["Nesne1"]!=null)
        {
            Cache.Remove("Nesne1");
        }
    }
    private void btnCacheEkle_Click(object sender, System.EventArgs e)
    {
        delCallBack=new CacheItemRemovedCallback(this.GeriBildirimMetodu);
        if(Cache["Nesne1"]==null)
        {
            KategorileriAl();

            Cache.Insert("Nesne1",dt,null,DateTime.Now.AddSeconds(10),Cache.NoSlidingExpiration,C
            acheItemPriority.High,delCallBack);
            m_Durum=false;
        }
    }
    private void WebForm1_PreRender(object sender, System.EventArgs e)
    {
        DataGrid1.DataSource=Cache["Nesne1"];
        DataGrid1.DataBind();
        if(m_Durum)
            lblDurum.Text=m_Sebepe;
        else
            lblDurum.Text="";
    }

```

Yukardaki örnekte, Cache nesnesine DataTable içeriğini eklerken Insert metodunun aşağıdaki versiyonu kullanılmıştır.

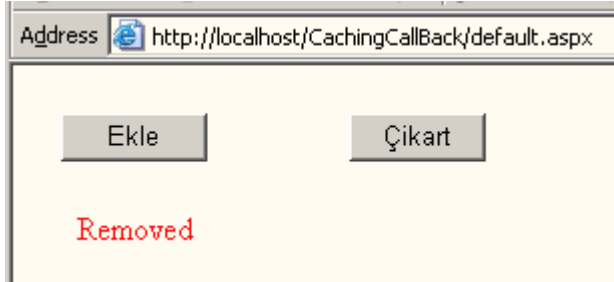
```

Cache.Insert("Nesne1",dt,null,DateTime.Now.AddSeconds(10),Cache.NoSlidingExpiration,C
acheItemPriority.High,delCallBack);

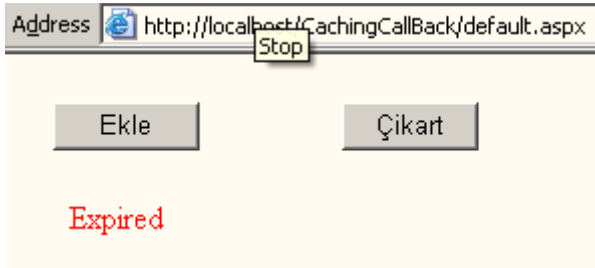
```

Burada delCallBack, Callback metodumuz olan GeriBildirimMetodunu işaret eden **CacheItemRemovedCallback** tipindeki temsilcimidir. Kullanıcı Remove işlemini

gerçekleştirdiğinde veya Cache nesnesi zaman aşımı nedeni ile ara bellekten atıldığında, geri bildirim metodu çalışacaktır. Örneğimizi çalıştırdığımızda ve 10 saniyelik zaman aşımı süresi dolmadan Ekle ve daha sonra Çıkart başlıklı butonlara tıkladığımızda aşağıdaki ekran görüntüsünü elde ederiz.



Eğer 10 saniye süresi dolduktan sonra Çıkart başlıklı butona basarsak Remove metodu çalıştırılmayacak ancak geri bildirim metodumuz çalışarak nesnenin ara bellekten atılma nedeni Expired olarak değişecektir.



CallBack tekniğinin uygulanışını daha iyi kavrayabilmek için, örnekleri debug modunda çalıştırıp breakpoint'ler vasıtasıyla kodları izlemenizi öneririm. Böylece geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşmek dileğiyle hepinize mutlu günler dilerim.

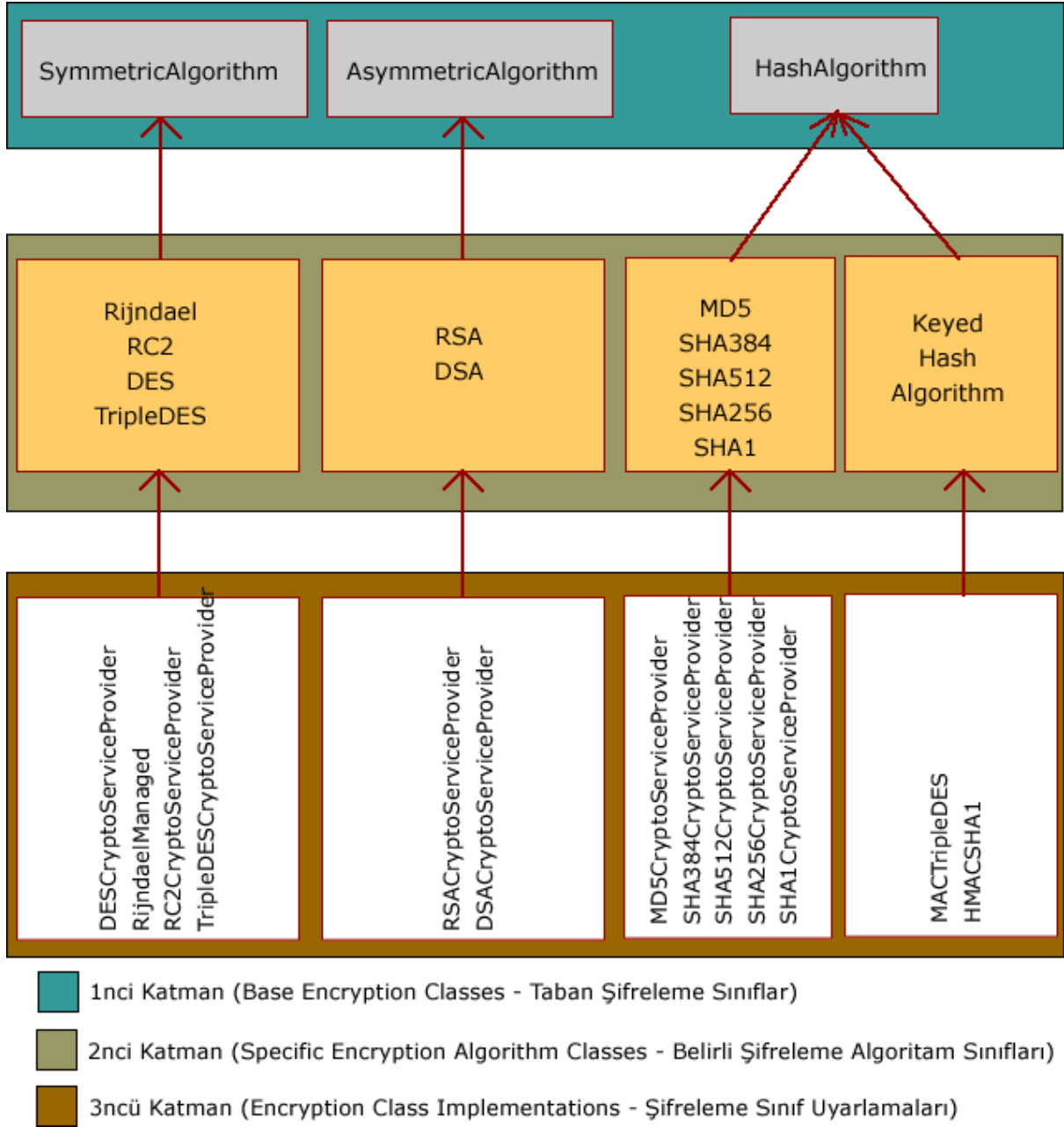
[Örnek uygulama için tıklayın.](#)

RijndaelManaged Vasıtasıyla Encryption(Şifreleme) ve Decryption(Deşifre) - 23 Şubat 2005 Çarşamba

Framework, cryptography, encryption, rijndael,

Değerli Okurlarım, Merhabalar

Bu makalemizde, Rijndael Algoritmasını kullanan Managed tiplerden **RijndaelManaged** sınıfı ile şifreleme (encryption) ve deşifre etme (decryption) işlemlerinin nasıl gerçekleştirilebileceğini incelemeye çalışacağız. Konu ile ilgili örneklerimize geçmeden önce .Net Framework içerisinde yer alan Cryptography mimarisinde kısaca bahsetmekte yarar olduğunu düşünüyorum. Aşağıdaki şekil, .Net Framework'te **System.Security.Cryptography** isim alanında yer alan şifreleme hiyerarşisini göstermektedir. Framework mimarisinde şifreleme sistemi ilk olarak üç ana katmandan oluşur. İlk katmanda taban sınıflar (base classes) yer alır. Bunlar SymmetricAlgorithm, AsymmetricAlgorithm ve HashAlgorithm sınıflarıdır. Bu sınıflar kendisinden türeyen ikinci katman sınıfları için temel ve ortak şifreleme özelliklerini içerirler.



SymmetricAlgorithm sınıfını kullanan şifreleme mekanizmalarında herhangi bir anahtar ile şifrelenen veriler, deşifre edilmek istendiklerinde yine aynı anahtarı kullanırlar. Bu özellikle internet gibi herkesin kullanımına açık olan ortamlarda güvenlik açısından tehlike yaratabilir. Nitekim anahtarın herhangi bir şekilde ele geçirilmesi, şifrelenen verinin çözülmesi için yeterli olacaktır. Diğer yandan bu tekniğe göre geliştirilen algoritmalar hızlı ve performanslı çalışırlar.



SymmetricAlgorithm katmanından türeyen Encryption sınıfları ile uygulanan şifreleme algoritmalarında veriyi şifrelerken kullandığımız key(anahtar) ve IV(vektör) değerleri, aynı veriyi deşifre ederken de gereklidir.

SymmetricAlgorithm yapısını kullanan şifreleme mimarilerinin neden olduğu güvenlik sorununun çözümü için AsymmetricAlgorithm taban sınıfı (base class) geliştirilmiştir. Bu mekanizmada veri şifreleneceği zaman public bir anahtar kullanılır. Bu anahtarın herhangi bir şekilde ele geçirilmesi, verinin deşifre edilebilmesi için yeterli değildir. Nitekim verinin deşifre (decryption) edilebilmesi için karşı tarafın private bir anahtara gereksinimi vardır. Bu avantajının yanında AsymmetricAlgorithm mekanizması SymmetricAlgorithm mekanizmasına göre daha yavaş çalışmaktadır.



AsymmetricAlgorithm katmanından türeyen Encryption sınıfları ile uygulanan şifreleme (encryption) algoritmalarında veriyi şifrelerken public bir key kullanırken, deşifre (decryption) işlemi sırasında farklı olan private key kullanılır.

Birinci katmanda yer alan taban sınıflar, abstract niteliktedir. Dolayısıyla kendisinden türeyen şifreleme sınıflarının içermesi ve uygulaması zorunlu olan üyeler içerirler. Bildiğiniz gibi abstract sınıflardan nesne örnekleri üretilemez. Ancak taban sınıfların static Create metotları yardımıyla bu sınıfları da şifreleme mekanizmalarında kullanabiliriz. İkinci katmanda yer alan sınıflar ise, özellikle belirli şifreleme algoritmalarını işaret ederler. Örneğin bu gün işleyeceğimiz Rijndael algoritması 256 bitlik bir anahtar (key) ile şifreleme (deşifre etme) sağlar. Buradaki sınıflar, taban sınıflardan (base classes) türemiştir ve abstract sınıflardır.

Asıl şifreleme metotlarını ve üyelerini bizim için kullanışlı hale getiren sınıflar üçüncü katmanda yer alırlar. Burada dikkat edecek olursanız bazı sınıflar ServiceProvider kelimesi ile bitirler. Bu sınıflar Windows'un CryptoApi kütüphanesini kullanan sınıflardır. Diğer yandan Managed kelimesi içeren sınıflar (örneğin RijndaelManaged) özellikle .net için geliştirilmiş yönetimsel uyarlamalardır (Managed Implementations). Buradaki sınıflar sealed olarak tanımlanmıştır. Yani kendilerinden türetme yapılamaz. Buna rağmen eğer istersek ikinci veya birinci katman sınıflarını kullanarak kendi şifreleme algoritma sınıflarımızı veya uyarlamalarımızı geliştirebiliriz.

Peki bir veri kümesini şifrelemek için yukarıdaki katmanları ve içeriklerini nasıl kullanabiliriz. Bu makalemizde biz örnek olarak Rijndael algoritmasını kullanan iki örnek geliştireceğiz. .Net içerisinde verileri şifrelemek için izleyeceğimiz yolda anahtar nokta CryptoStream sınıfıdır. Bu sınıfa ait nesne örnekleri yardımıyla verileri stream bazlı olarak, istenen algoritmaya göre şifreleyebilir ya da deşifre edebiliriz. CryptoStream sınıfından bir nesne örneğini aşağıdaki yapıcı metot (constructor) yardımıyla oluşturabiliriz.

```
public CryptoStream(Stream stream, ICryptoTransform transform, CryptoStreamMode mode);
```

Bu metodun ilk parametresine dikkat edecek olursanız bir Stream nesnesidir. CryptoStream sınıfı şifrelenecek veya deşifre edilecek verilerin işlenmesi sırasında Stream nesnelerini kullanılır. Dolayısıyla bellek üzerinde tutulan verileri, fiziki dosyalarda tutulan verileri veya network üzerinden akan verileri ilgili stream nesneleri yardımıyla (MemoryStream, FileStream, NetworkStream vb...) CryptoStream sınıfına ait bir nesne örneğine aktarabiliriz. İkinci parametre ise Stream üzerindeki verinin hangi algoritma ile şifreleneceğini (deşifre edileceğini) belirlemek üzere kullanılır. Bu parametre ICryptoTransform ara yüzü tipinden bir nesne örneğidir. Üçüncü parametre ise Stream' e yazma veya stream' den okuma yapılacağını belirtir.

Stream üzerindeki veriyi şifreleyeceğimiz zaman üçüncü parametre Write değerini alırken, deşifre işlemlerinde Read değerini alır. Karışık gibi görünmesine rağmen örneklerimizden de göreceğiniz gibi, şifreleme(deşifre etme) işlemleri sanıldığı kadar zor değildir. Dilerseniz vakit kaybetmeden örneklerimize geçelim. İlk örneğimizde metin tabanlı bir dosya içeriğini FileStream nesnesinden faydalanarak okuyor ve CryptoStream sınıfına ait nesne örneği yardımıyla şifreleyerek bir dosyaya yazıyoruz. Daha sona ise şifrelenen bu dosyayı tekrardan çözüyoruz.

```
using System;  
using System.IO;  
using System.Security.Cryptography;
```

```
namespace CryptoStreamS1
```

```
{
```

```
    class Class1
```

```
    {
```

```
        [STAThread]
```

```
        static void Main(string[] args)
```

```
        {
```

```
            #region Dosya Şifrelemesi (Rijndael algoritması ile)
```

```
            // İlk olarak şifrelemek istediğimiz dosya için bir stream oluşturuyoruz.
```

```
            FileStream fs=new
```

```
FileStream(@"SifreliDosya.txt",FileMode.OpenOrCreate,FileAccess.Write);
```

```
            /* Kullanacağımız şifreleme algoritmasını uygulatabileceğimiz managed Rijndael sınıfına ait nesne örneğimizi tanımlıyoruz. Şifreleme algoritması olarak Rijndael tekniğini kullanıyoruz. */
```

```
            RijndaelManaged rm=new RijndaelManaged();
```

```
            rm.GenerateKey(); // Algoritma için gerekli Key üretiliyor.
```

```
            /* Şimdi algoritma için gerekli key ve vektör değerlerini üretiyoruz. Burada kullanılan şifreleme algoritması simetrik yapıda olduğundan şifrelenen verinin açılması için (decrypting) aynı key ve vektör değerine ihtiyacımız var. Bu nedenle bunları bir byte dizisinde tutuyoruz. */
```

```
            // Elde ettiğimiz key değerini bir byte dizisine aktarıyoruz.
```

```
            byte[] k=new byte[rm.Key.Length];
```

```

for(int i=0;i<rm.Key.Length;i++)
{
    Console.Write(rm.Key[i]);
    k[i]=rm.Key[i];
}

Console.WriteLine();

rm.GenerateIV(); // Algoritma için gerekli IV vektör değeri üretiliyor.
byte[] v=new byte[rm.IV.Length];

// Elde ettiğimiz Vektör değerini bir byte dizisine aktarıyoruz.
for(int i=0;i<rm.IV.Length;i++)
{
    Console.Write(rm.IV[i]);
    v[i]=rm.IV[i];
}

Console.WriteLine();

/* Belirlediğimiz şifreleme algoritmasını kullanarak, stream üzerinde şifrelemeyi
yapacak CryptoStream nesnemizi oluşturuyoruz. Şifrelemeyi oluşturmak için
RijndaelManaged sınıfından örneklendirdiğimiz nesnemizin CreateEncryptor metodunu
kullanıyoruz. Oluşturulan şifreli dökümanı ilgili stream'e yazmak
istediğimizdenCryptoStreamMode olarak Write değerini seçiyoruz.*/
CryptoStream cs=new
CryptoStream(fs,rm.CreateEncryptor(),CryptoStreamMode.Write);

/*Şimdi şifrelenecek olan byte dizisini almak üzere dosyamız için bir akım
oluşturuyoruz. Nitekim CryptoStream' in aşağıda kullanılan aşırı yüklenmiş versiyonu ilk
parametre olarak şifrelenecek veri yapısını bir byte dizisi halinde alıyor.*/
FileStream fs2=new FileStream(@"Dosya.txt",FileMode.Open);
// dosyanın içeriğini byte dizisine aktarıyoruz.
byte[] veriler=new byte[fs2.Length];
fs2.Read(veriler,0,(int)fs2.Length);

// CryptoStream sınıfının write metodu ile dosya.txt' yi okuduğumuz byte dizisinin
içeriğini fs2 ile belirttiğimiz stream'e yazıyoruz.
cs.Write(veriler,0,veriler.Length);
fs2.Close();
cs.Close();
#endregion

#region Şifrelenmiş dosyanın örnek olarak ilk satırının decrypt edilerek okunması.

/* Bu kez işlemleri tersten yapıyoruz. İlk olarak şifrelenmiş ve decrypt edilmek istenen
stream nesnesinin oluşturuyoruz. Ardından bu stream' deki veriye Rijndael alogirtmasını
uygulayarak Decrypting yapıyoruz. */

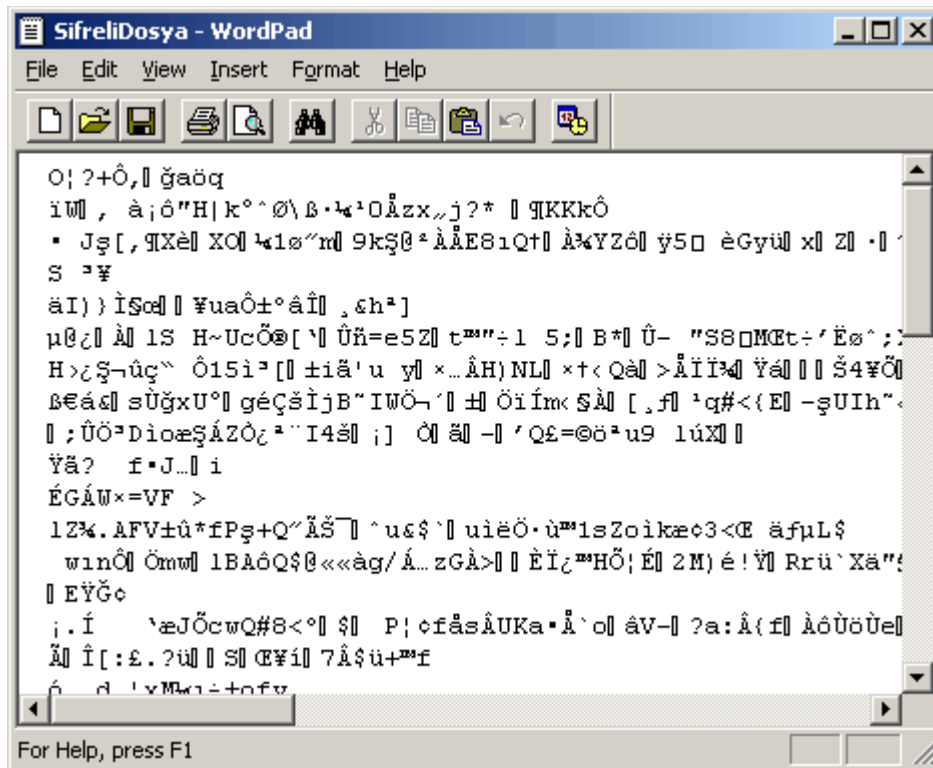
```

```

        FileStream fsSifreliDosya=new
FileStream(@"SifreliDosya.txt",FileMode.Open,FileAccess.Read);
        RijndaelManaged rm2=new RijndaelManaged();
        //simetrik algoritma kullandığımız için decrypting içinde aynı key ve vektör değerlerini
kullanmamız gerekiyor.
        rm2.Key=k;
        rm2.IV=v;
        CryptoStream cs2=new
CryptoStream(fsSifreliDosya,rm2.CreateDecryptor(),CryptoStreamMode.Read);
        StreamReader sr=new StreamReader(cs2);
        string satir=sr.ReadLine();
        Console.WriteLine(satir);
        #endregion
    }
}
}

```

Uygulamayı çalıştırdığımızda orijinal içerikli dosyanın aşağıdaki gibi şifrelendiğini görürüz.



İkinci örneğimizde ise, MemoryStream nesnesinden yararlanacağız. Bu kez, bir veri tablosundan çektiğimiz belli bir alanı bellek üzerinden şifreliyor ve daha sonra şifrelenen verinin orijinal içeriğini elde edecek şekilde deşifre işlemini uyguluyoruz.

```

using System;
using System.IO;
using System.Data;

```

```

using System.Data.SqlClient;
using System.Security.Cryptography;

namespace CryptoStreamS2
{
    class Kriptografi2
    {
        [STAThread]
        static void Main(string[] args)
        {
            #region verinin şifrelenmesi
            /* Öncelikle şifrelemek istediğimiz veriyi elde ediyoruz. Örnek olarak SQL
            Sunucusundaki Ogrenciler tablosundan belirli bir alanı aldık. */
            SqlConnection con=new SqlConnection("data
            source=BURKİ;database=Work;integrated security=SSPI");
            SqlCommand cmd=new SqlCommand("SELECT AD FROM Ogrenciler WHERE
            OGRENCINO=1",con);
            con.Open();
            string sifrelenecekVeri=cmd.ExecuteScalar().ToString();
            con.Close();

            /* Şifrelenecek verinin herşeyden önce bir byte dizisi olarak ele alınması gerekiyor.*/
            byte[] sv=new byte[sifrelenecekVeri.Length];
            for(int i=0;i<sv.Length;i++)
            {
                sv[i]=(byte)sifrelenecekVeri[i];
            }

            /* Şifrelenecek veriyi belleğe yazacağız. Bu nedenle MemoryStream sınıfı tipinden bir
            nesne örneği oluşturduk*/
            MemoryStream ms=new MemoryStream();
            /* Şifreleme algoritması olarak Rijndael tekniğini sağlayan Managed nesne
            örneğimizi oluşturuyoruz.*/
            System.Security.Cryptography.RijndaelManaged rm=new RijndaelManaged();

            /* Şifreleme için gerekli anahtar ve vektör değerlerini elde ediyoruz.*/
            rm.GenerateKey();
            rm.GenerateIV();

            /* RijndaelManaged nesnesi tarafından üretilen anahtar ve vektör değerlerini byte
            dizilerine alıyoruz. Nitekim karşı tarafın şifrelenen veriyi çözebilmesi için bu anahtar ve
            vektör değerlerinin aynılarına ihtiyaçları olacaktır.*/
            byte[] anahtar=rm.Key;
            byte[] vektor=rm.IV;

            /* Veriyi belirttiğimiz algoritmaya göre şifreleyerek parametre olarak verilen stream' e
            ki burada MemoryStream' e yazmak için CryptoStream sınıfımızdan nesne örneğimizi
            oluşturuyoruz.*/

```



```

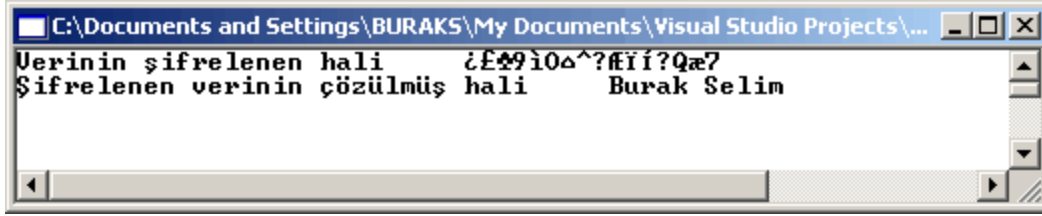
        CryptoStream cs=new
CryptoStream(ms,rm.CreateEncryptor(anahtar,vektor),CryptoStreamMode.Write);
        /* Veriyi şifreleyerek belleğe yazıyoruz. Başından sonuna kadar.*/
        cs.Write(sv,0,sv.Length);
        cs.FlushFinalBlock();

        Console.WriteLine("Verinin şifrelenen hali ");
        byte[] icerik=ms.ToArray(); /* Belleğe yazdığımız şifrelenmiş veriyi bir byte dizisine
alarak okuyor ve ekrana yazdırıyoruz.*/
        for(int i=0;i<icerik.Length;i++)
        {
            Console.WriteLine((char)icerik[i]);
        }
        Console.WriteLine();
        #endregion

        #region şifrelenen verinin çözülmesi
        /* Bellekte tutulan icerik değerini yani şifrelenmiş olan veriyi parametre alan stream
nesnemizi oluşturuyoruz.*/
        MemoryStream msCoz=new MemoryStream(icerik);
        RijndaelManaged rmCoz=new RijndaelManaged(); // Rijndael algoritmasını
kullanarak şifrelenen veriyi çözecek olan provider nesnemizi tanımlıyoruz.*/
        /* SymmetricAlgorithm söz konusu olduğundan RijndaelManaged sınıfına ait nesne
örneğin decryption işlemi için encrypt' te kullanılan key ve IV değerlerine ihtiyacımı var.*/
        rmCoz.Key=anahtar;
        rmCoz.IV=vektor;
        /* Bu kez CryptoStream nesnemiz stream' den okuduğu veri üzerinde Decypting
işlemini gerçekleştirecek. Bu nedenle Rijndael nesne örneğimizin CreateDecryptor
metodunu çağırıyoruz.*/
        CryptoStream csCoz=new
CryptoStream(msCoz,rmCoz.CreateDecryptor(anahtar,vektor),CryptoStreamMode.Read);
        byte[] cozulen=new byte[ms.Length]; // Çözülen veriyi tutacak bir byte dizisi
oluşturuyoruz.
        csCoz.Read(cozulen,0,icerik.Length); // Şifrelenen veriyi çözümleyerek okuyoruz.
        Console.WriteLine("Şifrelenen verinin çözülmüş hali ");
        /* Çözümlemiş veriyi son olarak ekrana yazdırıyoruz.*/
        for(int i=0;i<cozulen.Length;i++)
        {
            Console.WriteLine((char)cozulen[i]);
        }
        Console.ReadLine();
        #endregion
    }
}
}

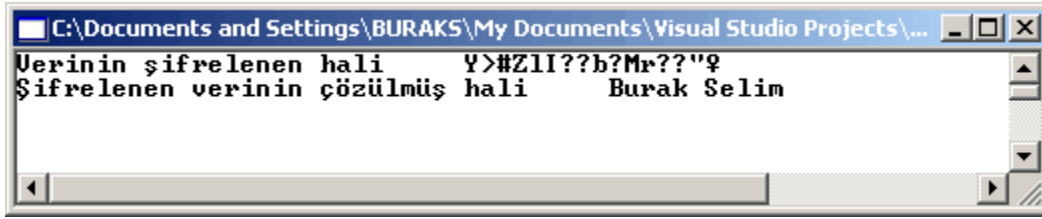
```

Uygulamamızı arka arkaya çalıştırdığımızda aşağıdakine benzer sonuçlar alırız. Dikkat ederseniz deşifre edilen veri her seferinde aynı olmasına rağmen, şifrelenen veri içeriği bir birlerinden farklıdır.



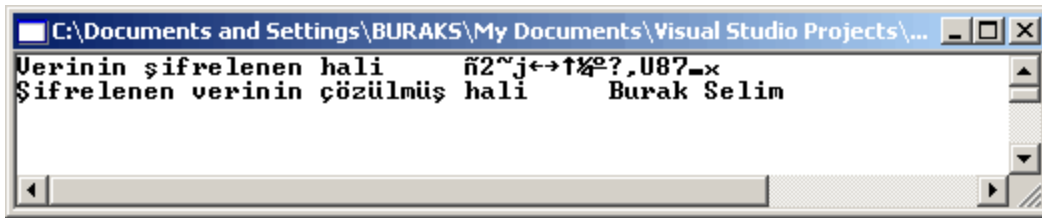
A screenshot of a Windows Notepad window. The title bar reads "C:\Documents and Settings\BURAKS\My Documents\Visual Studio Projects\...". The text inside the window is as follows:

```
Verinin şifrelenen hali      ÇE9i0Δ^?Æİİ?Qæ?
Şifrelenen verinin çözölmüş hali      Burak Selim
```



A screenshot of a Windows Notepad window. The title bar reads "C:\Documents and Settings\BURAKS\My Documents\Visual Studio Projects\...". The text inside the window is as follows:

```
Verinin şifrelenen hali      Y>#ZlI??b?Mr??"♀
Şifrelenen verinin çözölmüş hali      Burak Selim
```



A screenshot of a Windows Notepad window. The title bar reads "C:\Documents and Settings\BURAKS\My Documents\Visual Studio Projects\...". The text inside the window is as follows:

```
Verinin şifrelenen hali      ñ2~j↔↑¼º?„U87=x
Şifrelenen verinin çözölmüş hali      Burak Selim
```

Bu makalemizde kısaca Rijndael algoritmasını kullanan RijndaelManaged sınıfı ile şifreleme ve deşifre işlemlerini incelemeye çalıştık. İlerleyen makalelerimizde, AsymmetricAlgorithm tekniğinin nasıl uygulanabileceğini incelemeye çalışacağız. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

[Örnek kodlar için tıklayın.](#)

Bağlantısız Katmanda Concurrency Violation Durumu - 06 Mart 2005 Pazar

ado.net, concurrency, disconnected layer,

Değerli Okurlarım, Merhabalar.

Bağlantısız katman nesneleri ile çalışırken karşılaşılabileceğimiz problemlerden bir tanesi güncelleme işlemleri sırasında oluşabilecek DBConcurrencyException istisnasıdır. Bu makalemizde, bu hatanın fırlatılış nedenini inceleyecek ve alabileceğimiz tedbirleri ele almaya çalışacağız. Öncelikle istisnanın ne olduğunu anlamak ile işe başlayalım. Bir DataAdapter nesnesine ait Update metodu güncelleme işlemleri için Optimistic(iyimser) yaklaşımı kullanan sql sorgularını çalıştırıyorsa DBConcurrencyException istisnasının ortaya fırlatılması, başka bir deyişle Concurrency Violation (eş zamanlı uyumsuzluk) durumunun oluşması son derece doğaldır.



Optimistic yaklaşım modeli, Pessimistic yaklaşım modelinin aksine güncellenecek satırları kilitlemez. Buda sunucunun kilit açma, takip ve kapatma gibi işlemleri yapmaması dolayısıyla performansının artması anlamına gelir. Özellikle bağlantısız katman mimarisinde kullanılan optimistic yaklaşım modelinde tek sorun, güncelleme işlemlerini gerçekleştiren kullanıcıların bu işleri birbirlerinden habersiz şekilde yapmaları sonucu ortaya çıkabilecek durumlardır.

Örneğin belli bir satıra ait verileri güncellemek için kullanabileceğimiz aşağıdaki Sql sorgusunu ele alalım.

```
UPDATE MAILS SET AD=@AD,SOYAD=@SOYAD,EMAIL=@EMAIL WHERE  
ID=@ORGID AND AD=@ORGAD AND SOYAD=@ORGSOYAD AND  
EMAIL=@ORGEMAIL
```

Sorgumuz basitçe, MAILS isimli veritabanındaki AD, SOYAD ve EMAIL alanlarının değerlerini güncellemektedir. Bunu yaparkende optimistic (iyimser) yaklaşımını kullanır. Bu nedenle, Where koşulunda tabloya ait primary key alanı (ID) dahil olmak üzere tüm alanlar kullanılmaktadır. Böylece tüm alanların eşleştirme için kullanıldığı bir sorgu ortaya çıkar.

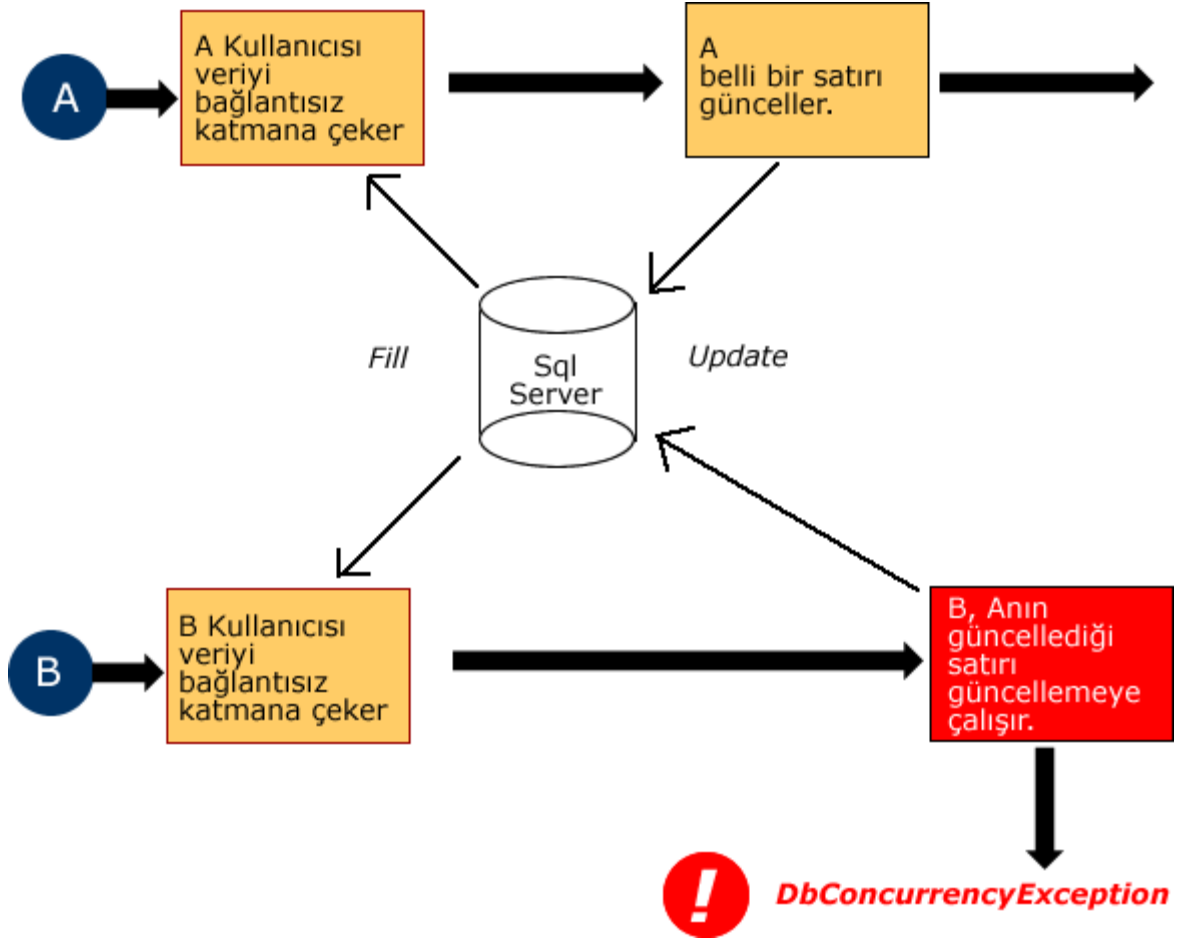


Optimistic yaklaşımda ele alınan yukarıdaki sorgu modeli için Sql Server ve benzeri veritabanı sistemlerinde daha etkili yöntemlerde vardır. Örneğin Sql üzerinde **timestamp** tipinden (yada **uniqueIdentifier** tipinden) alanlar kullanılabilir. Timestamp türünden olan alanlar satır üzerinde yapılacak herhangi bir güncelleme işleme sonrasında sistem tarafından otomatik olarak benzersiz bir karakter dizisi ile değiştirilen alanlardır. Böylece yukarıdaki sorgunun yaptığı işin aynısını aşağıdaki gibide yapabiliriz. (Buradaki Kontrol alanı tipi timestamp tipindendir.

Update Mails Set Ad=@Ad,Soyad=@Soyad,Email=@Email **Where Id=@OrgId And Kontrol=@Kontrol**

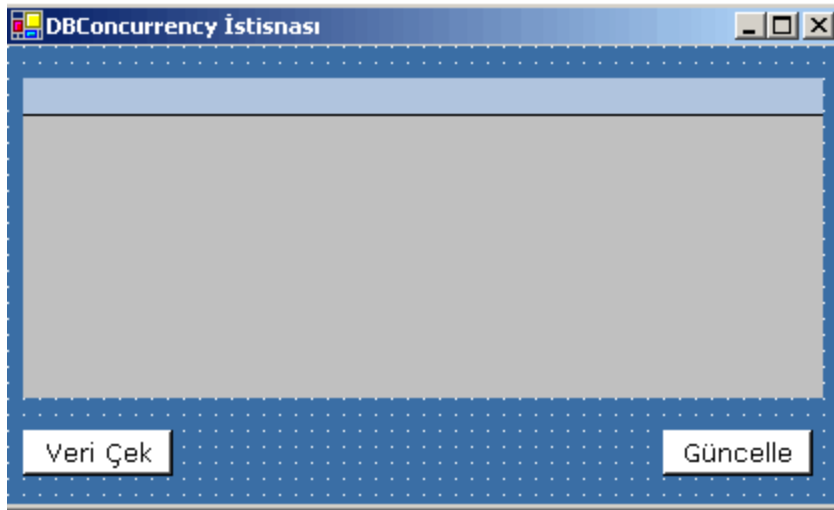
Bu ifadenin bize sağladığı en büyük avantaj elbetteki n sayıda alan içeren bir tabloda where ifadesinden sonra sadece iki alan kontrolü ile (primary key ve timestamp alanı) Concurrency Violation durumunu irdeleyebilecek olmamızdır. Biz makalemizde daha uzun olan yolu incelemeye çalışacağız. Lakin gerçek hayat modellerinde **timestamp** veya **uniqueidentifier** ve benzeri tipten alanların karşılaştırma işlemi için ele alınması daha doğru ve güçlü bir yaklaşım olacaktır.

Böyle bir sorgunun neden olacağı istisnai durumu anlayabilmek için aşağıdaki senaryoyu göz önüne almakta fayda olacağı inancındayım. Senaryomuzda en az iki kullanıcı rol almaktadır. Bu kullanıcılarımıza A ve B takma isimlerini verdiğimiz düşünelim. Her iki kullanıcıda database' den MAILS tablosundaki verileri bağlantısız katmana DataAdapter sınıfına ait nesne örneği vasıtasıyla almaktadır.



A ve B verileri çektikten sonra, A kullanıcısı herhangi bir satır üzerinde güncelleme işlemini uygular. Bu durumda DataAdapter nesnesinin UpdateCommand özelliğine karşılık gelen SqlCommand nesnesi, yukarıda yazdığımız sorguyu çalıştıracaktır. Bu sorguda, satırların orjinal değerleri ile veritabanındaki halleri aynı olacağından güncelleme işlemi başarılı bir şekilde gerçekleştirilecektir. Lakin B kullanıcısı şu anda, A'nın güncellemiş olduğu veri kümesinin eski haline bakmaktadır. Eğer B kullanıcısı, A kullanıcısının biraz önce güncellemiş olduğu satırı tekrar güncellemek isterse ne olacaktır?

İşte bu durumda, sorgu içindeki where koşuluna giren alan değerlerinin bağlantısız katmandaki orjinal halleri (*yani **DataRowVersion** numaralandırıcısı tipinden **Original** olan değerleri*), veritabanındaki tabloda az önce güncelleştirilmiş olan alanlara ait yeni değerler ile eşleşmeyeceğinden ilgili satır bulunamayacaktır. Bu da B kullanıcısının satırı update edememesine neden olur. Bu noktada CLR, DbConcurrencyException türünden bir istisnayı process içine fırlatacaktır. Dilerseniz bu hatayı basit bir uygulama yardımıyla elde etmeye çalışalım. Uygulamamız şimdilik sadece Update işlevini ele alacaktır. İlk olarak basit bir windows uygulaması açarak aşağıdakine benzer bir form ekranı oluşturalım.



Uygulamamız aşağıdaki field yapısına sahip olan ve Sql sunucusu üzerinde barındırdığımız MAILS tablosunu kullanacaktır. Tablomuzdaki ID alanı otomatik artan bir primary key olarak tanımlanmıştır.

Column Name	Data Type	Length	Allow Nulls
ID	int	4	
AD	nvarchar	50	
SOYAD	nvarchar	50	
EMAIL	nvarchar	50	

Şimdide uygulama kodlarımızı yazalım.

```
SqlConnection con;  
SqlDataAdapter da;  
DataSet ds;
```

```
/* SqlConnection nesnemizi oluşturduğumuz metodumuz. Bu metotda bağlantı bilgisini  
App.Config dosyasında tuttuğumuz connectionString isimli key' e ait value özelliğinden  
alıyoruz.*/
```

```
private void BaglantiHazirla()  
{  
    try  
    {  
        con=new  
SqlConnection(ConfigurationSettings.AppSettings["connectionString"].ToString());  
    }  
    catch(SqlException hata)  
    {  
        MessageBox.Show(hata.Message);  
    }  
}
```

/* Verileri yükleyen metodumuz parametre olarak aldığı string bilgiyi kullanan bir SqlDataAdapter nesnesi oluşturuyor. Daha sonra bu nesne yardımıyla DataSet' imiz dolduruluyor. Son olarak DataSet içindeki tablomuza ait primary key kolonu belirleniyor.*/

```
private void VerileriYukle(string sorguCumlesi)
{
    BaglantiHazirla();
    da=new SqlDataAdapter(sorguCumlesi,con);
    ds=new DataSet();
    da.Fill(ds);
    ds.Tables[0].PrimaryKey=new DataColumn[]{ds.Tables[0].Columns["ID"]};
}
```

/* Veri Çek başlıklı butona tıklandığında, MAILS tablosundaki tüm verileri çekeceğimiz sorguyu çalıştıracak VerileriYukle metodunu çağırıyor ve sonuç kümesini DataGrid kontrolümüze bağlıyoruz. Ardından eğer SqlConnection nesnemiz açık ise kapatıyoruz.*/

```
private void btnVeriCek_Click(object sender, System.EventArgs e)
{
    VerileriYukle("SELECT * FROM MAILS");
    dgVeriler.DataSource=ds.Tables[0];
    if(con.State==ConnectionState.Open)
    {
        con.Close();
    }
}
```

/* VeriGüncelle metodu Update sorgusunu bizim tanımladığımız SqlDataAdapter nesnesini kullanarak güncelleme işlemini gerçekleştiriyor.*/

```
private void VeriGuncelle()
{
    try
    {
        string guncellemeCumlesi="UPDATE MAILS SET
AD=@AD,SOYAD=@SOYAD,EMAIL=@EMAIL WHERE ID=@ORGID AND AD=@ORGAD
AND SOYAD=@ORGSOYAD AND EMAIL=@ORGEMAIL";
        // Timestamp alanı olduğunda : Update Mails Set
Ad=@Ad,Soyad=@Soyad,Email=@Email Where Id=@ORGID AND
KONTROL=@KONTROL
        SqlCommand cmdUpdate=new SqlCommand(guncellemeCumlesi,con);
        /* Sorgumuz için gerekli parametreleri ekliyoruz. Parametre adlarını, veri tiplerini,
boyutlarını ve DataTable daki hangi alanı source olarak alacaklarını belirliyoruz.*/
        cmdUpdate.Parameters.Add("@AD",SqlDbType.NVarChar,50,"AD");
        cmdUpdate.Parameters.Add("@SOYAD",SqlDbType.NVarChar,50,"SOYAD");
        cmdUpdate.Parameters.Add("@EMAIL",SqlDbType.NVarChar,50,"E MAIL");

        /* WHERE koşulunda kullanılan parametreleri giriyoruz. Burada parametre değerlerimiz
field' ların orjinal değerleri olacak. Bunu sağlamak için SourceVersion özelliğine
DataRowVersion numaralandırıcısının Original değerini atıyoruz.*/
        cmdUpdate.Parameters.Add("@ORGID",SqlDbType.NVarChar,50,"ID");
```

```

cmdUpdate.Parameters["@ORGID"].SourceVersion=DataRowVersion.Original;
cmdUpdate.Parameters.Add("@ORGAD",SqlDbType.NVarChar,50,"AD");
cmdUpdate.Parameters["@ORGAD"].SourceVersion=DataRowVersion.Original;
cmdUpdate.Parameters.Add("@ORGSOYAD",SqlDbType.NVarChar,50,"SOYAD");
cmdUpdate.Parameters["@ORGSOYAD"].SourceVersion=DataRowVersion.Original;
cmdUpdate.Parameters.Add("@ORGEMAIL",SqlDbType.NVarChar,50,"EMAIL");
cmdUpdate.Parameters["@ORGEMAIL"].SourceVersion=DataRowVersion.Original;
// Where cümleciğinden timestamp veya uniqueidentifier kullanıldığında yukarıdaki
parametre tanımlamaları yerine sadece Kontrol alanı için tek bir parametre tanımlamasının
yapılması yeterli olacaktır.
// cmdUpdate.Parameters.Add("@KONTROL",SqlDbType.Timestamp,8,"KONTROL");
// cmdUpdate.Parameters["@KONTROL"].SourceVersion=DataRowVersion.Original;

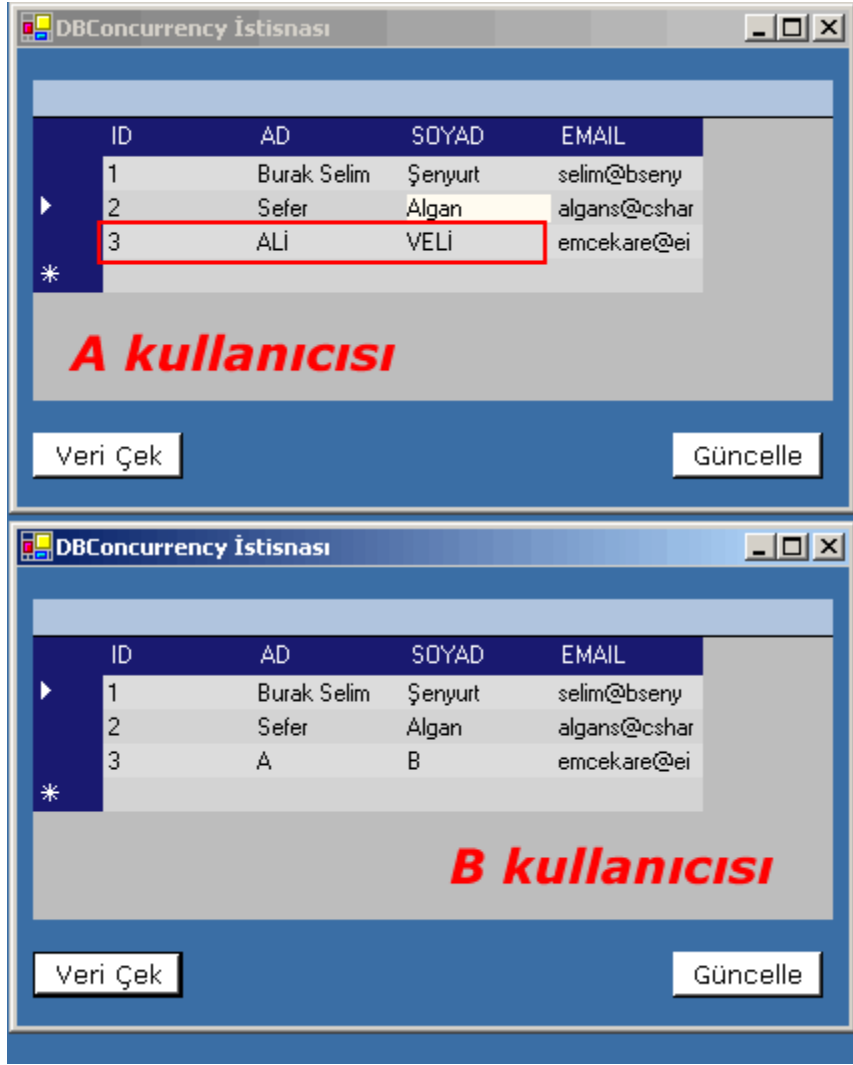
da.UpdateCommand=cmdUpdate;

/* Son olarak Update metodunu çalıştırıyoruz.*/
da.Update(ds);
}
catch(SqlException hata)
{
    MessageBox.Show(hata.Message);
}
}

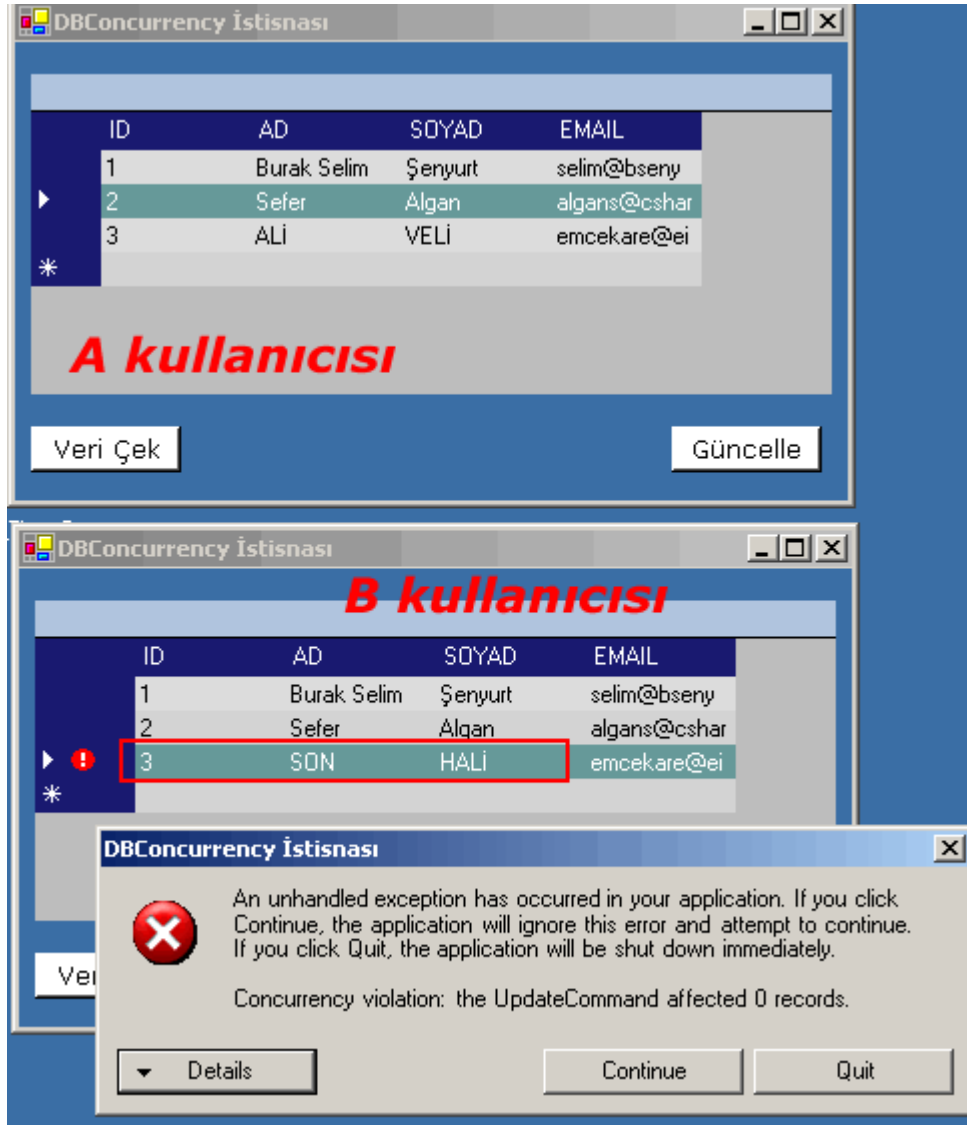
private void btnGuncelle_Click(object sender, System.EventArgs e)
{
    VeriGuncelle();
}

```

Uygulamamızdan iki tane çalıştırdığımızı ve örneğin AD alanı A ve SOYAD alanı B olan satırların değerlerini sırasıyla ALİ ile VELİ olarak değiştirdiğimizi düşünelim. Eğer Güncelle butonuna tıklarsak işlemin başarılı bir şekilde gerçekleştirildiğini görürüz.



Şimdi ikinci kullanıcımız aynı satırın verilerini değiştirelim ve yine Güncelle butonuna basalım. Bu durumda aşağıdaki gibi bir istisna mesajını alırız.



Görüldüğü gibi ikinci kullanıcı update işlemini gerçekleştirmeye çalıştığında Concurrency Violation (eş zamanlı uyumsuzluk) durumu oluşacaktır. Bu belirleyici olarak DBConcurrencyException türünden bir istisnadır. Peki oluşan bu istisnai durumun üstesinden nasıl gelebiliriz? İlk akla gelen yöntem, istisna yakalandığında kullanıcıların verilerin en güncel hallerini elde etmeleri konusunda uyarılmalarını sağlamak olacaktır. Ancak bağlantısız katman üzerinde çalışırken, ikinci kullanıcılar bu örnekte olduğu gibi tek bir satırı güncellemek dışında yeni satır girişleri, satır silmeler ve hatta başka satır güncellemeleri gibi birden fazla sayıda işlemi gerçekleştirmiş olabilirler.

Eğer güncelleme yapılan kod satırlarını istisna yakalama mekanizmaları ile izlemez ve DbConcurrencyException hatasını yakalamazsak, kullanıcının o ana kadar yaptığı tüm değişiklikler uygulamanın istem dışı sonlanması nedeni ile kaybolacaktır. Bu elbetteki istenen bir durum değildir. Alternatif bir yol olarak, DataAdapter nesnesinin **ContinueUpdateOnError** özelliğine **true** değeri verilebilir. Bu durumda Update işlemi sırasında oluşacak olan hatalar göz ardı edilecektir. Yani Concurrency'ye neden olan satırlar var ise, bunların

oluşturdukları istisnalar ortama fırlatılmayacaktır. Örneğimize bu durumu simüle edebileceğimiz bir checkBox kontrolü koyalım. Kullanıcı bu kutucuğu işaretler ise update işlemi sırasında oluşacak olan Concurrency Violation (eş zamanlı uyumsuzluk) istisnası görmezden gelinecektir. İlgili metodumuza ait kodlarımızı aşağıdaki gibi değiştirelim.

```
private void btnGuncelle_Click(object sender, System.EventArgs e)
{
    if(chkContinueUpdateOnError.Checked==true)
    {
        da.ContinueUpdateOnError=true;
        VeriGuncelle();
    }
    else if(chkContinueUpdateOnError.Checked==false)
    {
        da.ContinueUpdateOnError=false;
        VeriGuncelle();
    }
}
```

Şimdi, yine Concurrency olayına neden olacak şekilde değişiklikler yapalım. Yani her iki kullanıcımızda verileri çektikten sonra, birinci kullanıcımız belli bir satırı güncellesin. Ardından ikinci kullanıcımız aynı satırı tekrar güncellemeye çalışsın. Bu durumda her hangibir istisna fırlatılmaz ve uygulama istem dışı bir şekilde sonlanmaz. Dahası, ikinci kullanıcının yaptığı başka değişiklikler eğer var ise veritabanına başarılı bir şekilde yansıtılır.

Ancak halen daha sorunlu olan satıra ait kullanıcı yeterli bilgiye sahip değildir. *(Her ne kadar DataGrid bunu ünlem işaretleriyle belirtse de başka kontroller için bu özelliği sağlayamayabiliriz.)* Örneğin kullanıcıyı hangi satırların Concurrency Violation (eş zamanlı uyumsuzluk) istisnasına neden olduğu konusunda daha detaylı bir şekilde uyarabiliriz. Burada DBConcurrencyException sınıfının prototipi aşağıdaki gibi olan Row özelliği işimize yarayabilir.

```
public DataRow Row {get; set;}
```

Bu özellik geriye hataya neden olan satırı işaret edebilecek bir DataRow nesne örneği döndürür. Böylece ilgili satıra ait detaylı bilgilere ulaşabiliriz. Ancak, istisnai durum Concurrency' e neden olan ilk satır görüldüğünde devreye girmektedir. Dolayısıyla ikinci kullanıcının elinde Concurrency istisnasına neden olacak birden fazla satır varsa tüm bu satırları yakalamak için alternatif bir yol uygulamamız gerekmektedir. Ado.Net mimarisinde yer alan DataSet, DataTable ve DataRow sınıflarının **HasErrors** özellikleri bu noktada bizim işimize yarayabilir.

```
public bool HasErrors {get;}
```

Bu özellik bool tipinden olup, herhangi bir hata var ise geriye true değerini döndürecek. Concurrency durumunu bu hatalar arasında sayabiliriz. Şimdi uygulama kodlarımıza aşağıdaki metodu ekleyelim.

```

private void SonHaliAl()
{
    string satirBilgi;
    if(ds.Tables[0].HasErrors)
    {
        foreach(DataRow dr in ds.Tables[0].Rows)
        {
            if(dr.HasErrors)
            {
                satirBilgi=dr["AD"].ToString()+" "+dr["SOYAD"].ToString()+" Başkası tarafından
                değiştirilmiş. Satırın son halini elde etmek ister misiniz?";
                if(MessageBox.Show(satirBilgi,"Son hali
                al",MessageBoxButtons.YesNo,MessageBoxIcon.Question)==DialogResult.Yes)
                {
                    SqlCommand cmdSonHaliAl=new SqlCommand("SELECT * FROM MAİLS
                    WHERE ID="+int)dr["ID"],con);
                    if(con.State==ConnectionState.Closed)
                    {
                        con.Open();
                    }
                    SqlDataReader
                    drGuncelSatir=cmdSonHaliAl.ExecuteReader(CommandBehavior.SingleRow);
                    drGuncelSatir.Read();
                    dr.BeginEdit();
                    dr["ID"]=drGuncelSatir["ID"];
                    dr["AD"]=drGuncelSatir["AD"];
                    dr["SOYAD"]=drGuncelSatir["SOYAD"];
                    dr["EMAIL"]=drGuncelSatir["EMAIL"];
                    // Timestamp veya uniqueidentifier tipinden bir alan kullandıysak (örneğimizdeki
                    KONTROL alanı gibi) onuda güncellememiz gerekir.
                    // dr["KONTROL"]=drGuncelSatir["KONTROL"];
                    dr.EndEdit();
                    con.Close();
                }
            }
        }
        ds.Tables[0].AcceptChanges();
    }
}

```

Bu metod ile ilk olarak dataTable' ın HasErrors özelliğine bakıyoruz. Eğer bir hata var ise, her bir satırı taramaya başlıyoruz. Her bir satırın HasErrors özelliğinin değerine bakarak hatalı satırları, bir başka deyişle Concurrency Violation (eş zamanlı uyumsuzluk)' a neden olanları buluyoruz. Sonra, hatalı satırın primary key olduğunu bildiğimiz ID değerini kullanarak ilgili satırın birinci kullanıcı tarafından güncellenmiş olan halini çekiyoruz. Bunu yaparkende SqlCommand ve SqlDataReader nesnelerimizi kullanıyoruz. Burada ID alanı primary key olduğundan ve benzersiz olarak satırları işaret edebildiğinden tek satır

döneceğinden eminiz. Bu nedenle **CommandBehavior** numaralandırıcısının **SingleRow** değerini kullandık.

Bu bize performans açısından ekstra zaman kazandıracaktır. Ardından Concurrency Violation (eş zamanlı uyumsuzluk) içinde kalan satırın alanlarına ait değerleri, asıl veritabanından çektiklerimiz ile değiştiriyoruz. İşte bu noktadan sonra eğer kullanıcı tekrarda aynı satırları update eder ise hiç bir problem ile karşılaşmayacaktır. Nitekim, satırların **DataRowVersion.Original** değerleri veritabanındaki en güncel halleri ile değiştirilmiş olacaktır. Dilersek, ikinci kullanıcının o ana kadar yapmış olduğu ve Concurrency Violation (eş zamanlı uyumsuzluk) altında kalan değişikliklerin tekrardan yazılmasını sağlayabiliriz.

Tek yapmamız gereken Concurrency Violation (eş zamanlı uyumsuzluk)'de kalan alanların o anki değerlerini bir şekilde saklamak, alanların güncel hallerini çekerek orjinal değerleri yeni hallerine set etmek ve son olarak sakladığımız alan değerlerini tekrardan veritabanına göndermektir. Yazdığımız SonHaliAl isimli metodu catch bloğu içerisinde çağırılmaktadır. Nitekim Concurrency Violation (eş zamanlı uyumsuzluk) durumları ancak SqlDataAdapter nesnemizin Update metodunu çağırdıktan sonra ortaya çıkan istisna içerisinde ele alınabilir.

```
private void VeriGuncelle()
{
    try
    {
        // diğer kod satırları
        da.Update(ds);
    }
    catch(DBConcurrencyException)
    {
        SonHaliAl();
    }
}
```

Böylece geldik bir makalemizin daha sonuna. Bu makalemizde kısaca bağlantısız katmanda meydana gelebilecek eş zamanlı çakışmaları nasıl ele alabileceğimizi incelemeye çalıştık. Bir sonraki makalemizde görüşünceye dek hoşçakalın.

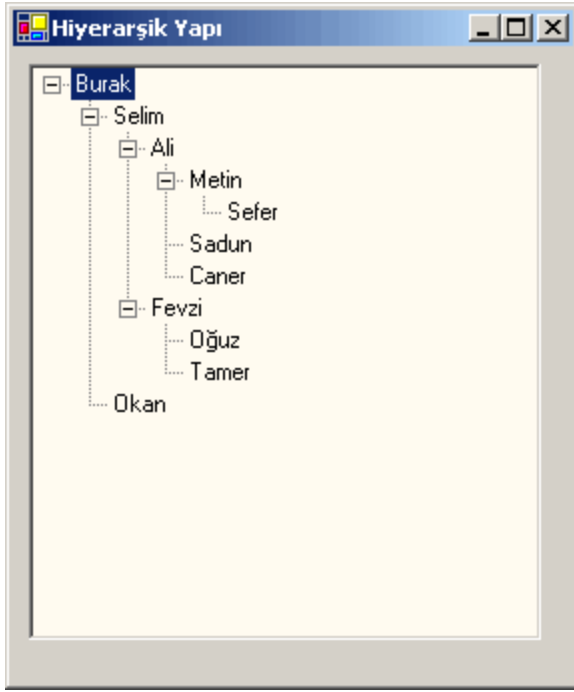
Self Referencing Relations ve Recursive(Yinelemeli) Metodlar - 14 Mart 2005 Pazartesi

ado.net, relations, recursive functions, self referencing,

Çoğumuz çalıştığımız projelerde müdür,müdür yardımcısı gibi ast üst ilişkisine sahip olan organizasyonel yapılarla karşılaşmışızdır. Örneğin işletmelerin Genel Müdür' den başlayarak en alt kademedeki personele kadar inen organizasyonel yapılar gibi. Burada söz konusu olan ilişkiler çoğunlukla pozisyonel bazdadır. Yani çok sayıda personel birbirlerine pozisyonel bazda bağımlıdır ve bu bağımlılık en üst pozisyondan en alt pozisyona kadar herkesi kapsar. Bu tarz bir sistemin uygulama ortamında canlandırılabilmesi için pozisyonel ilişkileri ifade edebilecek tablo yapılarına başvurulur. Özellikle pozisyonlar arasındaki ilişkiyi kendi içerisinde referans edebilen tablolar bu tarz ihtiyaçların karşılanması için biçilmiş kaftandır. Örneğin aşağıdaki tabloyu göz önüne alalım.

dbo.Kadro : Ta...KI.Veritabanım)			
PersonelNo	Personel	Unvan	Amiri
1	Burak	Genel Müdür	<NULL>
2	Selim	Müdür Yardımcısı	1
3	Ali	Müdür	2
4	Fevzi	Müdür	2
5	Metin	Mühendis	3
6	Sadun	Mühendis	3
7	Caner	Mühendis	3
8	Oğuz	Mühendis	4
9	Sefer	Mühendis	5
10	Okan	Mali İşler	1
11	Tamer	Uzman	4

Tabloda, X şirketinin çalışan personelleri arasındaki pozisyonel hiyerarşiyi temsil eden bir yapı kullanılmıştır. Dikkat edilecek olursa, her satırın Amiri alanının değeri yine tablo içinde yer alan bir PersonelNo alanını işaret etmektedir. Bu ilişki Self Referencing Relations olarak adlandırılır. Elbette bu tarz bir sistemde hiyerarşinin nereden başladığının bir şekilde bilinmesi gerekir. Çünkü tepeden aşağıya doğru inmek için en üst birimin diğerlerinden tamamen benzersiz bir şekilde ifade edilmesine ihtiyaç vardır. Buradaki gibi Amiri alanının değeri Null olan bir satır kimseye bağlı değildir. Ancak kendisine bağlı olan bir organizasyonel hiyerarşi söz konusudur. İşte bunu sağlayabilmek için en tepede yer alacak satırın Amiri field' ının değeri Null olarak belirlenmiştir. Tabloyu daha yakından analiz edecek olursak aşağıdaki hiyerarşik yapının oluşturulabileceğini kolayca görürüz.



İşte bu makalemizde yukarıda görmüş olduğumuz hiyerarşik yapıyı bir TreeView kontrolünde nasıl ifade edebileceğimizi incelemeye çalışacağız. Burada anahtar noktalar, **Self Referencing Relations** oluşturmak ve bu ilişkileri **Recursive (Yinelemeli)** bir Metod ile uygulayabilmektir. Öncelikle Self Referencing Relation tablomuz üzerinde görmeye çalışalım. Örneğin 5 numaralı PersonelNo değerine sahip satırımızı ele alalım. Bu satırın Amiri field' ının değeri 2 dir. Yani organizasyonel yapıda, 5 numaralı satır aslında 2 numaralı satırın altında yer almaktadır. Kaldı ki, 2 numaralı satırda 1 numaralı satırın altındadır. Bu şekilde tüm satırların birbirleri ile olan bağlantılarını tespit edebiliriz.

dbo.Kadro : Ta...KI.Veritabanım)				
	PersonelNo	Personel	Unvan	Amiri
	1	Burak	Genel Müdür	<NULL>
	2	Selim	Müdür Yardımcısı	1
	3	Ali	Müdür	2
	4	Fevzi	Müdür	2
	5	Metin	Mühendis	3
	6	Sadun	Mühendis	3
	7	Caner	Mühendis	3
	8	Oğuz	Mühendis	4
	9	Sefer	Mühendis	5
	10	Okan	Mali İşler	1
	11	Tamer	Uzman	4



Self Referencing Relation özelliğini sağlayan tablolarda bir satır(satırların) bağlı olduğu satırın yine bu tabloda var olmasını sağlamak veri tutarlılığı (consistency) açısından önemlidir.

Buradaki ilişkiyi uygulamalarımızda tanımlamak için DataRelation nesnelerini kullanabiliriz. Örneğin;

```
DataRelation dr=new
DataRelation("self",ds.Tables[0].Columns["PersonelNo"],ds.Tables[0].Columns["Amiri"],false)
;
```

Bizim için ikinci önemli nokta bu ilişkiyi kullanacak olan Recursive(Yinelemeli) bir metodun geliştirilmesidir. Yinelemeli metodumuzu geliştirirken tablo içerisindeki her bir satırı ele alacak ve bu satırların var ise alt satırlarına da bakacak bir algoritmayı göz önüne almamız gerekiyor. Örneğin aşağıdaki Console uygulamasını ele alalım.

```
class Worker
{
    SqlConnection con=new SqlConnection("data
source=BURKI;database=Veritabanim;integrated security=SSPI");
    SqlDataAdapter da;
    DataSet ds;
    DataRelation dr;

    public void Baglan()
    {
        if(con.State==ConnectionState.Closed)
            con.Open();
    }

    public void VeriCek()
    {
        da=new SqlDataAdapter("SELECT * FROM Kadro",con);
        ds=new DataSet();
        da.Fill(ds);
        dr=new
DataRelation("self",ds.Tables[0].Columns["PersonelNo"],ds.Tables[0].Columns["Amiri"],false)
;
        ds.Relations.Add(dr);
    }
    //Recursive metodumuz.
    public void DetayiniAl(DataRow dr,string giris)
    {
        Console.WriteLine(giris+dr["Personel"].ToString());
        foreach(DataRow drChild in dr.GetChildRows("self"))
        {
            DetayiniAl(drChild,giris+"...");
        }
    }

    public void AgacOlustur()
    {
        foreach(DataRow dr in ds.Tables[0].Rows)
        {
            if(dr.IsNull("Amiri"))
```



```

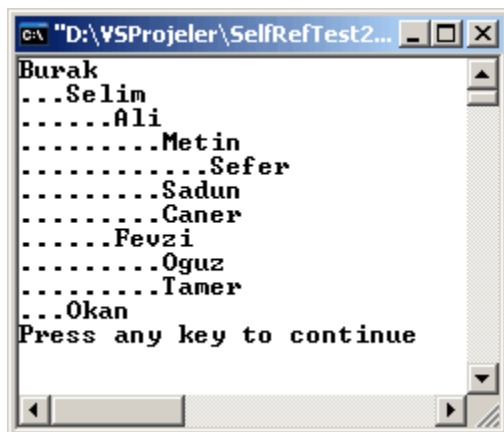
        {
            DetayiniAl(dr,"");
        }
    }
}

class Class1
{
    [STAThread]
    static void Main(string[] args)
    {
        Worker wrk=new Worker();
        wrk.Baglan();
        wrk.VeriCek();
        wrk.AgacOlustur();
    }
}

```

Bu uygulamada basit olarak tablomuzdaki verileri çekiyor ve organizasyonel yapıyı hiyerarşik olarak elde ediyoruz. Ağacımızı oluşturduğumuz AgacOlustur metodu, DataTable içerisindeki tüm satırları gezen bir foreach döngüsünü kullanıyor. Herşeyden önce bizim en tepedeki satırı bir başka deyişle en üstteki pozisyonu bulmamız gerekiyor.

Tablo yapımızdan Amiri alanının değeri Null olan satırın hiyerarşinin tepesinde olması gerektiğini biliyoruz. Bu nedenle if koşulu ile bu alanı buluyoruz. Ardından bulduğumuz satırı DetayiniAl isimli Recursive(Yinelemeli) metodumuza gönderiyoruz. Yinelemeli metodumuz gelen DataRow nesnesinin Child satırlarını gezen başka bir foreach döngüsü kullanıyor. Eğer Child satırlar var ise yinelemeli metodumuz tekrardan çağırılıyor. Bu işlem tüm satırların okunması bitene kadar devam edecektir. Sonuç itibarıyla Console uygulamamızı çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.



Gelelim, Windows uygulamamızda bu hiyerarşiyi nasıl şekillendirebileceğimize. İzleyeceğimiz yol Console uygulamamızdaki ile tamamen aynıdır. Tek fark bu kez bir DataRow'a bağlı alt satırları alırken yinelemeli metodumuza parent node'un (ki burada bir TreeNode nesnesidir) geçirilişidir. Örnek uygulamayı aşağıda

bulabilirsiniz. *(Uygulamanın çalışmasını daha iyi anlayabilmek için Trace etmenizi öneririm.)* Burada özellikle, child satırların hangi TreeNode' nesnesine eklenmesi gerektiğinin tespiti son derece önemlidir. Dikkat ederseniz AgacOlustur metodumuzda ilk olarak Amiri alanının değeri Null olan satırı temsil edecek bir TreeNode nesnesi oluşturulmuş ve TreeView kontrolüne eklenmiştir. Daha sonra yinelemeli metodumuza bu TreeNode ve o anki DataRow nesneleri gönderilmiştir. Böylece DetayiniAl metodu içerisinde child satırların hangi TreeNode içerisine alınacağı tespit edilebilir.

TreeView kullanımı;

```
SqlConnection con=new SqlConnection("data
source=BURKI;database=Veritabanim;integrated security=SSPI");
SqlDataAdapter da;
DataSet ds;
DataRelation dr;

private void Baglan()
{
    if(con.State==ConnectionState.Closed)
        con.Open();
}

private void VeriCek()
{
    da=new SqlDataAdapter("SELECT * FROM Kadro",con);
    ds=new DataSet();
    da.Fill(ds);
    dr=new
DataRelation("self",ds.Tables[0].Columns["PersonelNo"],ds.Tables[0].Columns["Amiri"],false)
;
    ds.Relations.Add(dr);
}

private void DetayiniAl(DataRow dr,TreeNode t)
{
    foreach(DataRow drChild in dr.GetChildRows("self"))
    {
        TreeNode tnChild=new TreeNode(drChild["Personel"].ToString());
        t.Nodes.Add(tnChild);
        DetayiniAl(drChild,tnChild);
    }
}

private void AgacOlustur()
{
    foreach(DataRow dr in ds.Tables[0].Rows)
```

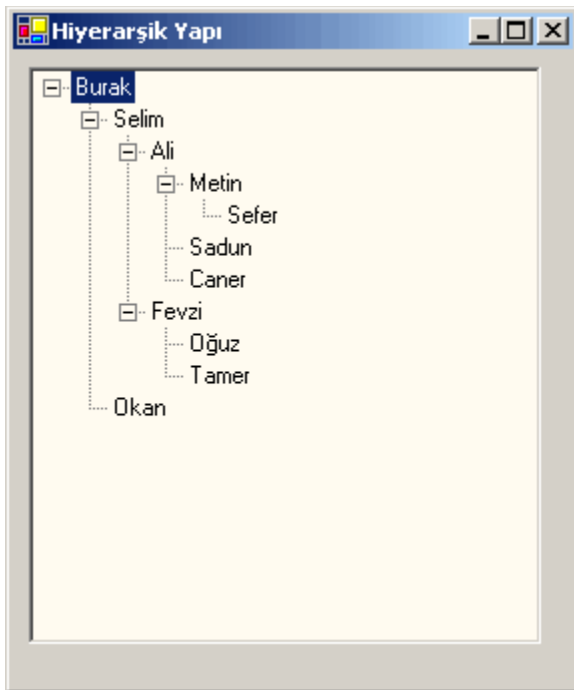
```

{
    if(dr.IsNull("Amiri"))
    {
        TreeNode tn=new TreeNode(dr["Personel"].ToString());
        treeView1.Nodes.Add(tn);
        DetayiniAl(dr,tn);
    }
}
}

private void Form1_Load(object sender, System.EventArgs e)
{
    Baglan();
    VeriCek();
    AgacOlustur();
}

```

Uygulamamızı çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.



Bu makalemizde kendi satırlarını işaret eden satırların var olduğu ilişkileri taşıyan tablolarda, satır arasındaki hiyerarşik yapının Recursive(Yinelemeli) metodlar ile uygulama ortamına nasıl aktarılabilceğini incelemeye çalıştık. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

[Örnek kodlar için tıklayın.](#)

Command Nesnelere Dikkat! - 27 Mart 2005 Pazar

ado.net, sqlcommand,

Değerli Okurlarım, Merhabalar.

Bu makalemizde, Command nesnelerini kullanırken performans artırıcı, kod okunurluğunu kolaylaştırıcı, güvenlik riskini azaltıcı etkenler üzerinde duracağız ve bu kazanımlar için gerekli teknikleri göreceğiz. Örneklerimizi SqlCommand sınıfına ait nesneler üzerinden geliştireceğiz. Bildiğiniz gibi Command nesneleri yardımıyla veritabanına doğru yürütmek istediğimiz sorguları çalıştırmaktayız. Bu sorgular basit Select, Insert, Update, Delete sorguları olabileceği gibi saklı yordamlar (Stored Procedures) veya tablolarda olabilir.

Command nesneleri ayrıca diğer ADO.NET nesnelerinin işletilmelerinde de etkin rol oynamaktadır. Örneğin bağlantısız katman (disconnected layer) nesnelerinin doldurulması veya güncellenmesi için kullanılan DataAdapter nesneleri veya bağlantılı katman (connected layer) üzerinde çalışan DataReader nesneleri gibi. Dolayısıyla Command nesnelere bağımlı olarak çalışan programların performans, güvenlik ve kod okunurluğu yönünden uygulaması tavsiye edilen bazı teknikler vardır. Command nesnelerinin hazırlanışı ve kullanılması sırasında dikkat etmemiz gereken noktalar aşağıdaki dört madde ile özetlenmiştir.

SqlCommand Nesneleri İçin Pozitif Yaklaşımlar
Parametrik sorguların kullanımı.
Sorguların yeniden kullanım için hazırlanması (Prepare Tekniği)
En etkin constructor ile nesne örneğinin oluşturulması.
Tek değerlik dönüşler için ExecuteScalar metodunun tercih edilmesi.

Şimdi bu maddelerimizi tek tek incelemeye başlayalım.

Parametrik Sorguların Kullanımı.

Parametrik sorgular diğer türlerine göre daha hızlı çalışır. Ayrıca Sql Injection' a karşı daha yüksek güvenlik sağlar. Son olarak, parametrik sorgularda örneğin

string değerler için tek tırnak kullanma zorunluluğunda kalmazsınız ki bu kodunuzun okunulabilirliğini arttırır. Örneğin aşağıdaki kod parçasını inceleyelim. Bu örneğimizde tabloya veri girişi için INSERT sorgusu kullanılıyor. Sorgumuza ait Sql cümlecğine dikkat edecek olursanız TextBox kontrollerinden değerler almakta. İfade okunurluk açısından oldukça zorlayıcı. Ayrıca tek tırnak kullanılması gerektiğinden yazımında büyük dikkat gerektiriyor.

```
SqlConnection con=new SqlConnection("data source=BURKİ;database=Work;integrated security=SSPI");
string insertText="INSERT INTO Maaslar (ADSOYAD,DOGUMTARIHI,MAAS) VALUES ('"+txtADSOYAD.Text.ToString()+"','"+txtDOGUMTARIHI.Text.ToString()+"','"+txtMAAS.Text.ToString()+"'";
SqlCommand cmd=new SqlCommand(insertText,con);
con.Open();
cmd.ExecuteNonQuery();
con.Close();
```

Oysaki bu tip bir kullanım yerine sorguya giren dış değerleri parametrik olarak tanımlamak çok daha avantajlıdır. Sürat, kolay okunurluk, tek tırnak' dan bağımsız olmak. Aynı kod parçasını şimdi aşağıdaki gibi değiştirelim. Bu sefer sorgumuza alacağımız değerleri SqlCommand nesnesine birer parametre olarak ekledik. ADSOYAD alanımız nvarchar, DOGUMTARIHI alanımız DateTime, MAAS alanımız ise Money tipindedir. Bu nedenle parametrelerde uygun SqlDbType değerlerini seçtik. Daha sonra parametrelerimiz için gerekli değerlerimizi Form üzerindeki kontrollerimizden alıyor. Dikkat ederseniz textBox kontrollerinden veri alırken herhangi bir tür dönüşümü işlemi uygulamadık.

```
/* Connection oluşturulur. */
SqlConnection con=new SqlConnection("data source=BURKİ;database=Work;integrated security=SSPI");
/* Sorgu cümlecği oluşturulur.*/
string insertText="INSERT INTO Maaslar (ADSOYAD,DOGUMTARIHI,MAAS) VALUES (@ADSOYAD,@DOGUMTARIHI,@MAAS)";
/* Command nesnesi oluşturulur */
SqlCommand cmd=new SqlCommand(insertText,con);
/* Komut için gerekli parametreler tanımlanır. */
cmd.Parameters.Add("@ADSOYAD",SqlDbType.NVarChar,50);
cmd.Parameters.Add("@DOGUMTARIHI",SqlDbType.DateTime);
cmd.Parameters.Add("@MAAS",SqlDbType.Money);
/* Parametre değerleri verilir. */
cmd.Parameters["@ADSOYAD"].Value=txtADSOYAD.Text;
cmd.Parameters["@DOGUMTARIHI"].Value=txtDOGUMTARIHI.Text;
cmd.Parameters["@MAAS"].Value=txtMAAS.Text;
/* Bağlantı açılır komut çalıştırılır ve bağlantı kapatılır. */
con.Open();
cmd.ExecuteNonQuery();
con.Close();
```

Sorguların Yeniden Kullanım için Hazırlanması (Prepare Tekniği)

Stored Procedure'lerin hızlı olmalarının en büyük nedeni sql sorgularına ait planlarının ara bellekte tutulmasıdır. Aynı işlevselliği uygulamalarımızda sık kullanılan sorgu cümleleri içinde gerçekleyebiliriz. Bunun için SqlCommand nesnesinin Prepare metodu kullanılır. Bu metod yardımıyla ilgili sql sorgusuna ait planın Sql sunucusu için ara bellekte tutulması sağlanmış olur. Böylece sorgunun ilk çalıştırılışından sonraki yürütmelerin daha hızlı olması sağlanmış olur. Aşağıdaki kod parçasını ele alalım. Bu sefer arka arkaya 3 satır girişi işlemi gerçekleştiriyoruz.

```
SqlConnection con=new SqlConnection("data source=BURKI;database=Work;integrated security=SSPI");
string insertText="INSERT INTO Maaslar (ADSOYAD,DOGUMTARIHI,MAAS) VALUES (@ADSOYAD,@DOGUMTARIHI,@MAAS)"; con.Open();
SqlCommand cmd=new SqlCommand(insertText,con);
cmd.Parameters.Add("@ADSOYAD",SqlDbType.NVarChar,50);
cmd.Parameters.Add("@DOGUMTARIHI",SqlDbType.DateTime);
cmd.Parameters.Add("@MAAS",SqlDbType.Money);
// İlk veri girişi
cmd.Parameters["@ADSOYAD"].Value="Burak";
cmd.Parameters["@DOGUMTARIHI"].Value="12.04.1976";
cmd.Parameters["@MAAS"].Value=1000;
cmd.ExecuteNonQuery();

// İkinci veri girişi
cmd.Parameters["@ADSOYAD"].Value="Bili";
cmd.Parameters["@DOGUMTARIHI"].Value="10.04.1965";
cmd.ExecuteNonQuery();

// Üçüncü veri girişi
cmd.Parameters["@ADSOYAD"].Value="Ali";
cmd.Parameters["@DOGUMTARIHI"].Value="09.04.1980";
cmd.ExecuteNonQuery();
con.Close();
```

Bu tarz bir kullanım yerine, aşağıdaki kullanım özellikle ağ ortamında işletilecek olan sorgulamalarda daha yüksek performans sağlayacaktır. Tek yapmamız gereken SqlCommand nesnesini ilk kez Execute edilmeden önce Prepare metodu ile Sql Sunucusu için ara belleğe aldırmaaktır. Yani;

```
SqlConnection con=new SqlConnection("data source=BURKI;database=Work;integrated security=SSPI");
string insertText="INSERT INTO Maaslar (ADSOYAD,DOGUMTARIHI,MAAS) VALUES (@ADSOYAD,@DOGUMTARIHI,@MAAS)"; con.Open();
SqlCommand cmd=new SqlCommand(insertText,con);
cmd.Parameters.Add("@ADSOYAD",SqlDbType.NVarChar,50);
cmd.Parameters.Add("@DOGUMTARIHI",SqlDbType.DateTime);
cmd.Parameters.Add("@MAAS",SqlDbType.Money);
// İlk veri girişi
```

```
cmd.Parameters["@ADSOYAD"].Value="Burak";
cmd.Parameters["@DOGUMTARIHI"].Value="12.04.1976";
cmd.Parameters["@MAAS"].Value=1000;
cmd.Prepare();
cmd.ExecuteNonQuery();
```

// İkinci veri girişi

```
cmd.Parameters["@ADSOYAD"].Value="Bili";
cmd.Parameters["@DOGUMTARIHI"].Value="10.04.1965";
cmd.ExecuteNonQuery();
```

// Üçüncü veri girişi

```
cmd.Parameters["@ADSOYAD"].Value="Ali";
cmd.Parameters["@DOGUMTARIHI"].Value="09.04.1980";
cmd.ExecuteNonQuery();
con.Close();
```

En Etkin Constructor ile Nesne Örneğinin Oluşturulması.

Bir SqlCommand nesnesinin oluşturulması sırasında kullanılacak Constructor metodun seçimi özellikle kod okunurluğu açısından önemlidir. Örneğin aşağıdaki kod kullanımını ele alalım.

```
// ConnectionString tanımlanır.
string conStr="data source=BURKİ;database=Work;integrated security=SSPI";
// Select sorgu cümlesi tanımlanır.
string selectText="SELECT * FROM Ogrenciler";
// SqlConnection nesnesi oluşturulur.
SqlConnection con=new SqlConnection();
// SqlConnection nesnesi için Connection String atanır.
con.ConnectionString=conStr;
// Connection açılır.
con.Open();
// Yeni bir SqlTransaction nesnesi başlatılır.
SqlTransaction trans=con.BeginTransaction();
// SqlCommand nesnesi tanımlanır.
SqlCommand cmd=new SqlCommand();
// SqlCommand nesnesinin kullanacağı SqlConnection belirlenir.
cmd.Connection=con;
// SqlCommand nesnesinin kullanacağı SqlTransaction belirlenir.
cmd.Transaction=trans;
// SqlCommand nesnesinin yürüteceği sorgu cümlesi belirlenir.
cmd.CommandText=selectText;
```

Bu kod aşağıdaki gibi daha etkin bir biçimde yazılabilir.

```
// ConnectionString tanımlanır.
string conStr="data source=BURKİ;database=Work;integrated security=SSPI";
```

```
// Select sorgu cümlesi tanımlanır.  
string selectText="SELECT * FROM Ogrenciler";  
// SqlConnection nesnesi oluşturulur ve açılır.  
SqlConnection con=new SqlConnection(conStr);  
con.Open();  
// Yeni bir SqlTransaction nesnesi başlatılır.  
SqlTransaction trans=con.BeginTransaction();  
// SqlCommand nesnesi tanımlanır.  
SqlCommand cmd=new SqlCommand(selectText,con,trans);
```

Tek Değerlik Dönüşler için ExecuteScalar Metodunun Tercih Edilmesi.

Bazen sorgularımızda Aggregate fonksiyonlarını kullanırız. Örneğin bir tablodaki satır sayısının öğrenilmesi için Count fonksiyonunun kullanılması veya belli bir alandaki değerlerin ortalamasının hesaplanması için AVG (ortalama) fonksiyonu vb. Aggregate fonksiyonlarının kullanıldığı durumlarda iki alternatifimiz vardır. Bu alternatiflerden birisi aşağıdaki gibi SqlDataReader nesnesinin ilgili SqlCommand nesnesi ile birlikte kullanılmasıdır.

```
// ConnectionString tanımlanır.  
string conStr="data source=BURKİ;database=Work;integrated security=SSPI";  
// Select sorgu cümlesi tanımlanır.  
string selectText="SELECT COUNT(*) FROM Ogrenciler";  
// SqlConnection nesnesi oluşturulur ve açılır.  
SqlConnection con=new SqlConnection(conStr);  
con.Open();  
// SqlCommand nesnesi tanımlanır.  
SqlCommand cmd=new SqlCommand(selectText,con);  
// SqlDataReader nesnesi satır sayısını almak amacıyla oluşturulur.  
SqlDataReader dr=cmd.ExecuteReader(CommandBehavior.SingleResult);  
// Elde edilen sonuç okunur.  
dr.Read();  
// Hücre değeri ekrana yazdırılır.  
Console.WriteLine("Öğrenci sayısı "+dr[0].ToString());  
// SqlDataReader ve SqlConnection kaynakları kapatılır.  
dr.Close();  
con.Close();
```

Bu teknikte aggregate fonksiyonun çalıştırılmasından dönen değeri elde edebilmek için SqlDataReader nesnesi kullanılmıştır. Ancak SqlCommand nesnesinin bu iş için tasarlanmış olan ExecuteScalar metodu yukarıdaki tekniğe göre daha yüksek bir performans sağlamaktadır. Çünkü çalıştırılması sırasında bir SqlDataReader nesnesine ihtiyaç duymaz. Bu da SqlDataReader nesnesinin kullanmak için harcadığı sistem kaynaklarının var olmaması anlamına gelmektedir. Dolayısıyla yukarıdaki örnekteki kodları aşağıdaki gibi kullanmak daha etkilidir.

```
// ConnectionString tanımlanır.  
string conStr="data source=BURKİ;database=Work;integrated security=SSPI";
```



```
// Select sorgu cümlesi tanımlanır.  
string selectText="SELECT COUNT(*) FROM Ogrenciler";  
// SqlConnection nesnesi oluşturulur ve açılır.  
SqlConnection con=new SqlConnection(conStr);  
con.Open();  
// SqlCommand nesnesi tanımlanır ve ExecuteScalar ile sonuç anında elde edilir.  
SqlCommand cmd=new SqlCommand(selectText,con);  
Console.WriteLine("Satır sayısı "+cmd.ExecuteScalar().ToString());
```

Bu makalemizde, Command nesnelerini kullanırken bize performans, hız, güvenlik kod okunurluğu açısından avantajlar sağlayacak teknikleri incelemeye çalıştık. Bir sonraki makalemizde görüşmek dileğiyle hepinize mutlu günler dilerim.

DataReader Nesnelerini Kullanırken... -

04 Nisan 2005 Pazartesi

ado.net, sqldatareader,

Değerli Okurlarım, Merhabalar.

Bir önceki makalemizde Command nesnelerini kullanırken dikkat etmemiz gereken noktalara değinmiştik. Bu makalemizde ise DataReader nesnelerini kullanırken bizlere avantaj sağlayacak tekniklere değinmeye çalışacağız. Önceki makalemizde olduğu gibi ağırlık olarak SqlDataReader nesnesini ve Sql veritabanını kullanacağız. DataReader nesneleri bildiğiniz gibi, bağlantılı katman (connected-layer) üzerinde çalışmaktadır. Görevleri veri kaynağından, uygulama ortamına doğru belli bir akım üzerinden hareket edecek veri parçalarının taşınmasını sağlamaktır.

DataReader nesneleri ile veri almak bağlantısız katman (disconnected-layer) nesnelere veri çekmekten çok daha hızlıdır. Çoğunlukla DataReader nesnelerinin kullanılmasının tercih edileceği durumlar vardır. Uygulamalarımız geliştirirken çoğu zaman bağlantılı katman ile bağlantısız katman nesneleri arasında seçim yapmakta zorlanırsınız. Aşağıdaki tablo "*Ne zaman DataReader kullanırsınız?*" sorusuna ışık tutan noktalara değinmektedir.

Eğer Windows veya Asp.Net uygulaması geliştiriyor ve birden fazla form(page) için veri bağlama (data-binding) <u>gerçekleştirmiyorsanız</u> ,
--

Veriyi ara belleğe alma (caching) <u>ihtiyacınız yok ise</u> .

Eğer tablolarınız arasındaki ilişkileri (relations) uygulamalarınızda <u>kullanmıyorsanız</u> .
--

DataReader Kullanmayı Tercih Edin.

Gelelim DataReader nesnelerini kullanırken dikkat edeceğimiz altın noktalara. Bu teknikler uygulamalarımızın performansını arttıracak nitelikte olup aşağıdaki tabloda belirtilmektedir.

DataReader nesnelerini kullanırken açık Connection'ların kapatılmasını unutmayın.

Sorgu sonucu sadece tek bir satır döneceği kesin ise SingleRow tekniğini kullanın.
--

Toplu sorgular (Batch Queries) için Next Result tekniğini kullanın.

Binary veya Text bazlı alan verilerini okurken SequentialAccess tekniğini kullanın.

Şimdi bu teknikleri birer birer inceleyelim.

Açık Connection' ları Kapatmayı Unutmamak İçin

DataReader nesneleri açık ve geçerli bir Connection nesnesine ihtiyaç duyarlar. Lakin aynı Connection' ı kullanan DataReader nesneleri söz konusu ise, her bir DataReader' ın kullanılabilmesi için bir önceki DataReader' ın kullandığı Connection nesnesinin kapatılmış olması gerekir. (Bu aynı Connection nesnesini kullanan DataReader' lar var ise geçerlidir.) Örneğin aşağıdaki uygulama kodunu ele alalım;

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace DataReaderDikkat
{
    public class DbWork
    {
        private SqlConnection con;
        private SqlDataReader dr;

        /* CloseConnection kullanımına örnek metod.*/
        public DbWork(string conStr)
        {
            con=new SqlConnection(conStr);
        }

        public SqlDataReader Results(string selectQuery)
        {
            SqlCommand cmd=new SqlCommand(selectQuery,con);
            con.Open();
            dr=cmd.ExecuteReader(); // Hatalı kullanım.
            return dr;
        }
    }
}
```

DbWork sınıfımız constructor metodunda bir SqlConnection nesnesi oluşturur. Results isimli metodumuz ise parametre olarak aldığı sorgu sonucu bir SqlDataReader nesnesi ile geri döndürür. Şimdi bu sınıfı kullanan aşağıdaki uygulamamızı ele alalım.

```

using System;
using System.Data;
using System.Data.SqlClient;

namespace DataReaderDikkat
{
    class ClosingDataReader
    {
        [STAThread]
        static void Main(string[] args)
        {
            DbWork worker=new DbWork("data source=BURKI;database=Northwind;integrated
security=SSPI");
            #region CloseConnection Kullanin.

            SqlDataReader drOrders=worker.Results("SELECT TOP 10 * FROM Orders");
            while(drOrders.Read())
            {
                Console.WriteLine(drOrders[0].ToString()+" "+drOrders[1].ToString());
            }
            drOrders.Close();
            SqlDataReader drCustomers=worker.Results("SELECT TOP 10 * FROM
Customers");
            while(drCustomers.Read())
            {
                Console.WriteLine(drCustomers[0].ToString()+" "+drCustomers[1].ToString());
            }
            drCustomers.Close();
            #endregion
        }
    }
}

```

Uygulamamızı bu haliyle çalıştırdığımızda aşağıdaki istisnayı alırız.



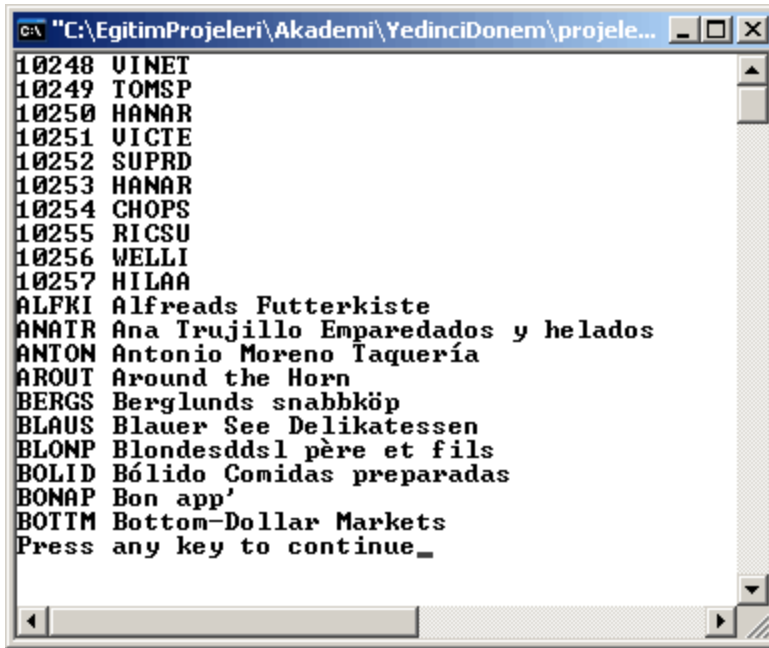
```

C:\EgitimProjeleri\Akademi\YedinciDonem\projeler\DataReaderDikkat\bin\Debug\DataReaderDikk...
10248 VINET
10249 TOMSP
10250 HANAR
10251 VICTE
10252 SUPRD
10253 HANAR
10254 CHOPS
10255 RICSU
10256 WELLI
10257 HILAA
Unhandled Exception: System.InvalidOperationException: The connection is already
Open (state=Open).
   at System.Data.SqlClient.SqlConnection.Open()
   at DataReaderDikkat.DbWork.Results(String selectQuery) in c:\egitimprojeleri\
akademi\yedincidonem\projeler\datareaderdikkat\dbwork.cs:line 21
   at DataReaderDikkat.ClosingDataReader.Main(String[] args) in c:\egitimprojele
ri\akademi\yedincidonem\projeler\datareaderdikkat\closingdatareader.cs:line 22
Press any key to continue_

```

Sebept ilk drOrders SqlDataReader nesnesinin kullandığı SqlConnection nesnesinin kapatılmamış olması ve bağlantının halen daha açık olarak kalmasıdır. Özellikle yukarıdaki gibi iş nesneleri üzerinden yürütülen sorgularda Connection nesnelerinin otomatik olarak kapatılmasını sağlamak için CommandBehavior numaralandırıcısının CloseConnection değerini kullanmayı unutmamak gerekir. Dolayısıyla DbWork sınıfımızdaki Results metodunu aşağıdaki gibi düzenlersek istediğimiz sonucu elde eder ve istisnanın üstesinden geliriz.

```
public SqlDataReader Results(string selectQuery)
{
    SqlCommand cmd=new SqlCommand(selectQuery,con);
    con.Open();
    dr=cmd.ExecuteReader(CommandBehavior.CloseConnection); // Doğru Kullanım.
    // dr=cmd.ExecuteReader(); // Hatalı kullanım.
    return dr;
}
```



Sorgu Sonucu Tek Bir Satır Döndüğü Kesin İse

Bazı durumlarda tablolardan dönen satır sayısının 1 olacağı kesindir. Bu satırlar çoğunlukla belirli key alanı üzerinden elde edilen parametrik sorguların sonucudur. Örneğin, benzersiz değer alan (unique), ve otomatik olarak artan alanların parametre olarak kullanıldığı sorgular göz önüne alabiliriz. Bu tarz sorgularda, bağlantısız katman nesnelerini kullanmak gereksiz yere kaynak tüketimine neden olacaktır. Böyle bir durumda DataReader nesneleri bağlantısız katman nesnelerine oranla çok daha performanslı ve hızlı çalışacaktır. Burada önemli olan DataReader ile dönen satırı okumak için ilgili Command nesnesinin ExecuteReader metoduna verilecek CommandBehavior numaralandırıcısının değeridir. Aşağıda bu tekniğin kullanımına bir örnek verilmiştir. Veritabanı

işlemlerimizi topladığımız DbWork sınıfı basit olarak constructor metodu ile bir SqlConnection nesnesi örneklendirir. GetRow metodumuz ise gelen sorguya ve parametre değerine göre bulunan satırı okuyacak bir SqlDataReader nesnesini geriye döndürür.

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace DataReaderDikkat
{
    public class DbWork
    {
        private SqlConnection con;
        private SqlDataReader dr;

        /* CloseConnection kullanımına örnek metod.*/
        public DbWork(string conStr)
        {
            con=new SqlConnection(conStr);
        }

        public SqlDataReader GetRow(string FindQuery,int orderID)
        {
            SqlCommand cmd=new SqlCommand(FindQuery,con);
            cmd.Parameters.Add("@OrderID",SqlDbType.Int);
            cmd.Parameters["@OrderID"].Value=orderID;
            con.Open();
            dr=cmd.ExecuteReader((CommandBehavior)40);
            return dr;
        }
    }
}
```

Burada ExecuteReader metodunun kullanımına dikkatinizi çekerim. CommandBehavior numaralandırıcısının alacağı değerlerin sayısal karşılıkları vardır. SingleRow değerinin integer karşılığı 8, CloseConnection numaralandırıcısının integer karşılığı ise 32' dir. Dolayısıyla (CommandBehavior)40, oluşturulan SqlDataReader nesnesine sadece tek bir satır döndüreceğini ve SqlDataReader nesnesi kapatıldığında SqlConnection nesnesinin de otomatik olarak kapatılacağını belirtir. Gelelim ana program kodlarımıza;

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace DataReaderDikkat
{
    class ClosingDataReader
```

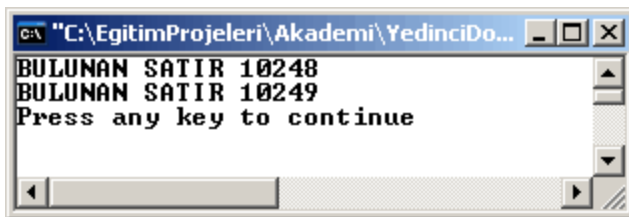
```

{
    [STAThread]
    static void Main(string[] args)
    {
        DbWork worker=new DbWork("data source=BURKI;database=Northwind;integrated
security=SSPI");

        #region SingleRow Kullanın.
        SqlDataReader drFind=worker.GetRow("SELECT * FROM Orders WHERE
OrderID=@OrderID",10248);
        drFind.Read();
        Console.WriteLine("BULUNAN SATIR "+drFind[0].ToString());
        drFind.Close();

        drFind=worker.GetRow("SELECT * FROM Orders WHERE
OrderID=@OrderID",10249);
        drFind.Read();
        Console.WriteLine("BULUNAN SATIR "+drFind[0].ToString());
        drFind.Close();
        #endregion
    }
}

```



Toplu Sorgula İçin DataReader Kullanın

Özellikle birden fazla sonuç kümesini (result set) almak istiyorsanız ve DataReader kullanmaya karar verdiyseniz en uygun yöntem NextResult metodunun uygulanmasıdır. Gerçek şu ki, böyle bir durumda birden fazla DataReader nesnesi peş peşe çalıştırılabilir. Aynı bu makalemizdeki ilk örneğimizde olduğu gibi. Eğer bu tarz sonuç kümelerini gerçekten arka arkaya alıyorsak ve aynı Connection'ı kullanıyorsak, birden fazla DataReader nesnesi kullandığımız için aynı Connection'ı kullanıyor olsakta veritabanına doğru birden fazla sayıda tur atmış oluruz. Çünkü her bir DataReader nesnesinden sonradan gelen DataReader nesnelerinin aynı Connection'ı kullanmalarına imkan sağlamamız için ilgili Connection'ları kapatmak gibi bir zorunluluğumuz vardır.

Oysaki bu sorguları Batch Query olarak hazırlarsak tek bir DataReader nesnesi ve tek bir Connection ile daha hızlı sonuç elde edebiliriz. Aynı aşağıdaki örnekte olduğu gibi. DbWork sınıfımıza bu sefer, Batch Query çalıştıracak bir metod ekledik. Metodumuz gelen string dizisi içindeki sorgu cümlelerini alıp bir

StringBuilder yardımıyla birleştiriyor. Bu işlemin sonucu olarak "sorgu cümlesi 1;sorgu cümlesi 2;sorgu cümlesi 3;" tarzında bir query string' i oluşturuyoruz ki bu bizim Batch Query' mizdir. Daha sonra ilgili sorgu topluluğunu çalıştıracak bir SqlCommand nesnesi oluşturuyor ve bu komutu yürüterek elde ettiğimiz SqlDataReader nesnesini geriye döndürüyoruz.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Text;

namespace DataReaderDikkat
{
    public class DbWork
    {
        private SqlConnection con;
        private SqlDataReader dr;

        /* CloseConnection kullanımına örnek metod.*/
        public DbWork(string conStr)
        {
            con=new SqlConnection(conStr);
        }

        public SqlDataReader BatchResults(string[] sorgular)
        {
            StringBuilder sbSorgular=new StringBuilder();
            for(int i=0;i<sorgular.Length;i++)
            {
                sbSorgular.Append(sorgular[i]+";");
            }
            SqlCommand cmd=new SqlCommand(sbSorgular.ToString(),con);
            con.Open();
            SqlDataReader dr=cmd.ExecuteReader(CommandBehavior.CloseConnection);
            return dr;
        }
    }
}
```

Gelelim uygulama kodlarımıza;

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace DataReaderDikkat
{
    class ClosingDataReader
```

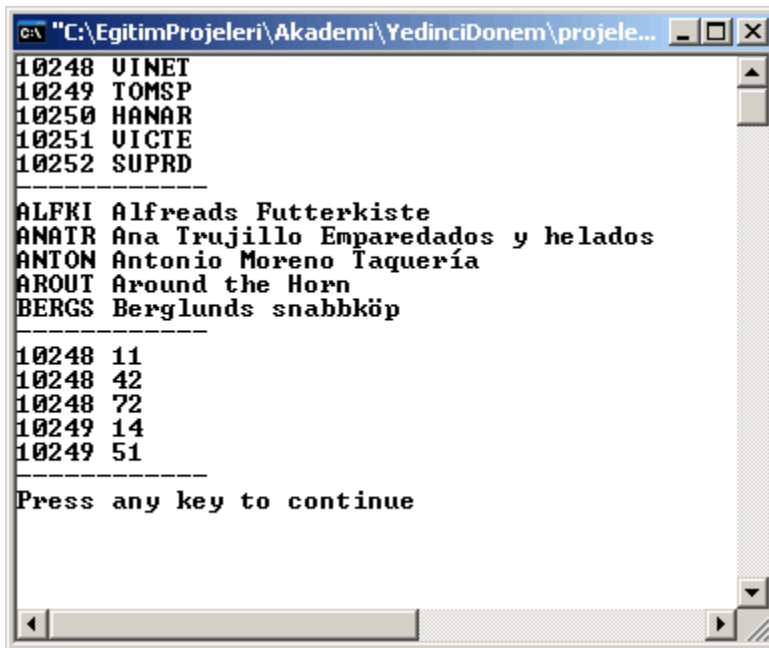


```

{
    [STAThread]
    static void Main(string[] args)
    {
        DbWork worker=new DbWork("data source=BURKI;database=Northwind;integrated
security=SSPI");

        #region BatchQuery' lerde
        string[] sorguKumesi={"SELECT TOP 5 * FROM Orders","SELECT TOP 5 * FROM
Customers","SELECT TOP 5 * FROM [Order Details]"};
        SqlDataReader drToplu=worker.BatchResults(sorguKumesi);
        do
        {
            while(drToplu.Read())
            {
                Console.WriteLine(drToplu[0].ToString()+" "+drToplu[1].ToString());
            }
            Console.WriteLine("-----");
        }while(drToplu.NextResult());
        drToplu.Close();
        #endregion
    }
}

```



Binary ve Text Tipindeki Alanları Okurken

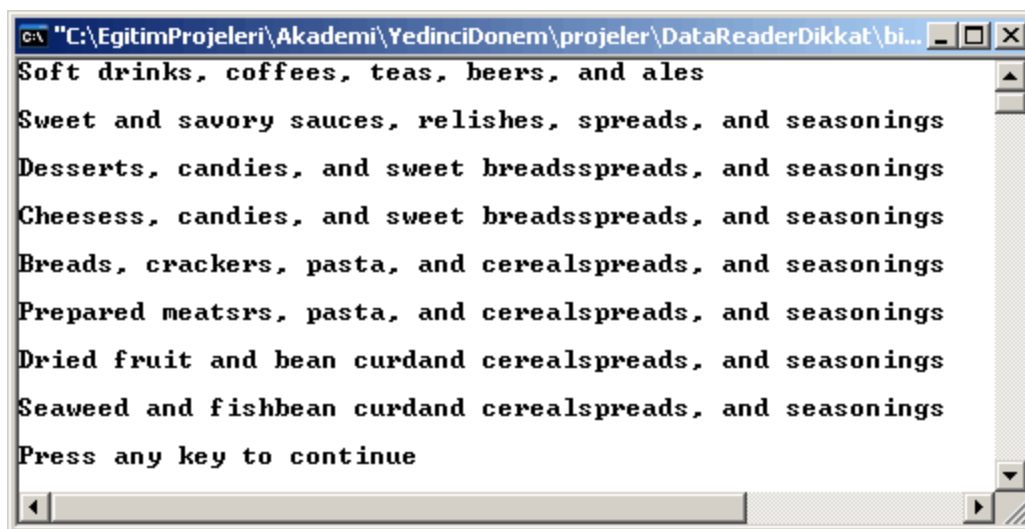
Bazı tablolar içerisinde text veya binary tabanlı alanlar tutarız. Örneğin resim dosyalarının tablolarda binary olarak saklanması veya makalelerin html verisinin text tipli alanlar olarak saklanması gibi. Özellikle bu tarz alanları okurken

DataReader nesnelerini kullanıyorsak, SequentialAccess tekniğini kullanmak bize avantaj sağlayabilir. Öyle ki bu tekniği uyguladığımızda ilgili satırın tamamı okunacağına bunun yerine bir stream oluşturulur. Siz bu stream' i kullanarak ilgili alana ait binary yada text veriyi okursunuz. Örneğin aşağıdaki kodlar ile Northwind database' inde yer alan Categories tablosundaki Text tipindeki Description alanının ilk 150 karakteri okunmaktadır.

```
public SqlDataReader ReadText(string Query)
{
    SqlCommand cmd=new SqlCommand(Query,con);
    con.Open();
    dr=cmd.ExecuteReader((CommandBehavior)48); // Bu kez hem SequentialAccess hem
    de CloseConnection seçili.
    return dr;
}
```

Metodumuzun kullanılışı ise aşağıdaki gibi olacaktır.

```
SqlDataReader drText=worker.ReadText("SELECT CategoryName,Description,Picture
FROM Categories");
char[] tampon=new char[150];
while(drText.Read())
{
    drText.GetChars(1,0,tampon,0,150);
    for(int i=0;i<150;i++)
    {
        Console.Write(tampon[i].ToString());
    }
    Console.WriteLine();
}
```



Bu makalemizde DataReader nesnelerini kullanırken dikkat edeceğimiz ve bize avantaj sağlayacak teknikleri incelemeye çalıştık. Bir sonraki makalemizde görüşmek dileğiyle hepinize mutlu günler dilerim.

[Örnek için tıklayın.](#)

Ado.Net 2.0' da Transaction Kavramı - 17 Nisan 2005 Pazar

ado.net 2.0, transaction, oledb transaction, lightweight transaction,

Değerli Okurlarım, Merhabalar.

Transaction kavramı ve kullanımı veritabanı programcılığının olmazsa olmaz temellerinden birisidir. Veritabanına doğru gerçekleştirilen işlemlerin tamamının onaylanması veya içlerinden birisinde meydana gelecek bir istisna sonrası o ana kadar yapılan tüm işlerin geri alınması veri bütünlüğünü korumak açısından son derece önemlidir. Ado.Net 1.0/1.1 için transactionların kullanımı, seçilen veri sağlayıcısına göre farklı sınıfların kullanılmasını gerektirir.

Örneğin SqlConnection isim alanındaki sınıfları kullandığınız bir veritabanı uygulamanız var ise, SqlTransaction sınıfını kullanırsınız. Oysa Ado.Net 2.0' da transaction mimarisi Ado.Net' ten ayrıştırılmış, bir başka deyişle provider' lardan bağımsız hale getirilmiştir. Aslında en büyük değişiklik transaction işlemlerinin artık System.Transactions isim alanı altında yer alan sınıflar ile gerçekleştirilecek olmasıdır. Bir diğer büyük değişiklik transaction' ların yazım tekniği ile ilgilidir. Ado.Net 2.0 da transaction oluşturacak ve kullanacak kodları çok daha basit biçimde yazabilirsiniz. Bunu ilerleyen paragraflarda sizde göreceksiniz.

Ado.Net 2.0' da transaction' ların kullanımı ile ilgili belkide en önemli özellik dağıtık (distributed) transaction' ların uygulanış biçimidir. Normal şartlarda Ado.Net 1.0/1.1 için dağıtık transaction' ları kullanırken System.EnterpriseServices isim alanını kullanan COM+ nesnelerini oluşturur ve ContextUtil sınıfına ait metodlar yardımıyla dağıtık transaction' ları kontrol altına alırız. ([Detaylı bilgi için tıklayın.](#)) Bu yapı özellikle yazımı ve oluşturulması itibarıyla karmaşık olup kullanımı da zor olan bir yapıdır. Oysa ki Ado.Net 2.0 olayı çok daha akıllı bir şekilde ele alır. Ado.Net 2.0' a göre, iki tip transaction olabilir. Tek veri kaynağı üzerinde çalışan **LightWeight Transaction** lar ve dağıtık transactionlar gibi davranan **OleDb Transaction** lar.

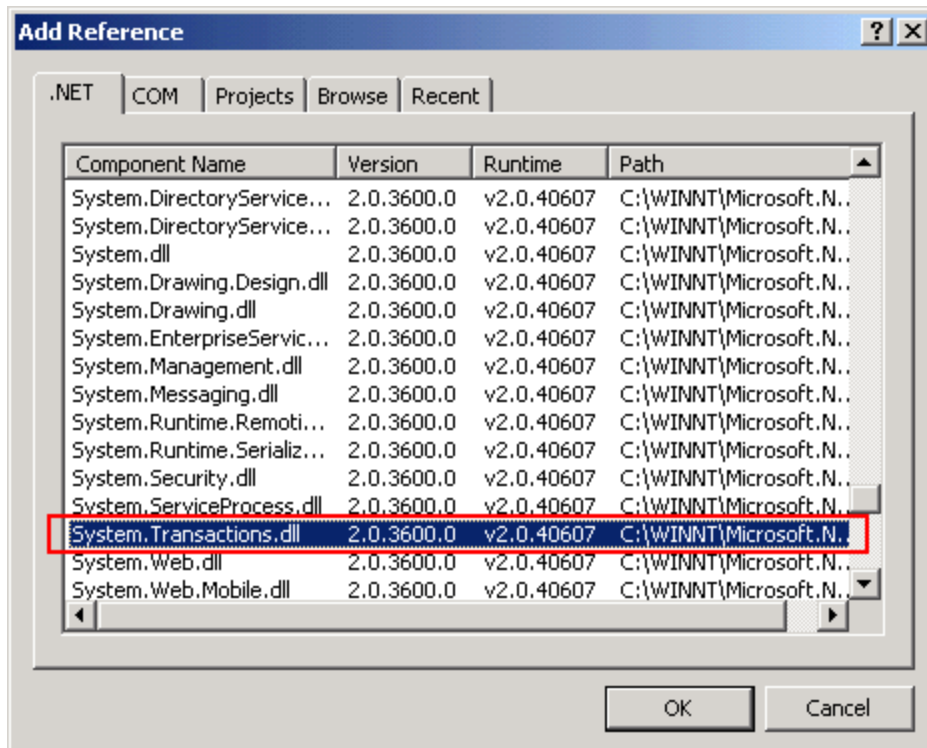
LightWeight Transaction' lar tek bir uygulama alanında (application-domain) çalışan iş parçalarıdır. OleDb tipi Transaction' lar ise birden fazla uygulama alanında (application-domain) veya aynı uygulama alanında iseler de farklı veri kaynaklarını kullanan transaction' lar dır. Dolayısıyla OleDb tipi transaction' ları Distributed Transaction' lara benzetebiliriz. Ancak arada önemli farklarda vardır. İlk olarak Ado.Net 1.0/1.1' de tek veri kaynağı üzerinde çalışan bir transaction' ın nasıl uygulandığını hatırlayalım.

```
SqlConnection con = new SqlConnection(connectionString);
con.Open();
```

```
SqlCommand cmd = new SqlCommand(sqlSorgusu,con);
SqlTransaction trans;
```

```
trans = con.BeginTransaction();
cmd.Transaction = trans;
try
{
    cmd.ExecuteNonQuery();
    trans.Commit();
}
catch(Exception e)
{
    trans.Rollback();
}
finally
{
    con.Close();
}
```

Yukarıdaki örnekte yerel (local) makine üzerinde tekil olarak çalışan bir transaction için gerekli kodlar yer almaktadır. Eğer komutun çalıştırılması sırasında herhangi bir aksilik olursa, catch bloğu devreye girer ve transaction geri çekilerek (RollBack) o ana kadar yapılmış olan tüm işlemler iptal edilir. Tam tersine bir aksilik olmaz ise transaction nesnesinin Commit metodu kullanılarak işlemler onaylanır ve veritabanına yazılır. Gelelim bu tarz bir örneğin Ado.Net 2.0' da nasıl gerçekleştirileceğine. Her şeyden önce bu sefer **System.Transactions** isim alanını kullanmamız gerekiyor. Şu anki versiyonda bu isim alanını uygulamamıza harici olarak referans etmemiz gerekmektedir.



Daha sonra ise aşağıdaki kodlara sahip olan Console uygulamasını oluşturalım.

```
#region Using directives
```

```
using System;  
using System.Collections.Generic;  
using System.Text;  
using System.Data;  
using System.Data.SqlClient;  
using System.Transactions;
```

```
#endregion
```

```
namespace Transactions
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            using (TransactionScope tsScope = new TransactionScope())  
            {  
                using (SqlConnection con = new SqlConnection("data  
source=localhost;database=AdventureWorks;integrated security=SSPI"))  
                {  
                    SqlCommand cmd = new SqlCommand("INSERT INTO Personel  
(AD,SOYAD,MAIL) VALUES ('Burak Selim','ŞENYURT','selim(at)buraksenyurt.com')", con);  
                    con.Open();  
                    cmd.ExecuteNonQuery();  
                    tsScope.Complete();  
                }  
            }  
        }  
    }  
}
```

Görüldüğü gibi ADO.NET 1.0/1.1'e göre oldukça farklı bir yapı kullanılmıştır.

TransactionScope sınıfına ait nesne örnekleri kendisinden sonra açılan bağlantıları otomatik olarak bir transaction scope (faaliyet alanı) içerisine alır. Buradaki temel mantık bir veya daha fazla transaction'ı kullanacak olan bir scope (faaliyet alanı) oluşturmak ve hepsi için gerekli bir takım özelliklerin ortak olarak belirlenmesini sağlamaktır. TransactionScope sınıfına ait nesne örneklerini oluşturabileceğimiz pek çok aşırı yüklenmiş (overload) yapıcı (constructor) metod mevcuttur.

Bu yapıcı metodlar yardımıyla, scope (faaliyet alanı) için pek çok özellik tanımlayabilirsiniz. TransactionScope, **IDisposable** arayüzünü (interface) uygulayan bir sınıftır. Bir TransactionScope nesnesi oluşturulduğunda ve bu nesnenin oluşturduğu faaliyet alanına ilk transaction eklendiğinde devam eden komutlara ilişkin transaction'lar da otomatik olarak var olan bu scope (faaliyet alanı) içerisinde gerçekleşmektedir. Bu elbetteki varsayılan

durumdur. Ancak dilerseniz **TransactionScopeOption** numaralandırıcısı (enumerator) yardımıyla, yeni açılan transaction' ların var olan scope' a (faaliyet alanına) dahil edilip edilmeyeceğini belirleyebilirsiniz.

Eğer veritabanına doğru çalışan komutlarda herhangi bir aksaklık olursa uygulama otomatik olarak using bloğunu terk edecektir. Bu durumda son satırdaki Complete metodu çağırılabilir hale gelecektir. Bu da transaction içerisindeki işlemlerin commit edilebileceği anlamına gelir. Bu yeni teknik, eskisine göre özellikle kod yazımını hem kolaylaştırmış hem de profesyonelleştirmiştir. Bununla birlikte var olan alışkanlıklarımızdan birisi meydana gelecek aksaklık nedeni ile kullanıcının bir istisna mekanizması ile uyarılabilmesini sağlamak veya başka işlemleri yaptırmaktır. Dolayısıyla aynı örneği aşağıdaki haliyle de yazabiliriz.

```
using (TransactionScope tsScope = new TransactionScope())
{
    SqlConnection con = new SqlConnection("data
source=localhost;database=AdventureWorks;integrated security=SSPI");
    SqlCommand cmd = new SqlCommand("INSERT INTO Personel (AD,SOYAD,MAIL)
VALUES ('Burak Selim','ŞENYURT','selim(at)buraksenyurt.com')", con);
    try
    {
        con.Open();
        cmd.ExecuteNonQuery();
        tsScope.Complete();
    }
    catch (TransactionException hata)
    {
        Console.WriteLine(hata.Message.ToString());
    }
    finally
    {
        con.Close();
    }
}
```

Burada oluşabilecek istisnayı yakalamak istediğimiz için bir try-catch-finally bloğunu kullandık. Ancak dikkat ederseniz TransactionScope nesnemiz yine using bloğu içerisinde kullanılmıştır ve transaction' ı commit etmek için Complete metodu çağırılmıştır. Her iki örnekte Lightweight Transaction tipindedir. Çünkü tek bir connection ve yine tek bir application domain mevcuttur. Elbette birden fazla komutun yer aldığı transaction' larda aynı teknik kullanılarak oluşturulabilir. Ancak farklı veritabanlarına bağlanan aksiyonlar söz konusu ise Transaction' ların oluşturulması ve arka planda gerçekleşen olaylar biraz farklıdır. Şimdi bu durumu örneklemek için aşağıdaki Console uygulamasını oluşturalım.

```
using (TransactionScope tsScope = new TransactionScope())
{
```

```

using (SqlConnection conAdventureWorks = new SqlConnection("data
source=localhost;database=AdventureWorks;integrated security=SSPI"))
{
    SqlCommand cmdAdvPersonel = new SqlCommand("INSERT INTO Personel
(AD,SOYAD,MAIL) VALUES ('Burak Selim','ŞENYURT','selim(at)buraksenyurt.com')",
conAdventureWorks);
    conAdventureWorks.Open();
    cmdAdvPersonel.ExecuteNonQuery();
    using (SqlConnection conNorthwind = new SqlConnection("data
source=localhost;database=Northwind;integrated security=SSPI"))
    {
        conNorthwind.Open();
        SqlCommand cmdNrtPersonel = new SqlCommand("UPDATE Personel SET
AD='Gustavo' WHERE ID=1", conNorthwind);
        cmdNrtPersonel.ExecuteNonQuery();
    }
}
tsScope.Complete();
}

```

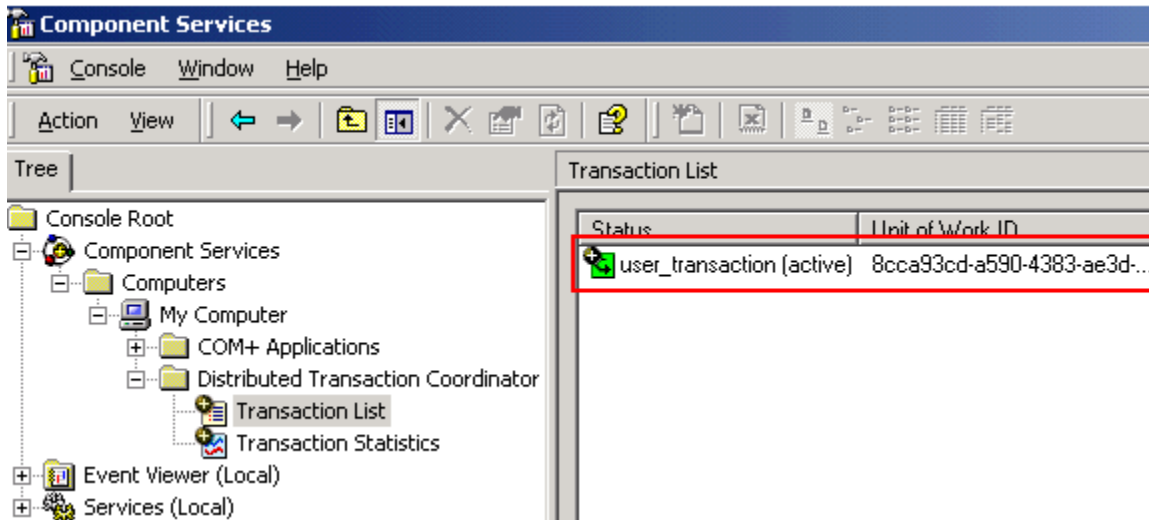
Görüldüğü gibi ilk yazdığımız örnekten pek farkı yok. Sadece iç içe geçmiş (nested) bir yapı var. Aynı application domain' e ait iki farklı database bağlantısı ihtiyacı olduğu için burada bir distributed transaction kullanılması gerekiyor. Normalde **DTC (Distributed Transaction Coordinator)** kontrolünde ele alınacak bu transaction' ları oluşturmak için Ado.Net 1.0/1.1' de bayağı uğraşmamız gerekecekti. Oysa ki Ado.Net 2.0' da herhangi bir şey yapmamıza gerek yok. Çünkü Ado.Net 2.0 otomatik olarak OleTx tipinde bir transaction oluşturacaktır. Nasıl mı? Bunu gözlemlemenin en iyi yolu, uygulama koduna **breakpoint** koymak ve **Administrative Tool->Component Services**' dan açılacak olan transaction' ları izlemekle olacaktır. İlk olarak kodumuza bir breakpoint koyalım ve adım adım uygulamamızda ilerleyelim.


```

using (TransactionScope tsScope = new TransactionScope())
{
    using (SqlConnection conAdventureWorks = new SqlConnection("data
source=localhost;database=AdventureWorks;integrated security=SSPI"))
    {
        SqlCommand cmdAdvPersonel = new SqlCommand("INSERT INTO
Personel (AD,SOYAD,MAIL) VALUES ('Burak Selim','ŞENYURT','selim@bsenyurt.com')",
conAdventureWorks);
        conAdventureWorks.Open();
        cmdAdvPersonel.ExecuteNonQuery();
        using (SqlConnection conNorthwind = new SqlConnection("data source
=localhost;database=Northwind;integrated security=SSPI"))
        {
            conNorthwind.Open();
            SqlCommand cmdNrtPersonel = new SqlCommand("UPDATE Personel
SET AD='Gusto' WHERE ID=1", conNorthwind);
            cmdNrtPersonel.ExecuteNonQuery();
        }
    }
    tsScope.Consistent = true;
}

```

Sarı noktaya gelinceye kadar ve conNorthwind isimli bağlantı Open metodu ile açılıncaya kadar aktif olan tek bir Connection nesnesi vardır. Buraya kadar transaction'ımız LightWeight tipindedir. Ancak ikinci bağlantıda açıldıktan sonra bu bağlantı TransactionScope nesnemize otomatik olarak eklenecektir. İşte sarı noktada iken Administrative Tool->Component Services' a bakarsak, **TransactionList** içerisinde, DTC kontrolü altında kullanıcı tanımlı bir transaction'ın otomatik olarak açıldığını görürüz.



Artık her iki connection üzerinde çalışan komutlar DTC altında oluşturulan bu Connection'ın kontrolü altındadır. İşlemler başarılı bir şekilde tamamlanırsa TransactionScope nesnesine ait using bloğunun sonundaki kod satırı çalışacaktır. Yani Complete metodu yürütülecektir. Bu sayede işlemler Commit edilir ve böylece tüm işlemler onaylanarak veritabanlarına yazılır. Using bloğundan

çıkıldıktan sonra ise, DTC kontrolü altındaki bu transaction otomatik olarak kaldırılır. DTC kontrolü altında oluşturulan transaction' lar her zaman unique bir ID değerine sahip olur. Böylece sunucu üzerinde aynı anda çalışan birden fazla distributed transaction var ise, bunların birbirlerinden ayırt edilmeleri ve uygun olan application domain' ler tarafından ele alınmaları sağlanmış olur.

TransactionScope nesnesinin belirlediği scope (faaliyet alanı) altında açılan transaction' lar bir takım özelliklere sahiptir. Örneğin eskiden olduğu gibi IsolationLevel değerleri veya Timeout süreleri vardır. Dilersek oluşturulacak bir TransactionScope nesnesinin ilgili değerlerini önceden manuel olarak ayarlayabilir böylece bu scope (faaliyet alanı) içindeki transaction' ların ortak özelliklerini belirleyebiliriz. Bunun için **TransactionOptions** sınıfına ait nesne örnekleri kullanılır.

```
TransactionOptions trOptions = new TransactionOptions();
trOptions.IsolationLevel = System.Transactions.IsolationLevel.ReadCommitted;
trOptions.Timeout = new TimeSpan(0, 0, 30);
using (TransactionScope tsScope = new
TransactionScope(TransactionScopeOption.RequiresNew,trOptions))
{
    using (SqlConnection con = new SqlConnection("data
source=localhost;database=AdventureWorks;integrated security=SSPI"))
    {
        SqlCommand cmd = new SqlCommand("INSERT INTO Personel (AD,SOYAD,MAIL)
VALUES ('Burak Selim','ŞENYURT','selim(at)buraksenyurt.com')", con);
        con.Open();
        cmd.ExecuteNonQuery();
        tsScope.Complete();
    }
}
```

Yukarıdaki örnekte, oluşturulacak olan transaction' ın izolasyon seviyesi ve zaman aşımı süreleri belirlenmiş ve TransactionScope nesnemiz bu opsiyonlar çerçevesinde aşağıdaki overload metot versiyonu ile oluşturulmuştur.

```
TransactionScope tsScope = new
TransactionScope(TransactionScopeOption.RequiresNew,trOptions)
```

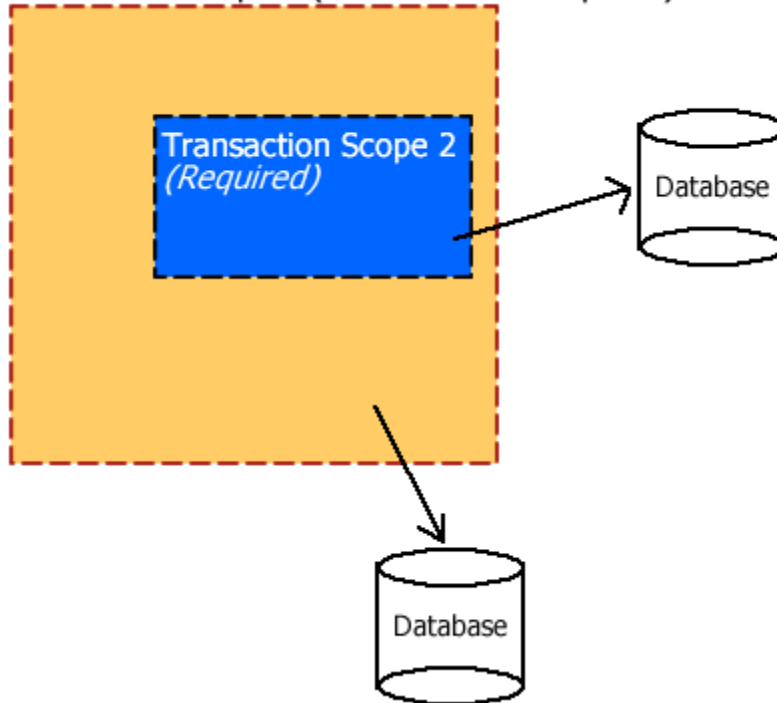
Burada ilk parametre birden fazla TransactionScope nesnesinin yer aldığı iç içe geçmiş yapılarda büyük önem arz etmektedir. Bu seçenek ile yeni açılan transaction scope' un (faaliyet alanının) var olan önceki bir transaction faaliyet alanına katılıp katılmayacağı gibi seçenekler belirlenir. Örneğin aşağıdaki basit yapıyı ele alalım. Burada ilk using bloğu ile bir Transaction Scope oluşturulur. İkinci using ifadesine gelindiğinde yeni transaction scope' un önceki transaction scope' a ilave edileceği TransactionScopeOption parametresi ile belirlenir. Nitekim **Required** değeri, yeni scope' u var olan önceki scope' a ekler. Eğer var olan bir scope yok ise yeni bir tane oluşturur. Elbetteki burada akla gelen soru scope içindeki transaction' ların kimin tarafından onaylanacağıdır. Burada root scope kim ise ona ait Complete metodu devreye girecektir.

```

using(TransactionScope faaliyetAlani1 = new TransactionScope())
{
    ...
    using(TransactionScope faaliyetAlani2 = new
TransactionScope(TransactionScopeOption.Required))
    {
        ...
    }
}

```

Transaction Scope 1 (default olarak Required)



Şimdi yukarıdaki nested scope yapısı içine üçüncü bir scope daha ilave edelim. Ancak yeni TransactionScope için TransactionScopeOption değerini **RequiresNew** olarak belirleyelim.

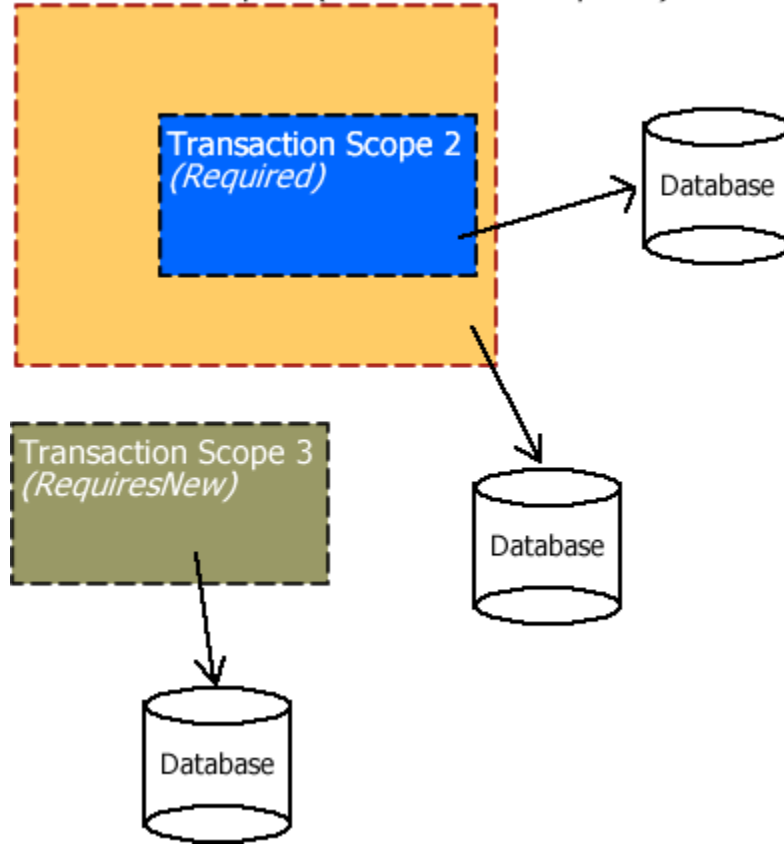
```

using(TransactionScope faaliyetAlani1 = new TransactionScope())
{
    ...
    using(TransactionScope faaliyetAlani2 = new
TransactionScope(TransactionScopeOption.Required))
    {
        ...
    }
    using(TransactionScope faaliyetAlani3 = new
TransactionScope(TransactionScopeOption.RequiresNew))
    {
        ...
    }
}

```

Bu durumda yapımız aşağıdaki gibi olacaktır.

Transaction Scope 1 (*default olarak required*)



Dilersek transaction' ları açıkça(explicit) kendimizde manüel olarak oluşturabiliriz. Şu ana kadar yaptığımız örneklerde implicit bir yaklaşım izledik. Yani ilgili transaction ve bunlara ait kaynakların otomatik olarak oluşturulmasını sağladık. Örneğin aşağıdaki kodlarda transaction' lar manuel olarak oluşturulmuştur. (*Örnek .Net 2.0 Beta sürümünde denenmiştir.*)

```

ICommittableTransaction trans = Transaction.Create();
try
{
    using (SqlConnection conNorthwind = new SqlConnection("data
source=localhost;database=Northwind;integrated security=SSPI"))
    {
        SqlCommand cmdInsert = new SqlCommand("INSERT INTO Personel (AD,SOYAD)
VALUES ('Burak Selim','Şenyurt'", conNorthwind);
        conNorthwind.Open();
        conNorthwind.EnlistTransaction(trans);
        cmdInsert.ExecuteNonQuery();
    }
    using (SqlConnection conAdv = new SqlConnection("data
source=localhost;database=AdventureWorks;integrated security=SSPI"))
    {

```

```

        SqlCommand cmdInsert = new SqlCommand("INSERT INTO Personel
(AD,SOYAD,MAIL) VALUES ('Cimi','Keri','cimi@keri.com')", conAdv);
        conAdv.Open();
        conAdv.EnlistTransaction(trans);
        cmdInsert.ExecuteNonQuery();
    }
    trans.Commit();
}
catch
{
    trans.Rollback();
}

```

ICommittableTransaction arayüzü bir Transaction Scope' un oluşturulmasını sağlar. Bunun için Transaction sınıfına ait Create metodu kullanılır. Create metodu varsayılan ayarları ile birlikte bir Scope oluşturacaktır. Eğer bu Scope' a transaction' lar eklemek istersek, ilgili bağlantıları temsil eden Connection nesnelerinin **EnlistTransaction** metodunu kullanırız. EnlistTransaction metodu parametre olarak transaction Scope' u temsil eden **ICommittableTransaction** arayüzü tipinden nesne örneğini alır. Elbette arayüze eklenen transaction' lara ait işlemlerin onaylanmasını sağlamak için arayüze ait **Commit** metodu kullanılır. Tam tersine bir sorun çıkar ve veritabanına doğru yapılan işlemlerden birisi gerçekleştirilemez ise o ana kadar yapılan işlemlerin geri alınması **ICommittableTransaction** arayüzüne ait **RollBack** metodu ile sağlanmış olur.

Bu makalemizde Transaction mimarisinin Ado.Net 2.0' daki yüzünü incelemeye çalıştık. Görüldüğü gibi kod yazımının basitleştirilmesinin yanında, özellikle EnterpriceServices bağımlılığından kurtularak Distributed Transaction' ların otomatik hale getirilmesi ve Transaction Scope kavramının getirilmesi göze çarpan önemli özellikler. Burada bahsedilen özellikler teorik olarak fazla bir değişikliğe uğramayacaktır. *Ancak bazı üyelerin isimlerin değişiklik beklenmektedir.* Örneğin ICommittableTransaction arayüzü yerine CommittableTransaction sınıfının geleceği düşünülmektedir. Bir sonraki makalemizde görüşmek dileğiyle hepinize mutlu günler dilerim.

Ado.Net 2.0 ve Data Provider- Independent Mimari - 24 Nisan 2005 Pazar

ado.net 2.0, data providers, dbconnection, dbcommand, dbdataadapter, dbproviderfactory,

Değerli Okurlarım, Merhabalar.

Veritabanı uygulamalarında başımızı ağrıtan noktalardan bir tanesi farklı tipte veritabanı sistemleri kullanan uygulamaların geliştirilmesi sırasında ortaya çıkar. Çoğu zaman geliştirdiğimiz bir ürün Sql sunucuları üzerinde yüksek performans gösterecek şekilde çalışmak zorunda iken, aynı ürünün Oracle üzerinde çalıştırılması da istenebilir. Bu durumda ortak bir çözüm olarak OleDb isim alanı altındaki sınıfları kullanmak oldukça mantıklıdır. Çünkü OleDb üzerinden her iki veri sunucusu için gerekli olan veri sağlayıcılarını kullanabiliriz.

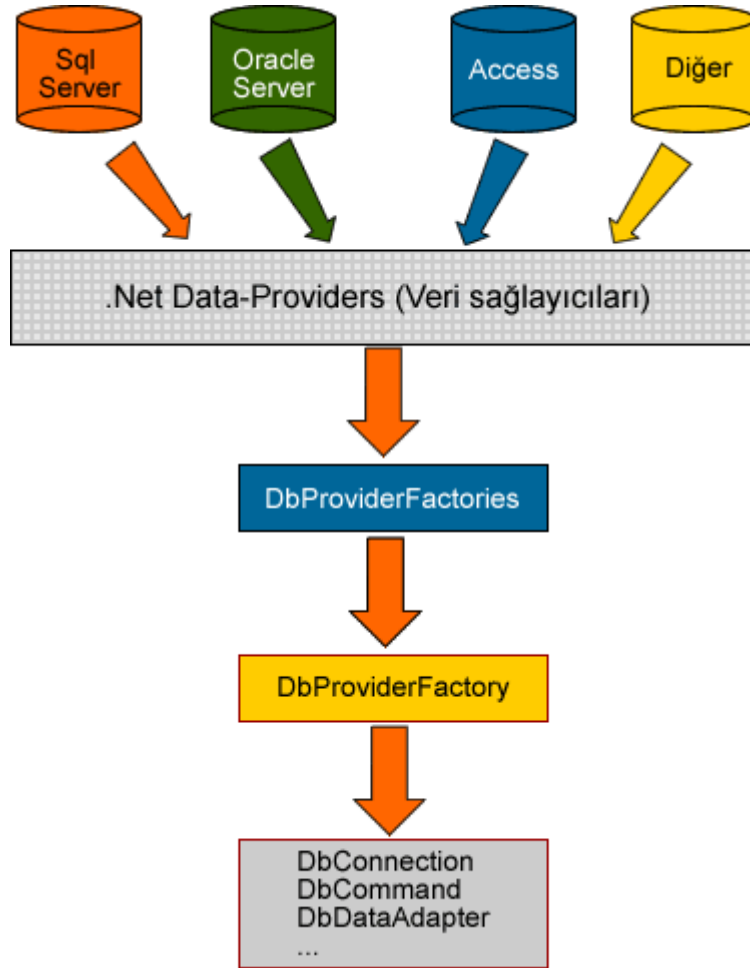
Diğer yandan böyle bir yol izlendiğinde direkt olarak OracleClient veya SqlClient isim alanını kullanarak kazanılacak performans avantajı ortadan kaybolacaktır. Peki .Net Framework için geliştirilen başka bir veri sağlayıcı (data-provider) işin içine girerse ne olacaktır. Bu durumda uygulama kodunda ilgili veri sağlayıcısına destek verecek şekilde düzenlemeler yapmamız gerekecektir. Öyleki, SqlClient isim alanı için gerekli Connection nesnesi ile OracleClient isim alanı için gerekli Connection nesnelerinin isimleri farklıdır. Aynı durum Command nesnelerinden tutun da DataReader nesnelerine kadar geçerlidir. İşte bu durum kodlarımızı her veri sağlayıcı için ayrı şekilde düzenlememizi gerektirebilir.

Diğer yandan System.Data.Common isim alanındaki sınıfları kullanarak farklı veri sağlayıcılarına destek verebilecek katmanları geliştirebiliriz. Bu Ado.Net 1.1' de çok esnek olmayan bir mimari üzerinde gerçekleştirilmektedir. Oysa ki Ado.Net 2.0, System.Data.Common isim alanına bir takım yeni özellikler eklemiştir. Bu özellikler arasında yeni eklenen iki sınıf büyük öneme sahiptir.

DbProviderFactories ve DbProviderFactory sınıfları. Ado.Net 2.0' da bu sınıfların System.Data.Common isim alanına eklenmelerinin en büyük nedeni, ilgili sistemlerde yüklü olan veri sağlayıcılarının öğrenilebilmesi ve seçilen herhangi bir veri sağlayıcısına özel Command, Connection, DataAdapter gibi veri üzerinde iş yapmamızı sağlayacak sınıfların tekil isimler altında örneklendirilebilmesidir.

Yeni eklenen özelliklerin sağladığı bu imkanlar sayesinde, geliştirilmiş olan bir ürünün farklı veri sağlayıcılar için destek verebilecek şekilde inşa edilmesi daha da kolaylaştırılmıştır. İşin güzel yanı, bu mimarinin performans olarak asıl veri sağlayıcılarına oranla oldukça iyi sonuçlar veriyor olmasıdır. İşte bu makalemizde System.Data.Common isim alanına eklenen bu yeni sınıflar ile neler

gerçekleştirebileceğimizi incelemeye çalışacağız. Yeni eklenen sınıfların önemini daha iyi anlayabilmek için aşağıdaki şekli göz önüne alabiliriz.



Aslında uygulamamıza katacağımız veri sağlayıcıdan bağımsız mimari için, yukarıdaki yolu izlememiz yeterli olacaktır. İlk olarak sistemde yüklü olan veri sağlayıcılarını elde edebiliriz. Ya da bunun seçimini kullanıcıya bırakabiliriz. Hangisinin kullanılacağına biz, kullanıcı veya sistemin kendisi karar verebilir. Son olarak seçilen veri sağlayıcı üzerinden gerçekleştireceğimiz veritabanı işlemleri için (örneğin kaynağa bağlantı açmak, komut yürütmek gibi) gerekli olan üreticiyi (ki bu DbProviderFactory sınıfıdır) hazırlarız. Daha sonra bu yeni sınıf yardımıyla gerekli olan nesneleri üretip kullanırız. Dilerseniz System.Data.Common isim alanına eklenen bu yeni iki sınıfı incelemekle işe başlayalım.

Öncelikle **DbProviderFactories** sınıfını ele alalım. Bu sınıf sistemde yüklü olan veri sağlayıcılarını elde etmemizi sağlayan **GetFactoryClasses** isimli static bir metoda sahiptir. Bu sınıfın diğer bir static metodu ise **GetFactory**'dir. GetFactory metodu geriye **DbProviderFactory** tipinden bir nesne örneğini döndürür. DbProviderFactory tipinden nesne örnekleri yoluyla, **DbConnection**, **DbCommand** gibi nesneleri elde edebiliriz. Dolayısıyla DbProviderFactories sınıfı yardımıyla bir sistemdeki veri sağlayıcılarını elde edebilir ve seçilen bir veri

sağlayıcısı için gerekli nesneleri üretmemizi sağlayacak DbProviderFactory sınıfını örnekleyebiliriz. İlk olarak aşağıdaki kod parçasını ele alalım.

```
using System.Data;  
using System.Data.Common;
```

```
#endregion
```

```
namespace UsingDbProviderFactories  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            DataTable dtProviders = new DataTable();  
            dtProviders = DbProviderFactories.GetFactoryClasses();  
            foreach (DataRow drProvider in dtProviders.Rows)  
            {  
                for (int i = 0; i < dtProviders.Columns.Count; i++)  
                {  
                    Console.WriteLine(drProvider[i].ToString());  
                }  
                Console.WriteLine("-----");  
            }  
            Console.ReadLine();  
        }  
    }  
}
```



```
file:///D:/Whidbey/AdoNet2/UsingDbProviderFactories/UsingDbProviderFactories/bin/...
Odbc Data Provider
.NET Framework Data Provider for Odbc
System.Data.Odbc
System.Data.Odbc.OdbcFactory, System.Data, Version=2.0.3600.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
191
-----
OleDb Data Provider
.NET Framework Data Provider for OleDb
System.Data.OleDb
System.Data.OleDb.OleDbFactory, System.Data, Version=2.0.3600.0, Culture=neutral
, PublicKeyToken=b77a5c561934e089
191
-----
OracleClient Data Provider
.NET Framework Data Provider for Oracle
System.Data.OracleClient
System.Data.OracleClient.OracleClientFactory, System.Data.OracleClient, Version=
2.0.3600.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
191
-----
SqlClient Data Provider
.NET Framework Data Provider for SqlServer
System.Data.SqlClient
System.Data.SqlClient.SqlClientFactory, System.Data, Version=2.0.3600.0, Culture
=neutral, PublicKeyToken=b77a5c561934e089
255
-----
SQL Server CE Data Provider
.NET Framework Data Provider for Microsoft SQL Server 2005 Mobile Edition
Microsoft.SqlServerCe.Client
Microsoft.SqlServerCe.Client.SqlCeClientFactory, Microsoft.SqlServerCe.Client, V
ersion=9.0.242.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91
1015
-----
```

Bu kod ile, sistemimizde yüklü olan veri sağlayıcılarını elde etmiş olduk. Dikkat ederseniz GetFactoryClasses metodundan dönen değer **DataTable** tipinden bir nesne örneğidir. Aynı örneği bir windows uygulamasında ele aldığımızda bu geri dönüş tipinden yararlanarak sonuçları veri bağlı kontroller üzerinde (datagridview gibi) gösterebiliriz.

```
DataTable dtProviders = new DataTable();
dtProviders = DbProviderFactories.GetFactoryClasses();
grdProviders.DataSource=dtProviders;
```

Sistemde Yüklü Data-Provider' lar					
	Name	Description	InvariantName	AssemblyQualifiedName	SupportedClasses
▶	Odbc Data Provider	.Net Framework Data Provider for Odbc	System.Data.Odbc	System.Data.Odbc.OdbcFac System.Data, Version=2.0.3600.0, Culture=neutral,	191
	OleDb Data Provider	.Net Framework Data Provider for OleDb	System.Data.OleDb	System.Data.OleDb.OleDbF System.Data, Version=2.0.3600.0, Culture=neutral,	191
	OracleClient Data Provider	.Net Framework Data Provider for Oracle	System.Data.OracleClient	System.Data.OracleClient.C System.Data.OracleClient, Version=2.0.3600.0, Culture=neutral,	191
	SqlClient Data Provider	.Net Framework Data Provider for SqlServer	System.Data.SqlClient	System.Data.SqlClient.SqlCl System.Data, Version=2.0.3600.0, Culture=neutral,	255
	SQL Server CE Data Provider	.NET Framework Data Provider for Microsoft SQL Server 2005 Mobile Edition	Microsoft.SqlServerCe.Client	Microsoft.SqlServerCe.Clien Microsoft.SqlServerCe.Clien Version=9.0.242.0, Culture=neutral,	1015

Peki sistemdeki veri sağlayıcılarının elde edilmesinin bize sağlayacağı avantajlar neler olabilir? Herşeyden önce ürünümüzün yüklendiği sistemlerdeki veri sağlayıcılarını görmek ve bunlardan seçişi olan ile uygulamayı çalıştırmak isteyebiliriz. Böyle bir durumda GetFactoryClasses metodu işimizi oldukça kolaylaştıracaktır. Diğer yandan, ürünümüzü sisteme yüklerken kurulan herhangi bir konfigürasyon ayarı ile de bir veri sağlayıcıyı seçebiliriz. Çoğunlukla bunu xml içerikli konfigürasyon dosyalarında belirtiriz. (Örneğin app.config dosyası içinde). Uygulamanın hangi veri sağlayıcısını baz alarak devam edeceğine bu dosyadaki ilgili konfigürasyon ayarından karar verebiliriz.

Elbette böyle bir durumda uygulamanın yüklendiği sistemde seçilen veri sağlayıcısının olup olmadığına bakmak için yine yukarıdaki teknik ile elde edilen DataTable nesnesinden faydalanabiliriz. Bu sayede sistemde yüklü olmayan bir veri sağlayıcısı ile devam edilmesini de henüz kurulum aşamasında engellemiş oluruz. Bunun sonrasında kullanıcıya kullanabileceği veri sağlayıcıları alternatif olarak sunabiliriz ve uygun olanı ile devam etmesini sağlayabiliriz. Gelelim, DbProviderFactory sınıfına. Bu sınıf, veritabanına bağlantı açma, sql komutu çalıştırmak gibi işlemleri yürütmemizi sağlayacak DbConnection, DbCommand gibi sınıfların üretilmesini sağlar. Bu sınıfın prototipi aşağıdaki gibidir.

```
public abstract class DbProviderFactory
```

Görüldüğü gibi DbProviderFactory abstract (soyut) bir sınıftır. Dolayısıyla bu sınıfa ait bir nesne örneğini üretemeyiz. Ancak DbProviderFactories sınıfımıza ait olan **GetFactory** static metodu bizim kullanabileceğimiz bir DbProviderFactory nesnesini sağlayacaktır. GetFactory metodunun aşırı yüklenmiş(overload) iki versiyonu vardır.

```
public static DbProviderFactory GetFactory(DataRow data-provider için dataRow);
public static DbProviderFactory GetFactory(string data-provider için invariant name);
```

Bu metodlardan ilki DataRow tipinden bir nesne örneğini alır. Bu DataRow nesnesinin temsil ettiği satır aslında DbProviderFactories sınıfının GetFactoryClasses metodundan dönen DataTable üzerindeki satırlardan herhangi biridir. Diğer yandan ikinci versiyonda string tipinden parametrenin alacağı değer, ilgili veri sağlayıcısının sistemdeki sabit adıdır (invariant-name).

```
DbProviderFactory fakto = DbProviderFactories.GetFactory("System.Data.SqlClient");
```

Elde ettiğimiz DbProviderFactory nesnesi vasıtasıyla artık veritabanı uygulamamız için gerekli nesneleri örnekleyebiliriz. Örneğin aşağıdaki kod parçası ile bir DbConnection nesnesi elde edilmektedir. DbConnection nesnesi tahmin edeceğimiz gibi ilgili veri kaynağına doğru bağlantı hattı tesis etmemizi sağlar.



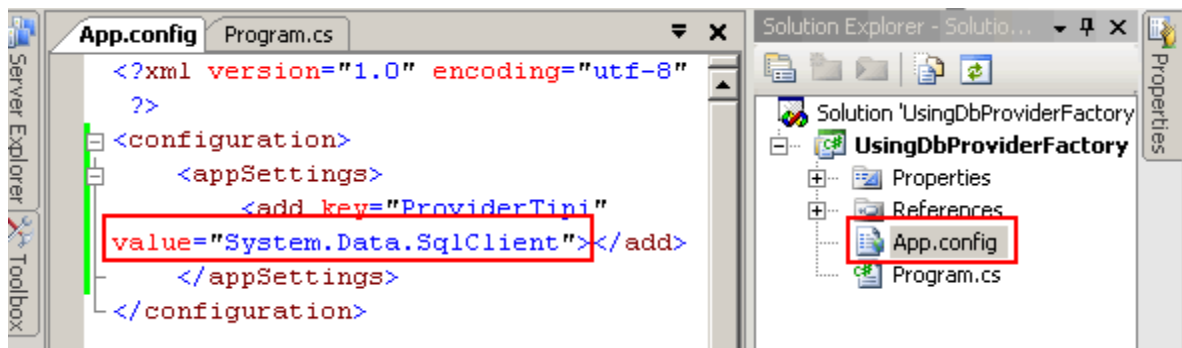
DbConnection, DbCommand, DbDataAdapter vb. abstract sınıflardır. Yani aslında bu sınıflara ait nesne örneklerini new operatörü yardımıyla oluşturamayız. Bu sınıflardan faydalanabilmek için DbProviderFactories sınıfının ilgili Create metodlarını kullanırız.

Kod parçamız ;

```
DbProviderFactory fakto = DbProviderFactories.GetFactory("System.Data.SqlClient");
DbConnection con = fakto.CreateConnection();
con.ConnectionString="data source=localhost;database=AdventureWorks;integrated
security=SSPI";
con.Open();
con.Close();
Console.Read();
...
```

Bu kod parçasında SqlConnection veri sağlayıcısını kullanacak şekilde bir DbConnection nesnesi elde edilmektedir. DbConnection nesnesini elde edebilmek için DbProviderFactory sınıfına ait CreateConnection metodu kullanılmaktadır. Ne yazık ki veri sağlayıcı bağımsız mimarinin de içinden şu an için çıkamayacağı sorunlar var. Bunlardan birisi ConnectionString' in bir veri sağlayıcıdan ötekine farklılık göstermesidir.

Yani Sql sunucularına SqlConnection nesnesi ile bağlantı kurarken kullandığımız Connection String ifadesi, OleDbConnection için olandan farklıdır. Bu sorunu çözmek için DbConnectionStringBuilder sınıfı kullanılmaktadır. Bu sınıf bir connection string içine yazılan özellikler anahtar-değer (key-value) çiftleri şeklinde temsil eder. Böylece uygun Connection String elde edilebilir. Ancak tabiki öncesinde seçilen veri sağlayıcısının her durumda kontrol edilmesi gerekecektir. Aşağıdaki kod parçası hem SqlConnection hem de OleDb için gerekli DbConnection nesnesinin doğru bir şekilde elde edilebilmesini sağlamaktadır. *(Burada veri sağlayıcısının seçimi için app.config dosyasını kullandığımıza dikkat edin.)*



#region Using directives

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data.Common;
using System.Configuration;
```

#endregion

```
namespace UsingDbProviderFactory
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            DbProviderFactory fakto;
            DbConnection con;
            DbConnectionStringBuilder conStr = new DbConnectionStringBuilder();
            string secilenProvider = ConfigurationSettings.AppSettings["ProviderType"];
            fakto = DbProviderFactories.GetFactory(secilenProvider);
            con = fakto.CreateConnection();
            if (secilenProvider == "System.Data.SqlClient")
            {
                conStr.Add("data source", "localhost");
                conStr.Add("database", "AdventureWorks");
                conStr.Add("integrated security", "SSPI");
            }
            else if (secilenProvider == "System.Data.OleDb")
            {
                conStr.Add("Provider", "SqlOleDb");
                conStr.Add("data source", "localhost");
                conStr.Add("database", "AdventureWorks");
                conStr.Add("integrated security", "SSPI");
            }
            else
            {
                Console.WriteLine("Doğru provider seçilmedi...");
            }
        }
    }
}
```

```

    }
    con.ConnectionString = conStr.ConnectionString;
    try
    {
        con.Open();
        Console.WriteLine("Bağlantı açıldı...");
        Console.ReadLine();
    }
    catch(Exception hata)
    {
        Console.WriteLine(hata.Message.ToString());
    }
    finally
    {
        con.Close();
    }
}
}
}

```

Uygulamamızda başlangıç olarak veri sağlayıcısını Sql sunucusuna direkt erişim sağlayan SqlClient olarak belirledik. If koşullarında app.config dosyasına eklediğimiz ProviderTipi anahtarının değerine bakarak uygun Connection String ifadesinin oluşturulmasını sağlıyoruz. Eğer OleDb kaynağını kullanarak erişim sağlamak istersek tek yapmamız gereken app.config dosyasında ProviderTipi anahtarının değerini System.Data.OleDb olarak değiştirmek olacaktır.



Seçilen veri sağlayıcısının (Data-Provider) ismi sistem de yüklü olan sabit isimdir. (Invariant Name)

Buradaki anahtarların değerlerinin sistemde tanımlı olan invariant-name değerleri olduğunu hatırlatalım. Aslında sistemde yüklü olan veri sağlayıcılarının özellikleri elde edilirken machine.config dosyasındaki ayarlara bakılır. Eğer D:\WINDOWS\Microsoft.NET\Framework\v2.0.40607\CONFIG (Windows 2003 için) adresinden **machine.config** dosyasına bakılırsa sistemde yüklü olan veri sağlayıcılarının listesinin **DbProviderFactories** takısında yer aldığını görebiliriz. İşte veri sağlayıcılarının sabit isimleri buradan alınmaktadır.

```
machine.config App.config Program.cs*
<compiler language="c++7;mc7;cpp7" extension=".h" type="Microsoft.
VisualC.CppCodeProvider7, CppCodeProvider, Version=8.0.1200.0, Culture=
neutral, PublicKeyToken=b03f5f7f11d50a3a" />
</compilers>
</system.codedom>
<system.data>
  <DbProviderFactories>
    <add name="Odbc Data Provider" invariant="System.Data.Odbc" support
="BF" description=".Net Framework Data Provider for Odbc" type="System.
Data.Odbc.OdbcFactory, System.Data, Version=2.0.3600.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" />
    <add name="OleDb Data Provider" invariant="System.Data.OleDb"
support="BF" description=".Net Framework Data Provider for OleDb" type=
"System.Data.OleDb.OleDbFactory, System.Data, Version=2.0.3600.0, Culture
=neutral, PublicKeyToken=b77a5c561934e089" />
    <add name="OracleClient Data Provider" invariant="System.Data.
OracleClient" support="BF" description=".Net Framework Data Provider for
Oracle" type="System.Data.OracleClient.OracleClientFactory, System.Data.
OracleClient, Version=2.0.3600.0, Culture=neutral, PublicKeyToken=
b77a5c561934e089" />
    <add name="SqlClient Data Provider" invariant="System.Data.
SqlClient" support="FF" description=".Net Framework Data Provider for
SqlServer" type="System.Data.SqlClient.SqlClientFactory, System.Data,
Version=2.0.3600.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
    <add name="SQL Server CE Data Provider" invariant="Microsoft.
SqlServerCe.Client" support="3F7" description=".NET Framework Data
Provider for Microsoft SQL Server 2005 Mobile Edition" type="Microsoft.
SqlServerCe.Client.SqlCeClientFactory, Microsoft.SqlServerCe.Client,
Version=9.0.242.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91" />
  </DbProviderFactories>
</system.data>
</!--
```

DbConnection nesnesi gibi **DbCommand** nesneside veri sağlayıcıdan bağımsız komut setlerinin yürütülmesine imkan sağlamaktadır. Bir DbCommand nesnesinin oluşturuluş biçimi DbConnection nesnesinde olduğu gibidir. Yani bu nesneyi elde edebilmek için DbProviderFactory sınıfının ilgili Create metodunu aşağıdaki gibi kullanırsınız.

```
DbProviderFactory fakto;
DbConnection con;
DbCommand cmd;
DbConnectionStringBuilder conStr = new DbConnectionStringBuilder();
string secilenProvider = ConfigurationSettings.AppSettings["ProviderTipi"];
fakto = DbProviderFactories.GetFactory(secilenProvider);
con = fakto.CreateConnection();
cmd = fakto.CreateCommand();
if (secilenProvider == "System.Data.SqlClient")
{
    conStr.Add("data source", "localhost");
    conStr.Add("database", "AdventureWorks");
    conStr.Add("integrated security", "SSPI");
}
```

```

}
else if (secilenProvider == "System.Data.OleDb")
{
    conStr.Add("Provider", "SqlOleDb");
    conStr.Add("data source", "localhost");
    conStr.Add("database", "AdventureWorks");
    conStr.Add("integrated security", "SSPI");
}
else
{
    Console.WriteLine("Doğru provider seçilmedi...");
}
con.ConnectionString = conStr.ConnectionString;
try
{
    con.Open();
    cmd.Connection = con;
    cmd.CommandText = "INSERT INTO Personel (AD,SOYAD,MAIL) VALUES ('Burak
Selim','Şenyurt','selim(at)buraksenyurt.com')";
    int eklenen=cmd.ExecuteNonQuery();
    Console.WriteLine("Bağlantı açıldı...");
    Console.WriteLine(eklenen + " SATIR EKLENDİ");
    Console.ReadLine();
}
catch(Exception hata)
{
    Console.WriteLine(hata.Message.ToString());
}
finally
{
    con.Close();
}

```

Görüldüğü gibi tek yapmamız gereken DbProviderFactory sınıfının **CreateCommand** metodunu kullanarak bir DbCommand nesnesini elde etmektir. Daha sonra bu komut nesnesinin kullanacağı sorgu ve bağlantı her zamanki yöntemlerimiz ile belirlenmiştir. Elbetteki, DbConnection nesnesi oluşturulurken yaşanan problemin benzeri burada da söz konusudur. Bu kez Command nesnelerinin parametre isimlendirmeleri bir veri sağlayıcıdan ötekine farklılık göstermektedir.

Örneğin SqlCommand nesnesinde kullanılan parametreler @ ile başlarken, OleDbCommand nesnelerinde sorgu cümlecisindeki parametreler ? ile tanımlanmak zorundadır. Bu sorunu yine yukarıdaki if yapısı ile halledebiliriz. Söz konusu olan parametreleri DbParameter sınıfı ile tanımlayabiliriz. Dolayısıyla yukarıdaki örneği aşağıdaki gibi yazarsak parametre farklılığını bu örnekte geçerli olan veri sağlayıcıları için çözmüş oluruz.

```

static void Main(string[] args)
{

```

```

DbProviderFactory fakto;
DbConnection con;
DbCommand cmd;
DbParameter prmAd;
DbParameter prmSoyad;
DbParameter prmMail;
DbConnectionStringBuilder conStr = new DbConnectionStringBuilder();
string secilenProvider = ConfigurationSettings.AppSettings["ProviderTipi"];
string cmdQuery;
fakto = DbProviderFactories.GetFactory(secilenProvider);
con = fakto.CreateConnection();
cmd = fakto.CreateCommand();
prmAd = fakto.CreateParameter();
prmSoyad = fakto.CreateParameter();
prmMail = fakto.CreateParameter();
if (secilenProvider == "System.Data.SqlClient")
{
    conStr.Add("data source", "localhost");
    conStr.Add("database", "AdventureWorks");
    conStr.Add("integrated security", "SSPI");
    cmdQuery = "INSERT INTO Personel (AD,SOYAD,MAIL) VALUES
(@AD,@SOYAD,@MAIL)";
    prmAd.ParameterName = "@AD";
    prmAd.DbType = DbType.String;
    prmAd.Size = 50;
    cmd.Parameters.Add(prmAd);
    prmSoyad.ParameterName = "@SOYAD";
    prmSoyad.DbType = DbType.String;
    prmSoyad.Size = 50;
    cmd.Parameters.Add(prmSoyad);
    prmMail.ParameterName = "@MAIL";
    prmMail.DbType = DbType.String;
    prmMail.Size = 50;
    cmd.Parameters.Add(prmMail);
    cmd.Connection = con;
    cmd.CommandText = cmdQuery;
}
else if (secilenProvider == "System.Data.OleDb")
{
    conStr.Add("Provider", "SqlOleDb");
    conStr.Add("data source", "localhost");
    conStr.Add("database", "AdventureWorks");
    conStr.Add("integrated security", "SSPI");
    cmdQuery = "INSERT INTO Personel (AD,SOYAD,MAIL) VALUES (?, ?, ?)";
    prmAd.DbType = DbType.String;
    prmAd.Size = 50;
    cmd.Parameters.Add(prmAd);
    prmSoyad.DbType = DbType.String;

```



```

        prmSoyad.Size = 50;
        cmd.Parameters.Add(prmSoyad);
        prmMail.DbType = DbType.String;
        prmMail.Size = 50;
        cmd.Parameters.Add(prmMail);
        cmd.Connection = con;
        cmd.CommandText = cmdQuery;
    }
    else
    {
        Console.WriteLine("Doğru provider seçilmedi...");
    }
    con.ConnectionString = conStr.ConnectionString;
    try
    {
        con.Open();
        prmAd.Value = "Burak Selim";
        prmSoyad.Value = "Şenyurt";
        prmMail.Value = "selim(at)buraksenyurt.com";
        int eklenen=cmd.ExecuteNonQuery();
        Console.WriteLine("Bağlantı açıldı...");
        Console.WriteLine(eklenen + " SATIR EKLENDİ");
        Console.ReadLine();
    }
    catch(Exception hata)
    {
        Console.WriteLine(hata.Message.ToString());
    }
    finally
    {
        con.Close();
    }
}

```

DbProviderFactory sınıfı ayrıca DbDataAdapter nesnelerini elde edebilmemizi sağlayan **CreatedDataAdapter** isimli bir metoda sahiptir. Bu sayede bağlantısız katman nesneleri ile veri kaynağı arasındaki veri taşıma işlemlerini gerçekleştirebileceğimiz DataAdapter nesne örneklerini veri sağlayıcısından bağımsız olacak şekilde elde edebiliriz. Aşağıdaki kod parçası ile bu işlemin nasıl gerçekleştirilebileceğini görmekteyiz. Bu örnekte kullanıcının seçtiği veri sağlayıcısının sistemde yüklü olup olmadığını kontrol ediyoruz. Örneğimizde veri sağlayıcılarını OleDb ve SqlClient ile sınırladık.

#region Using directives

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;

```

```

using System.Data.Common;
using System.Drawing;
using System.Windows.Forms;

#endregion

namespace UsingDbDataAdapter
{
    partial class Form1 : Form
    {
        private DbProviderFactory fakto;
        private DbConnection con;
        private DbConnectionStringBuilder conStr;
        private DbCommand cmd;
        private DbDataAdapter da;
        private DataTable dt;

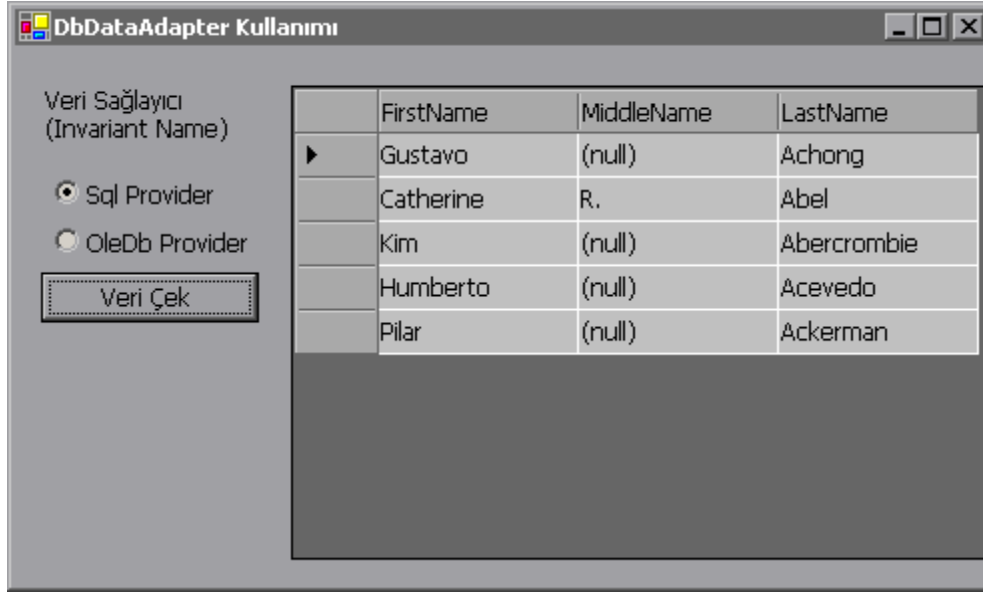
        public Form1()
        {
            InitializeComponent();
        }
        private bool VeriSaglayiciKontrol(string invariantName)
        {
            int varmi = DbProviderFactories.GetFactoryClasses().Select("InvariantName='" +
invariantName + "'").Length;
            if (varmi == 0)
                return false;
            else
                return true;
        }
        private void DbConnectionOlustur(DbConnectionStringBuilder connectionString)
        {
            con = fakto.CreateConnection();
            con.ConnectionString = connectionString.ConnectionString;
        }
        private void DbCommandOlustur(string sqlQuery)
        {
            cmd = fakto.CreateCommand();
            cmd.Connection = con;
            cmd.CommandText = sqlQuery;
        }
        private void FactoryOlustur(string veritabaniAdi)
        {
            if (radOleDb.Checked == true)
            {
                if (VeriSaglayiciKontrol("System.Data.OleDb"))
                {
                    fakto = DbProviderFactories.GetFactory("System.Data.OleDb");
                }
            }
        }
    }
}

```

```

        conStr = new DbConnectionStringBuilder();
        conStr.Add("provider", "SqlOleDb");
        conStr.Add("data source", "localhost");
        conStr.Add("database", veritabaniAdi);
        conStr.Add("integrated security", "SSPI");
        DbConnectionOlustur(conStr);
    }
}
if (radSql.Checked == true)
{
    if (VeriSaglayiciKontrol("System.Data.SqlClient"))
    {
        fakto = DbProviderFactories.GetFactory("System.Data.SqlClient");
        conStr = new DbConnectionStringBuilder();
        conStr.Add("data source", "localhost");
        conStr.Add("database", veritabaniAdi);
        conStr.Add("integrated security", "SSPI");
        DbConnectionOlustur(conStr);
    }
}
}
private void DbDataAdapterOlustur()
{
    da = fakto.CreateDataAdapter();
    da.SelectCommand = cmd;
}
private void btnVeriCek_Click(object sender, EventArgs e)
{
    FactoryOlustur("AdventureWorks");
    DbCommandOlustur("SELECT TOP 5 FirstName,MiddleName,LastName FROM
Person.Contact");
    DbDataAdapterOlustur();
    dt = new DataTable();
    da.Fill(dt);
    grdVeriler.DataSource = dt;
}
}
}

```



Uygulamamızı çalıştırdığımızda ister Sql veri sağlayıcısını ister OleDb veri sağlayıcısını seçelim aynı DbConnection, DbCommand ve DbDataAdapter nesneleri üzerinden işlem yapıyor olacağız. İşte bu yeni mimarinin bize sağladığı en büyük avantajdır.

Son olarak bir DataReader nesnesini veri sağlayıcıdan bağımsız mimaride nasıl kullanabileceğimizi inceleyeceğiz. Aslında şu an için bu mimaride geliştirilmiş bir DbDataReader sınıfı yok. Bunun yerine IDataReader arayüzünü kullanacağız. Dolayısıyla arayüzü kullanacağımız yere kadar yaptığımız işlemler yukarıdaki örnektekinden farksız olacak.

```
private void btnIDataReaderIleCek_Click(object sender, EventArgs e)
{
    FactoryOlustur("AdventureWorks");
    DbCommandOlustur("SELECT TOP 5 FirstName,MiddleName,LastName FROM
    Person.Contact");
    using (con)
    {
        con.Open();
        using (IDataReader dr = cmd.ExecuteReader())
        {
            while (dr.Read())
            {
                //bir takım okuma işlemleri.
            }
        }
    }
}
```

Ado.Net 2.0 için veri sağlayıcıdan bağımsız mimariyi kullanmak oldukça kolay ve yararlı görünüyor. Özellikle yeni eklenen sınıflar bu tarz yapıları oluşturmamızı son derece

kolaylaştırmış. Burada kafamızı kurcalayan tek konu performans farklılıkları. Microsoft kaynaklarına ve otoritelerine göre çok büyük performans kayıpları olmadığı (olmayacağı) söyleniyor. Açıkçası bu konunun ilerleyen zamanlarda daha da netleşeceği görüşündeyim. Yine de yeni kolaylıkların özellikle farklı veri sunucularına bağlanmamızı gerektiren durumlarda çok başarılı olacağını söyleyebilirim.

Herşeyden önemlisi, hangi veri sağlayıcısı olursa olsun aynı DbConnection, DbCommand veya DbDataAdapter nesnelerini kullanmak son derece büyük bir avantajdır. Mimarinin şu an için sadece başında olduğumuzu düşünüyorum. System.Data.Common isim alanına katılacak daha bir çok özellik olabilir. Örneğin buraya özel bir DbDataReader sınıfının oluşturulması gibi. Şunu da hatırlatmakta fayda var. Buradaki kod örnekleri ve kullanılan sınıflar, beta sürümüne aittir. Ürün piyasaya çıktığında bu sınıfların isimlerinde veya kullanış şekillerinde değişiklikler olabilir. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

Kendi İstisna Nesnelerimizi Kullanmak (ApplicationException) - 23 Mayıs 2005 Pazartesi

C#, exception handling, exception,

Değerli Okurlarım Merhabalar,

İstisna yakalama mekanizması (Exception Handling) dotNet mimarisinde oldukça önemli bir yere sahiptir. Bu mekanizma sayesinde uygulamalarımızın kilitlenmesi ve istem dışı bir şekilde sonlandırılmaya zorlanmasının önüne geçmiş oluruz. Framework içerisinde önceden tanımlanmış pek çok istisna sınıfı mevcuttur. Bu sınıflar yardımıyla, çalışma zamanında oluşabilecek istisnai durumlar kolayca tespit edilebilmektedir.

Böylece uygulamaların, CLR tarafından denetlendiği sırada ortama fırlatılan istisnalar nedeniyle yön değiştirebilmesi ve yaşamını sürdürebilmesi sağlanmış olmaktadır. Ancak bazı durumlarda kendi istisna sınıflarımızı yazma ihtiyacı duyabiliriz. Bunun pek çok nedeni olabilir. İlk ve en temel nedeni, sistemde var olan istisna sınıfları dışındaki bir istisnayı çalışma zamanında ele almak isteğimizdir. İşte bu makalemizde bu işlevselliği nasıl gerçekleştirebileceğimizi basit bir örnek üzerinde incelemeye çalışacağız.

.Net Framework' te var olan istisna sınıflarının tamamı System.Exception sınıfından dolayı bir şekilde türeyerek oluşturulmuşlardır. Kendi istisna nesnelerimizi oluşturabilmek ve yakalayabilmek için kullanacağımız sınıfları System isim alanında yer alan ApplicationException sınıfından türetiriz. (Aslında ApplicationException sınıfı da Exception sınıfından türemiştir.) Bu sayede throw anahtar sözcüğü vasıtasıyla oluşturduğumuz istisna nesnelerinin ortama fırlatılabilmesini sağlamış oluruz. Ortama fırlatılan istisna nesnelerini uygun try...catch bloklarında yakalayıp hem uygulamanın çökmesini engellemiş hem de kullanıcıların anlamlı mesajlar ile uyarılmasını sağlamış oluruz.

ApplicationException, kullanıcı tanımlı istisna sınıflarının tipik Exception sınıflarının sahip olduğu üyelerini kullanabilmesini sağlar. Bu tabiki kalıtımın bir sonucudur. Diğer yandan ApplicationException sınıfı, kendisinden türetilen sınıfın bir istisna nesnesi olduğunu ve throw anahtar sözcüğü ile fırlatılabileceğini de belirtir.

Konuyu daha iyi anlayabilmek amacıyla bir örnek üzerinden gideceğiz. Senaryomuzda aşağıda kodları bulunan Kitap isimli sınıfı kullanan bir uygulama yer alacak. Amacımız bu sınıfı kullanırken istisnaya neden olabilecek noktaları tespit etmek ve bu istisnaları yönetecek sınıfı yazarak uygulama içerisinde yön verebilmektir.

```

using System;

namespace KitapDukkani
{
    public class Kitap
    {
        private string m_Kitap_Yazar;
        private string m_Kitap_Baslik;
        private double m_Kitap_Fiyat;
        private DateTime m_Kitap_Basim;
        private string m_Kitap_Kategori;

        public string Yazar
        {
            get
            {
                return m_Kitap_Yazar;
            }
            set
            {
                m_Kitap_Yazar=value;
            }
        }
        public string Baslik
        {
            get
            {
                return m_Kitap_Baslik;
            }
            set
            {
                m_Kitap_Baslik=value;
            }
        }
        public double Fiyat
        {
            get
            {
                return m_Kitap_Fiyat;
            }
            set
            {
                m_Kitap_Fiyat=value;
            }
        }
        public DateTime Basim
        {
            get

```

```

        {
            return m_Kitap_Basim;
        }
        set
        {
            m_Kitap_Basim=value;
        }
    }
    public string Kategori
    {
        get
        {
            return m_Kitap_Kategori;
        }
        set
        {
            m_Kitap_Kategori=value;
        }
    }
}

public Kitap(string yazar,string baslik,double fiyat,DateTime basim,string kategori)
{
    Yazar=yazar;
    Baslik=baslik;
    Fiyat=fiyat;
    Basim=basim;
    Kategori=kategori;
}

public Kitap()
{
}
}
}

```

Kitap sınıfımız tipik olarak bir kitabın temel özelliklerini sunan bir yapıya sahiptir. Bu sınıfı kullanacak olan bir yazılım geliştiricinin dikkat edeceği bir takım noktalar olacaktır. Örneğin, kitabın fiyatının negatif değer almaması , harf yada karakterlerden oluşmaması, basım tarihinin mutlaka tarihsel formatta olması gerekliliği vb. Bunlar uygulamanın çalışması esnasında hataya neden olabilecek durumlardır. Var olan istisna sınıfları yardımıyla bu tip hataları çalışma zamanında bertaraf ederek uygulamanın yaşamına devam etmesini sağlayabiliriz. Bunun yanında bu sınıfı kullanacak olan yazılım geliştirici kullanıcının hataya neden olacak girişlerini engelleyecek tedbirleri elbette göz önüne alacaktır ve uygulayacaktır. Örneğin aşağıdaki windows uygulamasında kullanıcı girişlerinde oluşabilecek veri girişi hataları ele alınmaya çalışılmıştır.

İlk olarak alanlara girilecek olan karakter sayısı sınırlandırılarak çok uzun verilerin girilmeye çalışılması engellenebilir. Tarih girişlerinde oluşabilecek hataların önüne geçmek bir windows uygulaması için son derece kolaydır. DateTimePicker bileşeni bu kontrolü bizim için fazlasıyla sağlamaktadır. Kategori seçiminde ise ComboBox bileşeni kullanılabilir. Kategori verisinde aslında belirli bir listeden alınmak zorundadır. Bu liste bir veritabanından alınabileceği gibi, bir XML dosyasından da alınabilir vb...Bizim için hataya neden olabilecek bir diğer durumda Fiyat alanına girilecek sayısal değer karakter olarak girilmeye çalışılmasıdır. Bunu engelleyebilmek için sadece sayısal karakter girişine izin verecek bir metod kullanamız gerekir. Bunu gerçekleştirebileceğimiz en güzel yer ilgili TextBox kontrolünün KeyPress olay metodudur.

```
private void txtFiyat_KeyPress(object sender, System.Windows.Forms.KeyPressEventArgs e)
{
    if (((int)e.KeyChar < 48 || (int)e.KeyChar > 57) && ((int)e.KeyChar!=8))
    {
        e.Handled=true;
    }
}
```

Ama halen daha yazılım geliştiricinin bilmediği ve bu gibi durumlarda kullanıcının uyarılmasını istediğimiz istisnai durumlar olabilir. Örneğin fiyat alanına her ne kadar negatif değer girilemeyecek olsada, fiyat alanındaki güncel değer belli bir oranda azaltılmaya çalışıldığında negatif değer oluşabileceği görülebilir. Örneğin fiyat azaltımı için aşağıdaki metodu uyguladığımız varsayalım.

```
private void btnIndirim_Click(object sender, System.EventArgs e)
{
    if(kitap!=null)
    {
        kitap.Fiyat-=10;
        txtFiyat.Text=kitap.Fiyat.ToString();
    }
}
```

}

Bu durumda ekran görüntüsü aşağıdaki gibi olacaktır.

Görüldüğü gibi normal şartlar altında TextBox içerisine - karakterini basamasa da Kitap nesnemizi oluşturduktan sonra Fiyat özelliğinin değerinde yapacağımız 10 birimlik azaltmalar sonucu - değer görünebilmektedir. Bu bir bug olarak değerlendirilebilse de önüne geçmemiz gereken bir durumdur. Dahası Kitap sınıfını kullanan yazılım geliştirici bu tip bir kontrolü hiç yapmayabilir. İşte böyle bir durumda en azından girilen değer negatif olması durumunda ortama bir istisnanın fırlatılmasını sağlayabiliriz. Diğer yandan girilen Fiyat değerinin belli bir değerin üstünde olmamasını da isteyebiliriz. İşte bu iki basit nedeni ele alarak kendi istisna sınıfımızı yazabiliriz. İlk başta uygulamamızın Kitap nesnesi oluşturan kodunu standart try...catch yapısı içinde kullanamayı deneyelim.

```
private void btnOlustur_Click(object sender, System.EventArgs e)
```

```
{
    try
    {
        kitap=new Kitap();
        kitap.Baslik=txtBaslik.Text;
        kitap.Yazar=txtYazar.Text;
        kitap.Basim=dtgBasim.Value;
        kitap.Fiyat=Convert.ToDouble(txtFiyat.Text);
        kitap.Kategori=cmbKategori.SelectedText;
    }
    catch(System.Exception err)
    {
```

```
        MessageBox.Show(err.Message,"Hata",MessageBoxButtons.OK,MessageBoxIcon.Error);
    }
}
```

Bu haliyle uygulamayı çalıştırdığımızda ve fiyat alanı için eksi değere geçtiğimizde ortama herhangi bir istisna nesnesinin fırlatılmadığını görürüz. Çünkü oluşacak istisnai durum henüz tarafımızdan yaratılmamıştır. Dolayısıyla ApplicationException sınıfından türeteceğimiz bir exception sınıfı yazmamız gerekmektedir. İşte bu amacımızı sağlayan FiyatException isimli istisna sınıfımızın kodları;

```
using System;
```

```
namespace KitapDukkani
```

```
{
```

```
    /// <summary>
```

```
    /// Bir kitabın fiyatı ile ilgili istisna sınıfıdır.
```

```
    /// </summary>
```

```
    public class FiyatException:ApplicationException
```

```
    {
```

```
        /// <summary>
```

```
        /// İstisnaya neden olan fiyat değerini tutan özel field.
```

```
        /// </summary>
```

```
        private double m_Fiyat;
```

```
        /// <summary>
```

```
        /// İstisna ile ilgili kısa açıklama
```

```
        /// </summary>
```

```
        private string m_Mesaj;
```

```
        /// <summary>
```

```
        /// İstisna nesnesi oluşturulurken hataya neden olan fiyat değeri alınıp m_Fiyat özel alanına eşitlenir.
```

```
        /// </summary>
```

```
        /// <param name="fiyat">İstisnaya neden olan fiyat alanının değeridir.</param>
```

```
        public FiyatException(double fiyat,string mesaj)
```

```
        {
```

```
            m_Fiyat=fiyat;
```

```
            m_Mesaj=mesaj;
```

```
        }
```

```
        /// <summary>
```

```
        /// ApplicationException sınıfından gelen Message özelliği override edilmiştir. Geriye özel bir hata mesajı döndürmektedir.
```

```
        /// </summary>
```

```
        public override string Message
```

```
        {
```

```
            get
```

```
            {
```

```
                return "Kitaba ait fiyat değeri "+m_Fiyat+"belirtilen kriterlere uygun değildir."+"
```

```
                "+m_Mesaj+"";
```

```
            }
```

```
        }
```

```
    }
```

}

Görüldüğü gibi kendi yazdığımız istisna sınıflarının normal bir sınıfı yazmaktan hiç bir farkı yoktur. Kendi üyelerimizi ekleyebilir ve Exception sınıfından devralınan Message, TargetSite, InnerException gibi virtual özellikleri override edebiliriz. Sınıf tasarımını yaparken bu sınıfa ait nesne örneklerinin ne amaçla kullanılacağını belirlemeliyiz. FiyatException sınıfına ait bir nesne örneği, herhangi bir Kitap nesnesinin fiyatının bazı kriterlere uymaması durumunda fırlatılacaktır.



.Net içinde önceden tanımlanmış olan istisna sınıfları Exception kelimesi ile biterler. Kendi istisna sınıflarımızın isimlerinin de aynı şekilde yazılması kod standardizasyonu açısından önemlidir. Örneğin FiyatException gibi...

Dolayısıyla kriterlere uymayacak Fiyat değerini taşıması önemlidir. Ayrıca kriterin çeşitliliğine göre kullanıcıya verilmesi istenen mesaj içeriğinde bir field olarak saklanmalıdır. Bu iki field' ın alacağı değerleri ise constructor metodumuz içerisinde sağlayabiliriz. Oluşturduğumuz istisna nesnesinin tipik bir Exception nesnesi gibi davranabilmesini sağlamak amacıyla örnek olarak Message özelliği override edilmiştir. FiyatException sınıfı artık uygulama içerisinde yakalanabilecek tipik bir istisna halini almıştır. Ancak henüz uygulanmamıştır. Herşeyden önce istisnanın fırlatılmasını istediğimiz yerlerde bunları kodlamamız gerekecektir. Kitap nesnemizi göz önüne aldığımızda Fiyat özelliğinin değerinin verildiği set bloğu bu iş için biçilmiş kaftandır. Buradaki kodları aşağıdaki gibi düzenleyebiliriz.

```
public double Fiyat
{
    get
    {
        return m_Kitap_Fiyat;
    }
    set
    {
        if(value>150)
            throw new FiyatException(value,"Fiyat 150 YTL' den yüksek olamaz");
        if(value<0)
            throw new FiyatException(value,"Fiyat negatif değer olamaz");
        m_Kitap_Fiyat=value;
    }
}
```

Fiyat özelliğine değer atanırken eğer girilen değer 0' dan küçük ise buna uygun mesaja sahip bir istisna nesnesi, Fiyat 150 YTL' den büyük ise buna uygun mesaja sahip bir istisna nesnesi ortama fırlatılmaktadır. Böylece catch bloğunda FiyatException nesne örneklerini yakalayabiliriz. Kitap sınıfının yapıcı metodunda parametre üzerinden, sınıf içindeki alanlara

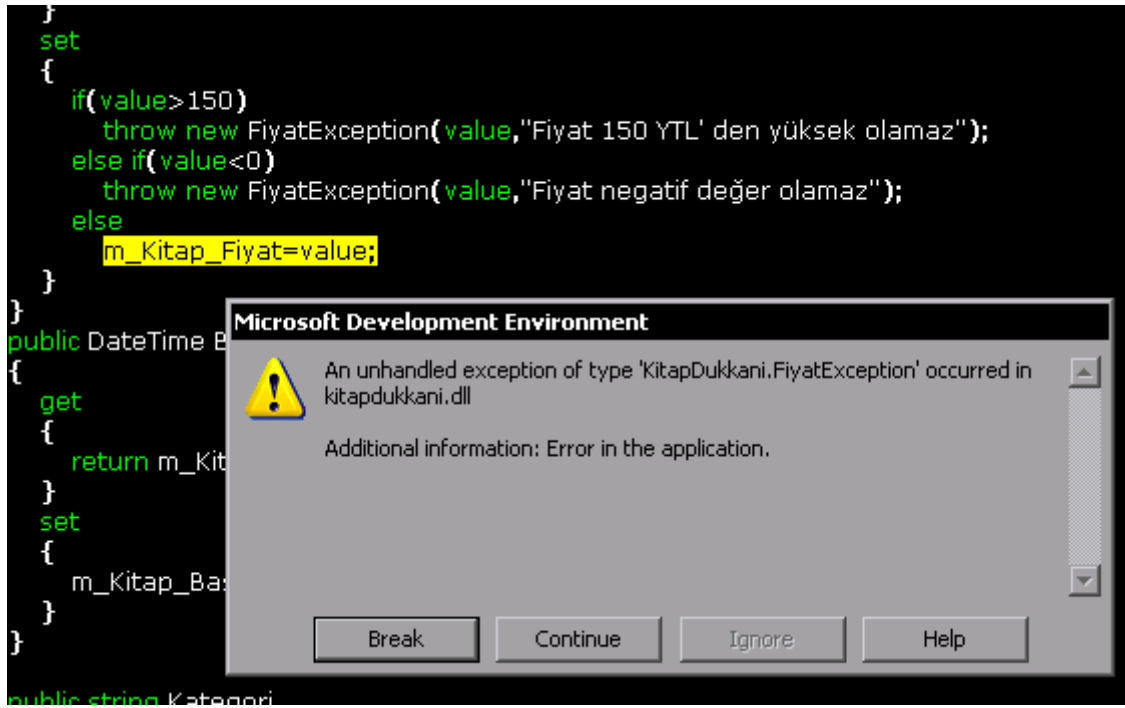
değer ataması yapılmaktadır. Ancak burada da parametre değerlerini direkt olarak özelliklere atadığımızda set blokları devreye girmektedir. Bu, Fiyat özelliğinin değeri için gerekli istisna kontrolünü yapıcı metod içerisinde de gerçekleştirebilmemizi sağlar. Kısacası bir taşla iki kuş vurmuş oluruz. Artık windows uygulamamızdaki kodları aşağıdaki gibi düzenleyebiliriz.

```
private void btnOlustur_Click(object sender, System.EventArgs e)
{
    try
    {
        kitap=new Kitap();
        kitap.Baslik=txtBaslik.Text;
        kitap.Yazar=txtYazar.Text;
        kitap.Basim=ctpBasim.Value;
        kitap.Fiyat=Convert.ToDouble(txtFiyat.Text);
        kitap.Kategori=cmbKategori.SelectedText;
    }
    catch(FiyatException err)
    {
        MessageBox.Show(err.Message,"Hata",MessageBoxButtons.OK,MessageBoxIcon.Error);
    }
    catch(System.Exception err)
    {
        MessageBox.Show(err.Message,"Hata",MessageBoxButtons.OK,MessageBoxIcon.Error);
    }
}
```

Örneğin Kitap sınıfına ait nesne örneğini oluştururken Fiyat alanının değerini 175 olarak girelim.



Görüldüğü gibi oluşturduğumuz istisna sınıfına ait nesne örneği, Kitap nesnesi oluşturulmaya çalışıldığında ortama fırlatılmıştır. Bir de Fiyat değerini 10' ar birim azalttığımızda eksi değere geçtiğimiz bir metodumuz vardı. Kitap nesnesinin istisnasız oluşturup fiyatını negatif değere çektiğimizde uygulamanın `FiyatException` istisnası nedeni ile kesilerek sonlandırıldığını görürüz.

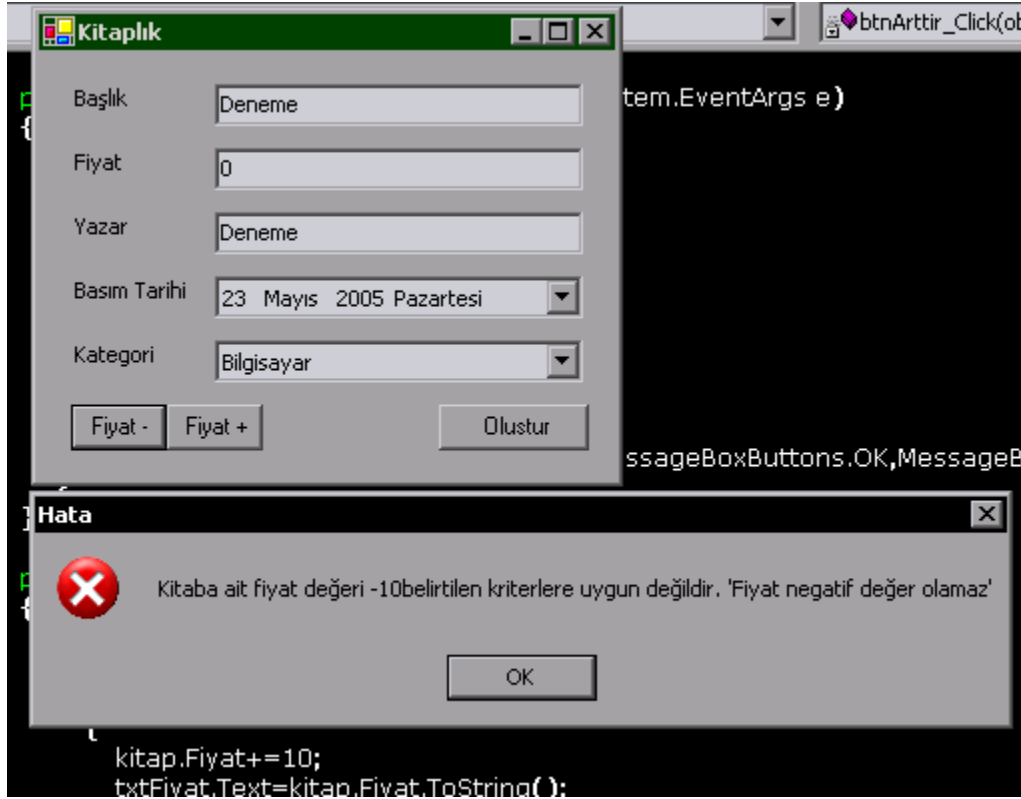


Uygulamanın istisnayı yakalayamamasının sebebi fiyat azaltımı yapan metodumuzun ilgili istisnayı yakalayacak bir try...catch yapısını kullanmayışdır. İster kendi istisna nesnelerimiz olsun ister sistemde var olan istisna nesneleri

olsun, bunların yakalanarak uygulamanın sonlandırılmadan yaşamaya devam edebilmesi için uygun catch blokları ile yakalanmaları şarttır. Dolayısıyla fiyat arttırma ve azaltma metodlarımızı aşağıdaki gibi yenilememiz gerekmektedir.

```
private void btnIndirim_Click(object sender, System.EventArgs e)
{
    try
    {
        if(kitap!=null)
        {
            kitap.Fiyat-=10;
            txtFiyat.Text=kitap.Fiyat.ToString();
        }
    }
    catch(FiyatException err)
    {
        MessageBox.Show(err.Message,"Hata",MessageBoxButtons.OK,MessageBoxIcon.Error);
    }
}

private void btnArttir_Click(object sender, System.EventArgs e)
{
    try
    {
        if(kitap!=null)
        {
            kitap.Fiyat+=10;
            txtFiyat.Text=kitap.Fiyat.ToString();
        }
    }
    catch(FiyatException err)
    {
        MessageBox.Show(err.Message,"Hata",MessageBoxButtons.OK,MessageBoxIcon.Error);
    }
}
```



Artık Fiyat alanı için eksi değeri oluşmamaktadır. Burada dikkat etmemiz gereken bir diğer noktada catch bloklarında kendi tanımladığımız istisna sınıflarını yakalayabilmek için ilerde o istisna sınıfına ait nesne örneğini belirtme zorunluluğumuzun olmayışdır. Yani

```
catch(FiyatException err)
{
    MessageBox.Show(err.Message,"Hata",MessageBoxButtons.OK,MessageBoxIcon.Error);
}
```

yerine,

```
catch(Exception err)
{
    MessageBox.Show(err.Message,"Hata",MessageBoxButtons.OK,MessageBoxIcon.Error);
}
```

formunuda kullanabiliriz. Yine aynı istisna mesajlarını yakalayacağız.

Görüldüğü gibi, uygulamalarımızda düşündüğümüz istisnai durumları yakalayabilmemizi sağlayacak sınıfları tasarlamak son derece kolaydır. Önemli olan nokta, istisna nesnemin gerçekten gerekli olup olmadığıdır. Eğer böyle bir gereklilik var ise ve kendi istisna sınıflarımızı oluşturduysak bunlara ait nesne örneklerini uygun yerlerde ortama fırlatmalıyız. Son olarak fırlattığımız istisna nesnelerimizi catch blokları ile yakalayarak uygulamaya yön vermeliyiz. Böylece

geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşmek dileğiyle hepinize mutlu günler dilerim.

[Örnek uygulama için tıklayın.](#)

Operator Overloading (Operatörlerin Aşırı Yüklenmesi) - 03 Haziran 2005

Cuma

C#, operator overloading, method overloading,

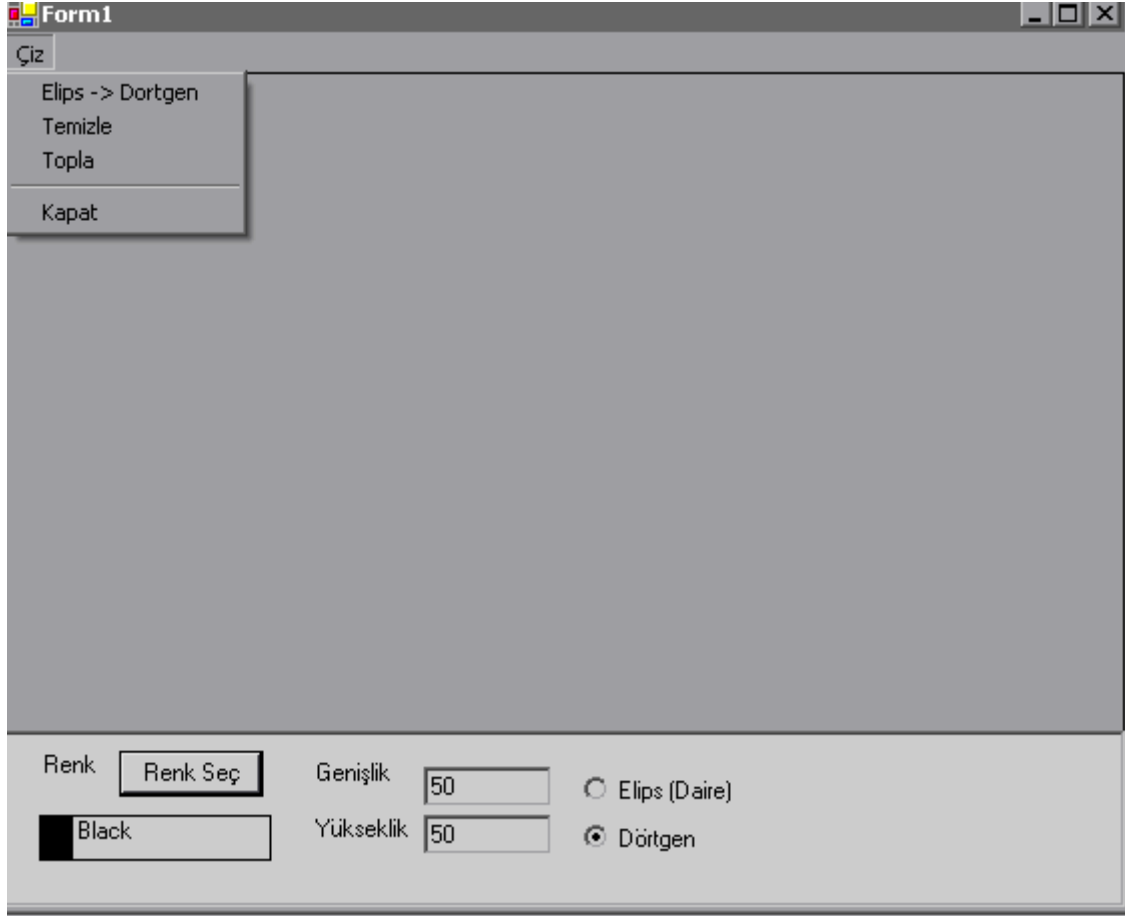
Değerli Okurlarım Merhabalar,

Hepimiz uygulamalarımızda sıklıkla operatörleri kullanmaktayız. Matematiksel işlemlerde, koşullu ifadelerde, tip dönüştürme işlemlerinde vb...Ancak onların kendi yazdığımız sınıflar için özel anlamlar ifade edecek şekilde yüklenmesi ile pek az uğraşmaktayız. Basit bir toplama operatörünün bile, yeri geldiğinde kendi sınıflarımıza ait nesne örnekleri üzerinde daha farklı davranışlar gösterecek şekilde yeniden yapılandırılması son derece önemlidir. Bu aynı zamanda dilin sağladığı esnekliği ve genişletilebilirliğini de gözler önüne sergilemektedir. İşte bu makalemizde, basit olarak operatörlerin aşırı yüklenmelerinin nasıl gerçekleştirilebileceğini örnek bir uygulama üzerinden incelemeye çalışacağız.

İlk olarak senaryomuzdan kısaca bahsedelim. Uygulamamızda System.Drawing isim alanını kullanarak dörtgen ve eliptik şekilleri çizmemizi sağlayacak iki adet sınıfımız olacak. Bu sınıfların çizim metodlarına baktığımızda ortak parametreler içerdiklerini görürüz. Bu nedenle bu ortak parametreleri bir arada toplayacağımız bir üst sınıfta işin içine katarak ilgili şekil sınıflarını buradan türeteceğiz.

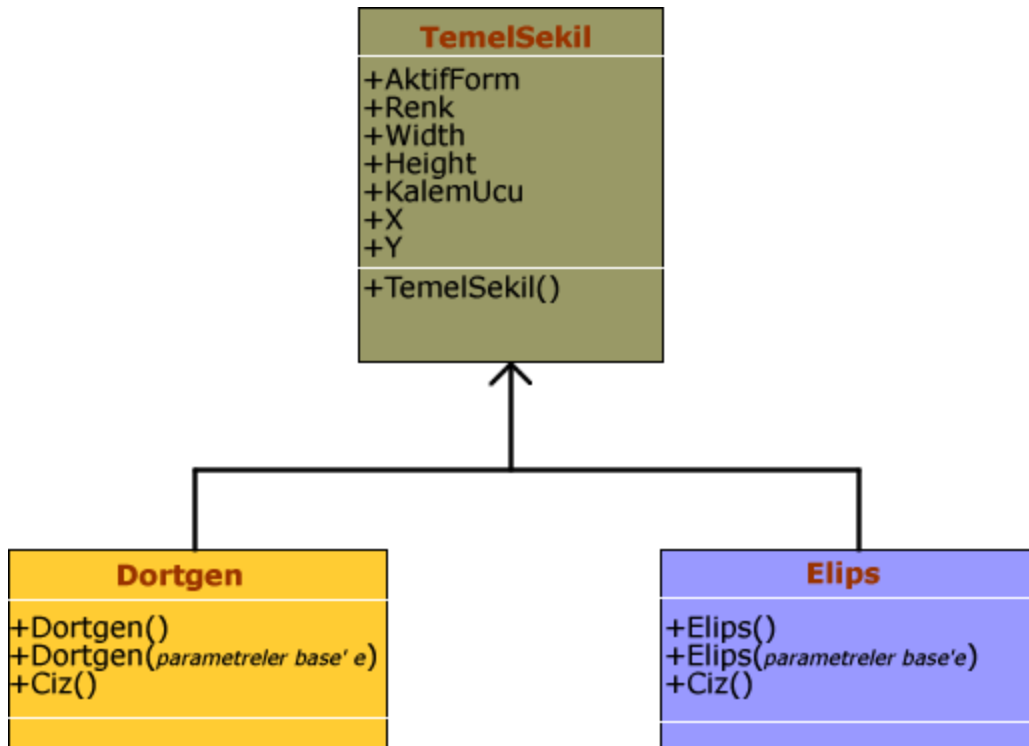
Amacımız kalıtım kavramı üzerinde durmak değil. Bunu sadece kod okunabilirliğini ve nesnelerinin kullanılabilirliğini kolaylaştırmak amacıyla gerçekleştiriyoruz. Peki bu sınıfların yer aldığı bir uygulamada hangi operatörleri ne amaçla aşırı yükleyebiliriz?

İlk başta akla gelen eliptik bir şeklin içerdği koordinat, boyut, renk gibi değerleri ile birlikte bir dörtgene çevrilmesi olabilir. Burada dörtgen tipinden bir nesne örneğinin, bilinçli (explicit) veya bilinçsiz (implicit) olarak eliptik bir nesne örneğine dönüştürülmesi söz konusudur. Bunun için cast operatörünü aşırı yükleyebiliriz. Diğer taraftan, var olan dörtgen veya eliptik nesnelerinin kendi aralarında toplama operatörleri ile toplanması sonucu çeşitli kriterlere uyum sağlayacak yeni bir dörtgen veya elips nesnesini elde etmeyi düşünebiliriz. Örneğin, iki kareyi toplayıp, yeni boyutları bu iki karenin toplamı kadar olan başka bir kare nesnesini çizdirebiliriz. Bu işlevsellikte ancak ve ancak toplama operatörünün burada söz konusu olan sınıflar için aşırı yüklenmesi ile mümkün olabilir. Şimdi gelin uygulamamızı geliştirmeye başlayalım. Operatörlerin aşırı yüklenmesini aşağıdaki görünüme sahip bir windows uygulamasında inceleyeceğiz.



Uygulamayı mümkün olduğu kadar basit tasarlamaya çalıştım. Amacımız operatörlerin aşırı yüklenmesini incelemek. Bu nedenle Macromedia Fireworks gibi bir grafik tasarım programını icat etmeye çalışmıyoruz. Programımız temel olarak belirli renkte çizgilere sahip olan dörtgensel ve eliptik şekilleri çiziyor. Bir şekli çizmek için genişlik ve yüksekliğini ilgili textBox kontrollerine atadıktan sonra mouse ile ekranın herhangi bir yerine tıklamanız yeterli olacaktır. Bununla birlikte işe biraz renk katmak amacı ile çizgi renklerini seçebiliyorsunuz. Menüde bizim asıl ilgilendiğimiz iki seçenek var. Bunlardan birisi bir Elips nesnesini, Dörtgen tipinden bir nesneye dönüştürerek ekrana çiziyor. Diğer menü seçeneği ilede iki Dörtgen nesnesini toplayıp sonucunu ekrana çizdiriyoruz. Makalenin ilerleyen safhalarında iki şeklin boyutsal bazda birbirlerine eşit olup olmadığını bildirecek şekilde koşul operatörlerini de aşırı yükleyeceğiz. Nesneleri tutmak amacıyla iki ArrayList koleksiyonu kullanmayı tercih ettim. Elbetteki siz bu programı dahada geliştirmeli ve nesnelerin daha esnek olarak tutulabileceği bir yapıyı kurgulamalısınız.

Gelelim uygulamamızdaki kritik sınıflara. Bu sınıflar, Dörtgen, Elips ve TemelSekil sınıflarıdır.



Dortgen ve Elips sınıfları TemelSekil sınıfından türetilmiştir. Sebebi, Dortgen ve Elips sınıflarının çizimi için kullanılan metodların aynı tipte ve sayıda parametre alıyor olmalarıdır. Dolayısıyla çizim için gerekli materyalleri bir üst sınıfta tutmak ve bunlara tek bir yerden erişebilmek amacıyla bu tarz bir yapı tercih edilmiştir. Sınıflarımıza ilişkin başlangıç kodları aşağıdaki gibidir.

Dörtgen.cs

```
using System;
using System.Drawing;
using System.Windows.Forms;
```

```
namespace UsingGDIWithOperatorOverloading
{
    public class Dortgen:TemelSekil
    {
        public Dortgen(Panel aktifForm,Color color,int penSize,int x,int y,int width,int height)
        {
            base.m_X=x;
            base.m_Y=y;
            base.m_Width=width;
            base.m_Height=height;
            base.m_Color=color;
            base.m_PenSize=penSize;
            base.m_AktifForm=aktifForm;
        }

        public Dortgen()
    }
}
```

```

    {
    }

    public void Ciz()
    {
        Graphics cizici=m_AktifForm.CreateGraphics();
        Pen kalem=new Pen(m_Color,m_PenSize);
        cizici.DrawRectangle(kalem,m_X,m_Y,m_Width,m_Height);
    }
}

```

Dortgen sınıfında şu an için sadece Ciz isimli bir metodumuz var. Constructor metodumuz aldığı parametreleri direkt olarak TemelSekil sınıfına göndermekte. Ciz metodu, Dortgen sınıfına ait nesne örneğini parametre olarak gelen alan üzerinde çizen işlevlere sahiptir. Dikkat ederseniz, dörtgenin çizileceği yer, çizgi kalınlığı, X ve Y koordinatları, çizgi rengi, şeklin genişliği ve yüksekliği gibi bilgiler parametrik olarak kullanılmaktadır. Elips sınıfında Dortgen sınıfına çok benzer bir yapıdadır.

Elips.cs

```

using System;
using System.Drawing;
using System.Windows.Forms;

namespace UsingGDIWithOperatorOverloading
{
    public class Elips:TemelSekil
    {
        public Elips(Panel aktifForm,Color color,int penSize,int x,int y,int width,int height)
        {
            base.m_X=x;
            base.m_Y=y;
            base.m_Width=width;
            base.m_Height=height;
            base.m_Color=color;
            base.m_PenSize=penSize;
            base.m_AktifForm=aktifForm;
        }

        public Elips()
        {

        }

        public void Ciz()
        {
            Graphics cizici=m_AktifForm.CreateGraphics();

```

```

        Pen kalem=new Pen(m_Color,m_PenSize);
        cizici.DrawEllipse(kalem,m_X,m_Y,m_Width,m_Height);
    }

    public static Elips operator+(Elips k1,Elips k2)
    {
        int R=(k1.m_Color.R+k2.m_Color.R)%255;
        int G=(k1.m_Color.B+k2.m_Color.B)%255;
        int B=(k1.m_Color.G+k2.m_Color.G)%255;
        Color c=Color.FromArgb(R,G,B);
        Elips elips=new Elips(k1.m_AktifForm,c,k1.m_PenSize+k2.m_PenSize,
        k1.m_X+k2.m_X,k1.m_Y+k2.m_Y,k1.m_Width+k2.m_Width,k1.m_Height+k2.m_Height);
        return elips;
    }
}

```

Son olarak TemelSekil.cs sınıfımız ise aşağıdaki gibidir.

```

using System;
using System.Drawing;
using System.Windows.Forms;

namespace UsingGDIWithOperatorOverloading
{
    public class TemelSekil
    {
        protected int m_X;
        protected int m_Y;
        protected int m_Width;
        protected int m_Height;
        protected Color m_Color;
        protected int m_PenSize;
        protected Panel m_AktifForm;

        public int X
        {
            get
            {
                return m_X;
            }
        }
        public int Y
        {
            get
            {
                return m_Y;
            }
        }
    }
}

```

```

    }
    public int Width
    {
        get
        {
            return m_Width;
        }
    }
    public int Height
    {
        get
        {
            return m_Height;
        }
    }
    public Color Renk
    {
        get
        {
            return m_Color;
        }
    }
    public Panel AktifForm
    {
        get
        {
            return m_AktifForm;
        }
    }

    public int KalemUcu
    {
        get
        {
            return m_PenSize;
        }
    }
    public TemelSekil()
    {
    }
}

```

Şimdi gelelim asıl sorunumuza; bir Dortgen nesne örneğini oluşturmak son derece basittir.

```

Dortgen dortgen=new
Dortgen(this.pnlKaraTahta,Renk,2,Baslangic_X,Baslangic_Y,Genislik,Yukseklık);

```

Hatta bu nesneyi ekrana çizdirmek artık çok daha kolaydır.

```
dortgen.Ciz();
```

Gel gelelim aşağıdaki kod satırlarının işletilmesi sonrasında nasıl bir sonuç alacağımı meçhuldür?

```
Dortgen dortgen1=new Dortgen(this.pnlKaraTahta,Color.Black,2,5,20,100,50);  
Dortgen dortgen2=new Dortgen(this.pnlKaraTahta,Color.Yellow,2,10,40,75,80);  
Dortgen d=dortgen1+dortgen2;  
d.Ciz();
```

Bu haliyle uygulamamızı derlediğimizde aşağıdaki hata mesajını alırız;



Operator '+' cannot be applied to operands of type 'UsingGDIWithOperatorOverloading.Dortgen' and 'UsingGDIWithOperatorOverloading.Dortgen'

Sebebi gayet açıktır. Dortgen sınıfı toplam işleminin nasıl yapılacağını bilemez. Bunu geliştirici olarak bizim ona öğretmemiz gerekmektedir. O halde gelin toplama işlemini bu sınıfa nasıl öğreteceğimize bakalım. Herşeyden önce operatörlerin aşırı yüklenmesi ile ilgili olarak bir takım kurallara vardır. Aslında bu kuralları bizde tahmin edebiliriz.



Kural 1; operatörler, operatör metodları yardımıyla aşırı yüklenirler. Bu sebepten bir metod gövdeleri, parametreleri ve dönüş değerleri vardır.

Kural 2; operatör metodları static olmalıdır. Bunun sebebi operatör işlevselliği için nesne örneğine ihtiyaç duyulmamasıdır.

Kural 3; operatör metodları operator anahtar kelimesini içermelidir. Örneğin: operator + gibi.

Kural 4; elbette her yerden erişilebilmeleri gerektiğinden public olmalıdır.

Kural 5; operatörler doğaları gereği en az bir operand ile çalışır. Dolayısıyla aşırı yükleyeceğimiz operator metodların en az bir parametre alması şarttır.

Bu kuralları dikkate aldığımızda Dortgen sınıfı için toplama operatörünü aşağıdaki haliyle aşırı yükleyebiliriz.

```
public static Dortgen operator+(Dortgen k1,Dortgen k2)
```

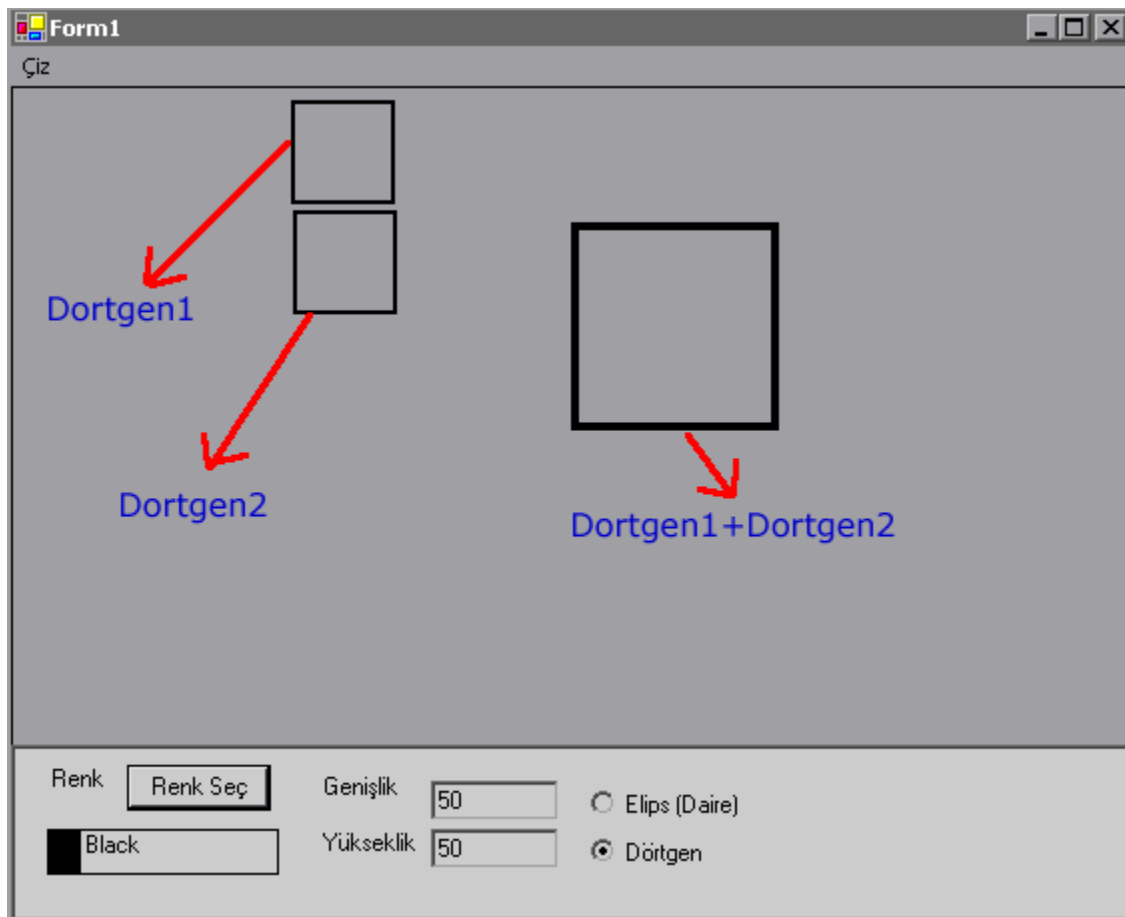


```

{
    int R=(k1.m_Color.R+k2.m_Color.R)%255;
    int G=(k1.m_Color.B+k2.m_Color.B)%255;
    int B=(k1.m_Color.G+k2.m_Color.G)%255;
    Color c=Color.FromArgb(R,G,B);
    Dortgen Dortgen=new
    Dortgen(k1.m_AktifForm,c,k1.m_PenSize+k2.m_PenSize,k1.m_X+k2.m_X,k1.m_Y+k2.m_Y,
    k1.m_Width+k2.m_Width,k1.m_Height+k2.m_Height);
    return Dortgen;
}

```

Dikkat ederseniz + operatörümüze ilişkin metodumuz, Dortgen tipinde iki nesne örneğini alıp bunlar üzerinde bir takım işlemler yaparak sonuç olarak ürettiği Dortgen tipinden nesneyi geriye döndürmektedir. Artık biraz önce hata veren kodlarımız şimdi çalışacaktır. Asıl uygulamamızı yürüttüğümüzde aşağıdakine benzer bir sonuç elde ederiz.



Toplama operatörüne yaptığımız yüklemeyi diğer operatörlere de yapabiliriz. Ancak aşırı yüklenecek operatörler arasında özel öneme sahip olanlar ve hatta aşırı yükleme yapılamıyacak olanlar da vardır. Sözelimi ekrandaki bir elips şeklini dörtgen tipine çevirmek istediğimizi varsayalım. Burada bilinçsiz olarak aşağıdaki gibi bir atama yapmak isteyebiliriz.

```
Dortgen d=elipsOrnegi;
```

Diğer yandan bilinçli olarakta aşağıdaki tarzda bir dönüşüm de yapmamız istenebilir.

Dortgen d=(Dortgen)elipsOrnegi;

Dikkat ederseniz ilk örnekte bilinçsiz, ikinci örnekte ise bilinçli tür dönüşümü söz konusudur. Burada mevzu bahis olan dönüştürme işlemlerini ilgili sınıfa öğretebilmek için yine operatör aşırı yüklemekten faydalanabiliriz. Örneğin aşağıdaki metod bilinçli olarak Dortgen tipine ait cast operatörünü aşırı yüklemektedir.

```
public static explicit operator Dortgen(Elips elips)
{
    TemelSekil ts=elips;
    Dortgen Dortgen=new
    Dortgen(ts.AktifForm,ts.Renk,ts.KalemUcu,ts.X,ts.Y,ts.Width,ts.Height);
    return Dortgen;
}
```

Metodumuzda dikkat çeken en önemli nokta explicit anahtar sözcüğüdür. Bu Dortgen anahtar sözcüğünün cast operatörü olarak kullanıldığı durumlarda blok içerisindeki kod satırlarının çalıştırılacağını ifade etmektedir. Aynı şekilde bilinçsiz tür dönüşümüne izin verecek operatör yüklemelerini de yapabiliriz. Tek yapmamız gereken implicit anahtar sözcüğünü kullanmaktır.

```
public static implicit operator Dortgen(Elips elips)
{
    TemelSekil ts=elips;
    Dortgen Dortgen=new
    Dortgen(ts.AktifForm,ts.Renk,ts.KalemUcu,ts.X,ts.Y,ts.Width,ts.Height);
    return Dortgen;
}
```

Elbette dönüştürme operatörlerinin aşırı yüklenmesi ile ilgili olarak dikkat etmemiz gereken önemli bir ayrıntı vardır.



Hem implicit hem de explicit operatörlerini aynı anda aşırı yükleyemeyiz.

Ancak, sadece implicit operatörünün yüklemesi ile, çalışma zamanında hem explicit hem de implicit dönüşümlere izin vermiş oluruz. Yani aşağıdaki iki kod satırında başarılı bir şekilde çalışacaktır.

```
Dortgen dortgen=elipsNesnesi1;
Dortgen dortgen2=(Dortgen)elipsNesnesi2;
```

Görüldüğü gibi operatörlerin aşırı yüklenmesi son derece kolay. Bunları kullandığımız windows uygulamasına ait kod satırları ise aşağıdaki gibidir.

```

private Color Renk;
private int Baslangic_X;
private int Baslangic_Y;
private int Genislik;
private int Yukseklik;
private ArrayList alDortgenler;
private ArrayList alElipsler;

private void menuElipsToDortgen_Click(object sender, System.EventArgs e)
{
    if(alElipsler.Count>0)
    {
        Elips el=(Elips)alElipsler[0];
        Dortgen dortgen=el;
        dortgen.Ciz();
    }
}

private void BaslangicAyarlari()
{
    lblSecilenRenk.Text=Renk.Name;
    lblRenk.BackColor=Renk;
}

private void btnRenkSec_Click(object sender, System.EventArgs e)
{
    colors.ShowDialog();
    Renk=colors.Color;
    BaslangicAyarlari();
}

private void Form1_Load(object sender, System.EventArgs e)
{
    Renk=Color.Black;
    alDortgenler=new ArrayList();
    alElipsler=new ArrayList();
    BaslangicAyarlari();
}

private void pnlKaraTahta_MouseDown(object sender,
System.Windows.Forms.MouseEventHandler e)
{
    Baslangic_X=e.X;
    Baslangic_Y=e.Y;
}

private void Ciz()
{

```

```

        if(rdbDortgen.Checked==true)
        {
            Dortgen dortgen=new
            Dortgen(this.pnlKaraTahta,Renk,2,Baslangic_X,Baslangic_Y,Genislik,Yukseklk);
            dortgen.Ciz();
            alDortgenler.Add(dortgen);
        }
        if(rdbElips.Checked==true)
        {
            Elips elips=new
            Elips(this.pnlKaraTahta,Renk,2,Baslangic_X,Baslangic_Y,Genislik,Yukseklk);
            elips.Ciz();
            alElipsler.Add(elips);
        }
    }

    private void menuTopla_Click(object sender, System.EventArgs e)
    {
        if(alDortgenler.Count>=2)
        {
            Dortgen d1=(Dortgen)alDortgenler[0];
            Dortgen d2=(Dortgen)alDortgenler[1];
            Dortgen d3=d1+d2;
            d3.Ciz();
        }
    }

    private void menuTemizle_Click(object sender, System.EventArgs e)
    {
        this.Refresh();
        Baslangic_X=0;
        Baslangic_Y=0;
        alDortgenler.Clear();
        alElipsler.Clear();
    }

    private void pnlKaraTahta_DoubleClick(object sender, System.EventArgs e)
    {
        Genislik=Convert.ToInt32(txtGenislik.Text);
        Yukseklik=Convert.ToInt32(txtYukseklk.Text);
        Ciz();
    }

    private void menuKapat_Click(object sender, System.EventArgs e)
    {
        Close();
    }

```

Dortgen sınıfı içerisinde aritmetik operatörlerin ve dönüştürme operatörlerinin nasıl yükleneceğini kısaca inceledik. Dilersek koşullu ifadelerde kullanılan operatörleride aşırı yükleyebiliriz. Örneğin == operatörünü yeniden yükleyerek dortgen sınıfımız için özel olarak kullanabiliriz.



== gibi karşılaştırma operatörlerinin aşırı yüklenmesinde tek şart, zıt operatörlerinde yüklenme zorunluluğunun olmasıdır. Örneğin, == için !=, < için > operatörünün aşırı yüklenmesi gibi...

Örneğimiz çok basit olduğundan genellikle koleksiyonlarda tuttuğumuz ilk iki nesne üzerinde işlem yapıyoruz. Yine bu tarz bir işlem yaptığımızı düşünelim ve iki Dortgen nesnesinin boyutlarının aynı olması halinde eşit olduklarını gösterelim. Bunun için Dortgen sınıfında == ve != operatörlerini aynı anda aşırı yüklemeliyiz. Aşağıdaki kodlarda Dortgen sınıfına bu işlevselliği nasıl kazandırdığımızı görebilirsiniz.

Dortgen.cs sınıfına eklenen operator metodlarımız;

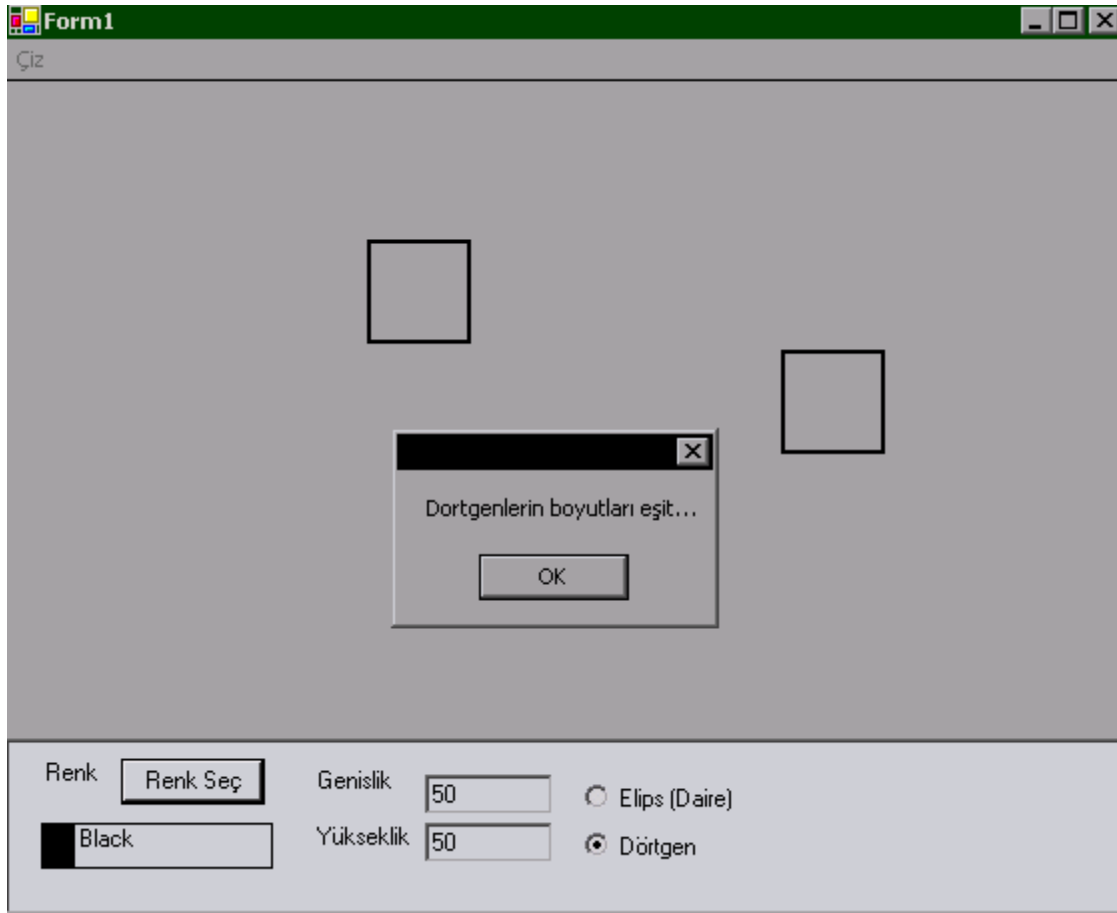
```
public static bool operator==(Dortgen d1,Dortgen d2)
{
    if((d1.Width==d2.Width)&&(d1.Height==d2.Height))
        return true;
    else
        return false;
}
```

```
public static bool operator!=(Dortgen d1,Dortgen d2)
{
    return !(d1==d2);
}
```

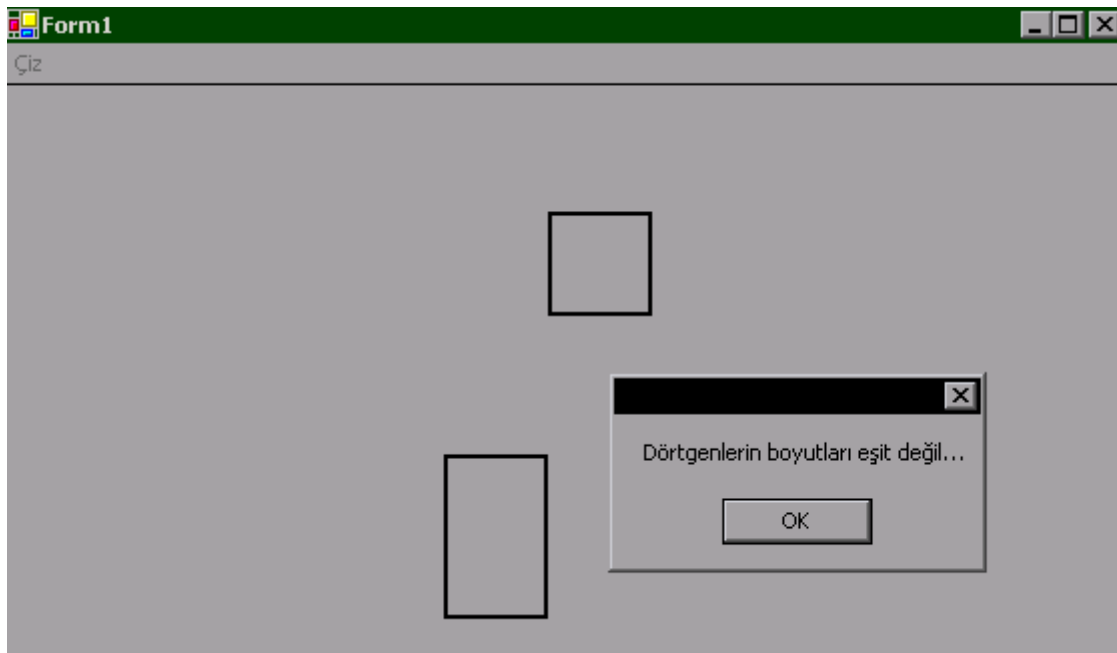
Operatörlerin uygulama içerisinde kullanımı;

```
private void menuEsitmi_Click(object sender, System.EventArgs e)
{
    Dortgen d1=(Dortgen)alDortgenler[0];
    Dortgen d2=(Dortgen)alDortgenler[1];
    if(d1==d2)
    {
        MessageBox.Show("Dortgenlerin boyutları eşit...");
    }
    if(d1!=d2)
    {
        MessageBox.Show("Dörtgenlerin boyutları eşit değil...");
    }
}
```

Eşitlik kontrolünün sonucu;



Eşit değildir kontrolünün sonucu;



Elbetteki aşırı yükleme yapamayacağımız operatörlerde vardır.



=, ., ?:, ->, new, is, as, sizeof, &&, ||, () operatörlerini aşırı yüklememiz yasaklanmıştır.

Görüldüğü gibi nesnelerimiz için, C# dilinde var olan operatörleri aşırı yüklemek son derece kolaydır. Dikkat etmemiz gereken bir takım kurallar vardır ki bunlar zamanla öğrenilebilir. Operatörlerin özellikle aşırı yüklenmesine ihtiyaç duyulacağı durumları göz önüne aldığımızda, grafik ve matematik uygulamalarının üst sıralarda yer aldığını görürüz. Örneğin sevgili Sefer ALGAN, Her Yönüyle C# Kitabında operatörlerin aşırı yüklenmesi ile ilgili olaraktan Kompleks sayıları incelemiştir. Özetle operatörleri aşırı yüklemek özellikle kendi oluşturduğumuz nesnelerin esnekliği açısından önemlidir. Bu makalemizde işlediğimiz [örnekte](#) bahsedilen aşırı yükleme işlemleri sadece Dortgen sınıfı için yapılmıştır. Size tavsiyem Elips sınıfı içinde benzer yüklemeleri yapmaya çalışmanızdır.

C# 2.0 ve Anonymous (İsimsiz) Metodlar

- 16 Haziran 2005 Perşembe

C# 2,

Değerli Okurlarım Merhabalar,

İsimsiz metodlar bildiğiniz gibi C# 2.0' a eklenmiş olan yeni özelliklerden birisidir. Temeli C# dilinin temsilci tipine dayanan bu yeni teknikte amaç, temsilcileri işaret edecekleri metodların sahip oldukları kod blokları ile bir seferde tanımlayabilmektir. İsimsiz metodları anlayabilmek için herşeyden önce temsilcilerin (delegates) iyi kavranmış olması gerekmektedir(*Ön bilgi veya hatırlatma açısından Örnek [makale](#) ve [video](#) larımızı incelemenizi öneririm*)

Kısaca temsilciler, çalışma zamanında metodların başlangıç adreslerini işaret eden tip (type) lerdir. Temsilcilerin herhangi bir metodu çalışma zamanında işaret edebilmesinin yanı sıra bu metodu(metodları) çağırabilmesi ve hatta parametreler göndererek dönüş değerleri vermesi gibi yetenekleride vardır. Ama tüm bu özellikleri arasında en önemlisi, çalışma zamanında hangi metodu çalıştıracığına karar vermesidir.



Temsilciler (delegates), multithreading(çok kanallı programlama) modelinde, event-driven (olay güdümlü) programlamada, Callback Modeli ile Asenkron erişim tekniklerinde etkin olarak kullanılmaktadır.

C# 2.0 ile gelen isimsiz (anonymous) metodları anlamak için öncelikle C# dilinde bir temsilciyi nasıl kullandığımıza bakmamızda fayda var. Aşağıdaki örnekte basit olarak bir temsilci tanımlanmış ve kullanılmıştır. Bu temsilci iki adet double tipte parametre alan ve geriye double tipinden değerler döndüren metodları işaret edebilecek şekilde tanımlanmıştır. Dikkat ederseniz temsilci nesnemize ait nesne örneğimiz oluşturulurken işaret edeceği metod parametre olarak verilmektedir. Daha sonra bu temsilci nesne örneği üzerinden işaret edilen metod çağırılmaktadır.

```
using System;
```

```
namespace DefiningDelegate
```

```
{
```

```
    //Temsilcimiz iki adet double tipinden parametre alan ve geriye double tipinden değer döndüren metodları işaret edebilecek.
```

```
    public delegate double Temsilci(double pi,double r);
```

```
    class Class1
```



```

{
    [STAThread]
    static void Main(string[] args)
    {
        // Temsilci nesnemiz oluşturuluyor ve Alan isimli metodu işaret edeceği söyleniyor.
        Temsilci t=new Temsilci(Alan);
        // Temsilcimizin çalışma zamanında işaret ettiği metod çağırılıyor.
        double daire_Alan=t(3.14,10);
        Console.WriteLine(daire_Alan);
    }

    // Tanımladığımız temsilci tarafından işaret edilebilecek formatta bir metod bildirimi.
    static double Alan(double pi_Degeri,double yarıcap)
    {
        return pi_Degeri*(yarıcap*yarıcap);
    }
}

```

C# 2.0 için anonymous metodları kullanarak yukarıdaki uygulamayı aşağıdaki kod parçasında görüldüğü gibi yazabiliriz. Yeni versiyonda temsilci kullanımındaki tek fark temsilcinin işaret edeceği metod bloğunun, çalışma zamanında bu metodu çağırarak olan temsilci nesnesine eklenmiş oluşudur. Buradan temsilcilerin inline (satır içi) kodlama yeteneği kazanmış olduklarını söyleyebiliriz.



İsimsiz(Anonymous) metodlar dışarıdan parametre alabilirler ve geriye değer döndürebilirler.

C# 2.0 versiyonu

```

using System;
using System.Collections.Generic;
using System.Text;

namespace UsingAnonymousMethods
{
    // Temsilcimizi tanımlıyoruz.
    public delegate double Temsilci(double a,double b);

    class Program
    {
        static void Main(string[] args)
        {
            // Temsilcimizi hem oluşturuyor hemde işaret edeceği metod bloğunu anonymous
            olarak tanımlıyoruz.

```

```

        Temsilci t = delegate(double pi,double r)
        {
            return pi * r*r;
        };
        // Temsilcimizi parametreler ile birlikte çağırıyoruz ve dönüş değerini double tipinden
        bir değişkene atıyoruz.
        double alan = t(3.14, 10);
        Console.WriteLine(alan);
    }
}

```

Gelelim isimsiz metodların bir diğer kullanım şekline. Çok kanallı (Multithreading) programlama modelinde bildiğiniz gibi ThreadStart isimli bir temsilci(delegate) tipi kullanılmaktadır. Bu temsilci bir Thread nesne örneğinin oluşturulması sırasında, yapıcı metod için parametre olarak kullanılmaktadır. C# 1.1 versiyonunda basit bir thread modeli aşağıdaki kod parçasında olduğu gibi örneklenebilir. Lütfen temsilcilerin çalışma zamanında işaret edecekleri metodlar ile birlikte nasıl ilişkilendirildiğine dikkat edin.

```

using System;
using System.Threading;

namespace UsingThreading1
{
    class Class1
    {
        //ThreadStart temsilcimiz çalışma zamanında ilgili process içindeki metodu temsil
        edecek.
        static ThreadStart threadStart1;
        static ThreadStart threadStart2;

        // Thread nesnemiz parametre olarak aldığı ThreadStart temsilcisinin işaret ettiğim
        metodun ait olduğu process için gerekli işlemleri (start,abort,resume vb...) gerçekleştirecek.
        static Thread thread1;
        static Thread thread2;

        // Thread içinde çalışacak metodlarımız.
        static void Say1()
        {
            for(int i=0;i<100;i++)
            {
                Console.Write(i);
                Thread.Sleep(100);
            }
        }

        static void Say2()
        {

```

```

        for(int i=0;i<100;i++)
        {
            Console.WriteLine(i);
            Thread.Sleep(150);
        }
    }

    [STAThread]
    static void Main(string[] args)
    {
        // ThreadStart temsilcilerimiz çalışma zamanında işaret edecekleri parametre olarak
        alacak şekilde tanımlanıyor.
        threadStart1=new ThreadStart(Say1);
        threadStart2=new ThreadStart(Say2);
        // Thread nesnelerimiz ThreadStart temsilcilerinin parametre olarak alacak şekilde
        tanımlanıyor.
        thread1=new Thread(threadStart1);
        thread2=new Thread(threadStart2);
        // Threadler çalıştırılmaya başlanıyor.
        thread1.Start();
        thread2.Start();
    }
}

```

Şimdi aynı örneğin C# 2.0' da isimsiz metodlar yardımıyla nasıl yazılabileceğine bakalım. Burada dikkat ederseniz ThreadStart temsilcisi görülmemektedir. Bu elbetteki ThreadStart temsilcisinin kullanılmadığı anlamına gelmemelidir. Aslında bu tanımlama gizli olarak Thread sınıfına ait nesne örnekleri oluşturulurken yapılmaktadır. Yani çalışma zamanında Thread nesnesinin zaten bir ThreadStart temsilci nesnesine ihtiyacı olduğu çalışma ortamı tarafından bilinmektedir. Diğer yandan kod yazımı açısından bakıldığında iki işlemin, (ThreadStart' ın işaret edeceği metodu gösterecek şekilde örneklenmesi ve daha sonrada Thread nesnesinin oluşturulması için parametre olarak kullanılması) tek seferde yapılmaktadır.

C# 2.0 Versiyonu

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace UsingAnonymousMethods
{
    class Program
    {
        // Thread içerisinde çalışacak metodumuz.
        static void Say1()
    }
}

```

```

    {
        for(int i=0;i<100;i++)
        {
            Console.Write(i);
            Thread.Sleep(100);
        }
    }

    static void Main(string[] args)
    {
        // İlk Thread nesnemizi örneklerken ThreadStart temsilcisinin işaret edeceği metodu
        burada anonymous metod olarak tanımlıyoruz.
        Thread thread1=new Thread(delegate(){
            Say1();
        });
        // Tanımladığımız thread' i çalıştırıyoruz.
        thread1.Start();
        // İkinci Thread nesne örneğimizi oluşturuyoruz. Bu sefer thread1 nesnesinden farklı
        olarak anonymous metodumuz içerisine direkt kodları gömdük. thread1 nesnesinde ise
        kodları içeren metodu, anonymous metod bloğumuz içine gömmüştük.
        Thread thread2 = new Thread(delegate(){
            for (int i = 0; i < 100; i++)
            {
                Console.WriteLine(i);
                Thread.Sleep(150);
            }
        });
        thread2.Start();
    }
}

```

Temsilcilerin kullanıldığı diğer bir tekniğinde olay güdümlü (event-driven) programlama modeli olduğunu söylemiştik. Bu modelde de Threading modeline benzer bir yapı vardır. Her event (olay) tanımlanırken bir temsilci kullanılır. Bu temsilci olay meydana geldiğinde çalıştırılacak olan metodu çalışma zamanında işaret etmek ve çağırmakla yükümlüdür. Örneğin aşağıdaki kod parçasında bir windows formu üzerinde yer alan button kontrolünün Click olayına ilişkin kodlar yer almaktadır.

Bu örneğimizde, dikkat ederseniz button kontrolümüze Click event' ını yükleyebilmek için System.EventHandler temsilci tipi kullanılmaktadır. Bu temsilci, diğer olay metodlarında kullanılan temsilciler gibi sistemde önceden tanımlanmış halde yer almaktadır. InitializeComponent metodunda button kontrolümüze click olayı yüklenirken, hangi temsilcinin çalışma zamanının hangi metodu işaret edeceği ve çalıştıracağıda bildirilir. Sonrasında ise bu olay metodunun tanımlanması gerekmektedir.

```

public class Form1 : System.Windows.Forms.Form
{

```

```
private System.Windows.Forms.Button btnAksiyon;
```

```
// Diğer kodlar
```

```
private void InitializeComponent()  
{
```

```
// Diğer Kodlar
```

```
this.btnAksiyon.Click += new System.EventHandler(this.btnAksiyon_Click);  
}
```

```
// Diğer Kodlar
```

```
private void btnAksiyon_Click(object sender, System.EventArgs e)  
{  
    // Bir takım kodlar.  
}  
}
```

Şimdi aynı örneğin C# 2.0' da isimsiz metodlar yardımıyla nasıl yazılabileceğine bakalım. Görüldüğü gibi System.EventHandler temsilcisi burada görülmemektedir. Aslında tüm isimsiz metod modellerinde, temsilcilerden hangisi kullanılırsa kullanılsın (örneğin System.EventHandler veya ThreadStart gibi) bizim tek kullandığımız delegate anahtar sözcüğü ile bir metod bloğunun kombinasyonudur. Bu isimsiz metodların kullanımının bir faydası olarakta görülebilir.



İsimsiz(Anonymous) metodlarda, kullanılan temsilcinin bilinmesine gerek yoktur. delegate anahtar sözcüğü bu işi üstlenir.

C# 2.0 Versiyonu

```
private void InitializeComponent()  
{
```

```
// Diğer kodlar
```

```
this.btnOnay.Click += delegate(object sender, System.EventArgs arg)  
{  
    System.Windows.Forms.MessageBox.Show("Onay");  
};
```

```
// Diğer kodlar
```

```
}
```

İsimsiz metodlar uygulamaları açısından biraz karışık görülebilir. En azından alışınca kadar. Ancak kavrandıklarında çok faydalı olduklarını söyleyebiliriz. Nitekim temsilci nesnelerimizin tanımlaması gereken yerlerde direkt olarak kod bloklarını kullanmak oldukça kullanışlı bir teknik olarak karşımıza çıkmaktadır. Bu makalemizde C# 2.0 ile birlikte gelen isimsiz metod kavarmını, her zaman için içiçe olduğu temsilciler ile birlikte incelemeye çalıştık. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

C# 2.0 ve Static Sınıflar - 20 Haziran 2005

Pazartesi

C# 2, static, static class,

Değerli Okurlarım Merhabalar,

Çoğu zaman uygulamalarımızda, nesne örneğinin oluşturulmasına gerek duymayacağımız üyeleri kullanmak isteriz. Bu amaçla static üyeleri kullanırız. Şimdi bir de sadece static üyelerden oluşacak bir sınıf tasarlamak istediğimizi düşünelim. C# programlama dilinin ilk versiyonunda bu tip bir sınıfı yazmak için dikkat etmemiz gereken bir takım noktalar vardır. Static üyeler kullanılabilmeleri için tanımlı oldukları sınıfın nesne örneğine ihtiyaç duymazlar. Bu sebepten sadece static üyeler içerecek olan bir sınıfın örneklendirilememesi tercih edilecektir. Örneğin aşağıdaki kod parçasını dikkate alalım. TemelAritmetik isimli sınıfımız Toplam isimli static bir metod içermektedir.

```
using System;
```

```
namespace UsingStaticClasses
```

```
{  
    public class TemelAritmetik  
    {  
        public static double Toplam(double deger1,double deger2)  
        {  
            return deger1+deger2;  
        }  
    }  
}
```

```
class Class1
```

```
{  
    static void Main(string[] args)  
    {  
        double toplamSonuc=TemelAritmetik.Toplam(10,15);  
        Console.WriteLine(toplamSonuc);  
    }  
}
```

Bu kod parçasında aşağıda olduğu gibi TemelAritmetik sınıfına ait bir nesne örneği tanımlayıp Toplam metodunu bu nesne örneği üzerinden çağırmaya çalışmak derleme zamanında hataya neden olacaktır. Az öncede belirttiğimiz gibi, static üyelere tanımlandıkları sınıfa ait nesne örnekleri üzerinden erişilemezler.

```
TemelAritmetik temelAritmetik=new TemelAritmetik();  
double toplamSonuc=temelAritmetik.Toplam(4,5);
```

Ancak yine de dikkat ederseniz TemelAritmetik sınıfına ait nesne örneğini oluşturabilmekteyiz. Bu durumda ,ilk olarak static üyelerle dolu bir sınıfın kesin olarak örneklendirilmesini önlemek isteyeceğizdir. Bu nedenle varsayılan yapıcı metodu (default constructor) private olarak tanımlayarak bu durumun önüne geçebiliriz.

```
public class TemelAritmetik
{
    // Varsayılan yapıcı private olduğundan sınıfa ait nesne örneği oluşturulamayacaktır.
    private TemelAritmetik()
    {
    }

    public static double Toplam(double deger1,double deger2)
    {
        return deger1+deger2;
    }
}
```

Görüldüğü gibi artık TemelAritmetik sınıfına ait nesne örneklerinin üretilmesinin önüne geçtik. Aynı şekilde varsayılan yapıcı metodun private olarak tanımlanması bu sınıfta türetilme yapılmasında engellemektedir. Ancak bazen static üyeler içeren sınıfımızın yapıcı metodu public veya protected olarak tanımlanmış olabilir. Bu durumda türetme işlemi gerçekleştirilecektir. Örneğin aşağıdaki kod parçasını ele alalım.

```
using System;
```

```
namespace UsingStaticClasses
{
    public class TemelAritmetik
    {
        public TemelAritmetik()
        {
        }

        public static double Toplam(double deger1,double deger2)
        {
            return deger1+deger2;
        }
    }

    public class AltAritmetik:TemelAritmetik
    {
        public void AltIslemler()
        {
        }
    }

    class Class1
```



```

{
    [STAThread]
    static void Main(string[] args)
    {
        AltAritmetik altAritmetik=new AltAritmetik();
        altAritmetik.AltIslemler();
        double sonuc=AltAritmetik.Toplam(1,2);
    }
}

```

Dolayısıyla sadece static üyeler içerecek bir sınıfın türetme işlemi için kullanılmasında bir şekilde önlemek isteyebiliriz. Bu durumda sealed anahtar sözcüğü yardımıyla, ilgili sınıfın türetme amacıyla kullanılamayacağını belirtiriz.

```
public sealed class TemelAritmetik
```

Bu sadece static üyeler içeren, türetilmeye izin vermeyen ve nesne örneğini kesin olarak oluşturmak istemediğimiz bir sınıf için acaba yeterli midir? sealed olarak tanımlanmış sınıflar her ne kadar türetme amacıyla kullanılamasada, public erişim belirleyicisine sahip yapıcı metodları nedeni ile örnekendirilebilirler. Bize biraz daha zorlayıcı, kesin kurallar sunacak yeni bir yapı C# 2.0 ile birlikte gelmektedir. C# 2.0 sınıfların static olarak tanımlanabilmesine izin vermektedir. Static sınıflar beraberinde bir takım kurallarda getirir. Aşağıdaki kod parçası C# 2.0 için örnek bir static sınıf tanımlamasını içermektedir.

```

using System;
using System.Collections.Generic;
using System.Text;

namespace UsingStaticClasses
{
    public static class TemelAritmetik
    {
        public static double Toplam(double deger1,double deger2)
        {
            return deger1+deger2;
        }
        public static double pi = 3.14;
        public static double PI
        {
            get
            {
                return pi;
            }
            set
            {
                pi = value;
            }
        }
    }
}

```

```

    }
}

class Program
{
    static void Main(string[] args)
    {
        double sonuc=TemelAritmetik.Toplam(3, 5);
        Console.WriteLine(sonuc);
        Console.WriteLine(TemelAritmetik.pi);
        TemelAritmetik.PI = 3;
        Console.WriteLine(TemelAritmetik.PI);
    }
}
}

```

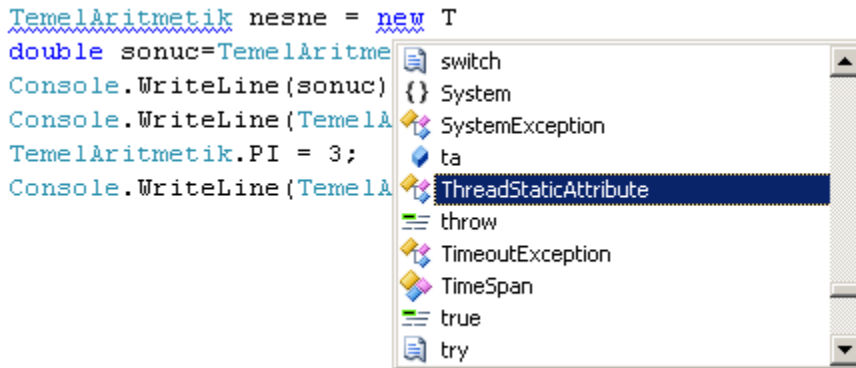
Örneğimizde içeriğinin static olarak tanımlanmış metod, field ve özellik bulunan bir sınıf görmekteyiz. Bu sınıfın bir önceki versiyona göre en önemli özelliği static olarak tanımlanabiliyor oluşudur. Peki bu sınıfın static olarak tanımlanmasını getirdiği kısıtlamalar nelerdir? İlk olarak bu sınıfa ait bir nesne örneği oluşturmaya çalışalım.

```

TemelAritmetik ta;
TemelAritmetik tAritmetik = new TemelAritmetik();

```

Bu tip bir kod yazımına Visual Studio.2005 zaten intellisense özelliği yardımıyla izin vermeyecektir. Lakin tAritmetik isimli nesne örneğini new operatörü ile oluşturmaya çalıştığımızda TemelAritmetik sınıfının adının listeye gelmediğini görürüz.



Diğer yandan kodu derlediğimizde aşağıdaki hataları alırız. Toplam 3 hatamız vardır. İlk satır için verilen hatadan static olarak tanımlanmış bir sınıfa ait nesne tanımlaması yapamadığımızı anlayabiliriz. Diğer yandan, ikinci satırda static sınıfa ait nesne örneğinin oluşturulamayacağı ve aynı zamanda tanımlanamayacağına dair hata mesajlarını alırız.

```

class Program
{
    static void Main(string[] args)
    {
        TemelAritmetik ta;
        TemelAritmetik nesne = new TemelAritmetik();
        double sonuc=TemelAritmetik.Toplam(3, 5);
        Console.WriteLine(sonuc);
    }
}


```

Error List

3 Errors 0 Warnings 0 Messages

	Description	File
1	Cannot declare variable of static type 'UsingStaticClasses.TemelAritmetik'	Program.cs
2	Cannot declare variable of static type 'UsingStaticClasses.TemelAritmetik'	Program.cs
3	Cannot create an instance of the static class 'UsingStaticClasses.TemelAritmetik'	Program.cs

Buradan şu sonuçlara varabiliriz;



Static olarak tanımlanmış sınıflara ait nesne örneklerini üretemeyiz. Ayrıca static sınıflara ait nesne tanımlamalarını da yapamayız.

Static sınıfların sınırlamaları sadece bunlarla sınırlı değildir. Şimdi yukarıdaki static sınıfımıza aşağıda olduğu gibi static olmayan bir kaç yeni üye daha ekleyelim.

```

public static class TemelAritmetik
{
    public TemelAritmetik()
    {
    }
    private int e;
    protected void BilgiVer()
    {
        // Bir takım kodlar.
    }

    public static double Toplam(double deger1,double deger2)
    {
        return deger1+deger2;
    }
    public static double pi = 3.14;
    public static double PI
    {
        get
        {

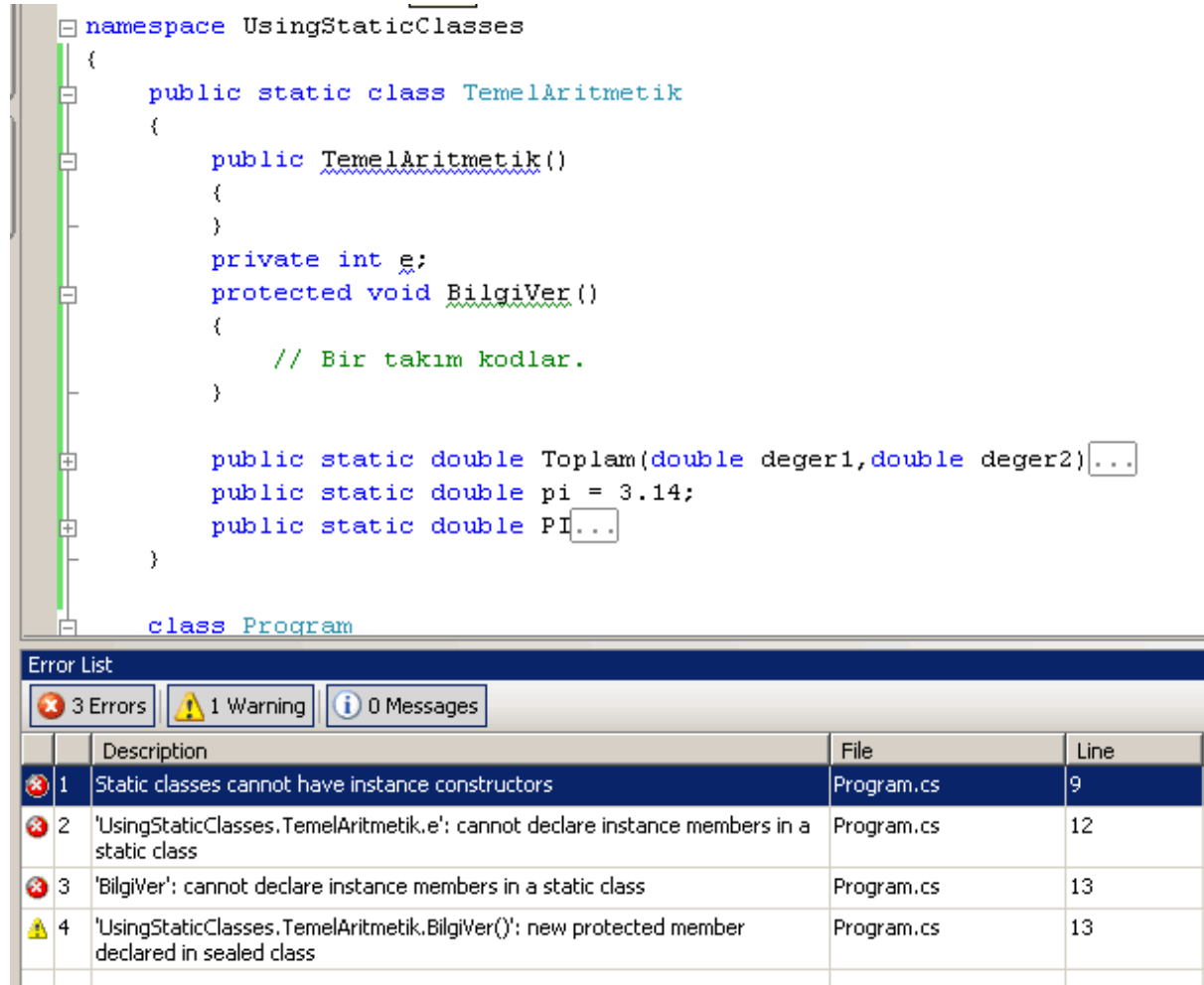
```

```

        return pi;
    }
    set
    {
        pi = value;
    }
}

```

Kaynak kodumuzu derlediğimizde 4 adet derleme zamanı hatası alırız.



```

namespace UsingStaticClasses
{
    public static class TemelAritmetik
    {
        public TemelAritmetik()
        {
        }
        private int e;
        protected void BilgiVer()
        {
            // Bir takım kodlar.
        }

        public static double Toplam(double deger1, double deger2)
        public static double pi = 3.14;
        public static double PI
    }
}

class Program

```

Error List			
3 Errors 1 Warning 0 Messages			
	Description	File	Line
1	Static classes cannot have instance constructors	Program.cs	9
2	'UsingStaticClasses.TemelAritmetik.e': cannot declare instance members in a static class	Program.cs	12
3	'BilgiVer()': cannot declare instance members in a static class	Program.cs	13
4	'UsingStaticClasses.TemelAritmetik.BilgiVer()': new protected member declared in sealed class	Program.cs	13

İlk hatamız static sınıf içerisinde varsayılan yapıcı(default constructor) metod tanımlamaya çalışmaktır. Dilerseniz varsayılan yapıcı metodu aşırı yükleyerek başka versiyonları ile de aynı uygulamayı derlemeyi deneyebilirsiniz. Sonuç hep aynı olacaktır. Elbette doğal olarak yapıcı metodları static olarak tanımlamaya çalışabiliriz ki bu zaten izin verilmeyen bir durumdur.



Static olarak tanımlanmış sınıflar yapıcı metodları (varsayılan yapıcı ve aşırı yüklenmiş versiyonları) içeremez.

Diğer hatalarımız ise, static sınıf içerisinde static olmayan üyeler tanımlamaya çalışmamızdır. Buradan da şu sonuca varabiliriz.



Static olarak tanımlanmış sınıflar sadece static üyeler içerebilir.

İncelememiz gereken bir diğer durum static sınıfların türetilip türetilmeyeceğidir. Aşağıdaki kod parçasını göz önüne alalım.

```
public static class TemelAritmetik
{
    // Static sınıf kodlarımız
}
```

```
public class AltAritmetik : TemelAritmetik
{
}
```

Visual Studio.2005 her zamanki gibi bu tarz bir yazıma zaten izin vermeyecektir. Ancak elimizin altında visual studio gibi bir geliştirme ortamı olmadığını ve notepad gibi bir editor yardımıyla bu kodu yazdığımızı düşünecek olursak aşağıdaki hata mesajını alırız.

```
namespace UsingStaticClasses
{
    public static class TemelAritmetik
    {
        public static double Toplam(double deger1, double deger2) ...
        public static double pi = 3.14;
        public static double PI ...
    }

    public class AltAritmetik : TemelAritmetik
    {
    }


    class Program
    {
        static void Main(string[] args)
        {
            double sonuc=TemelAritmetik.Toplam(3, 5);
            Console.WriteLine(sonuc);
        }
    }
}
```

Error List

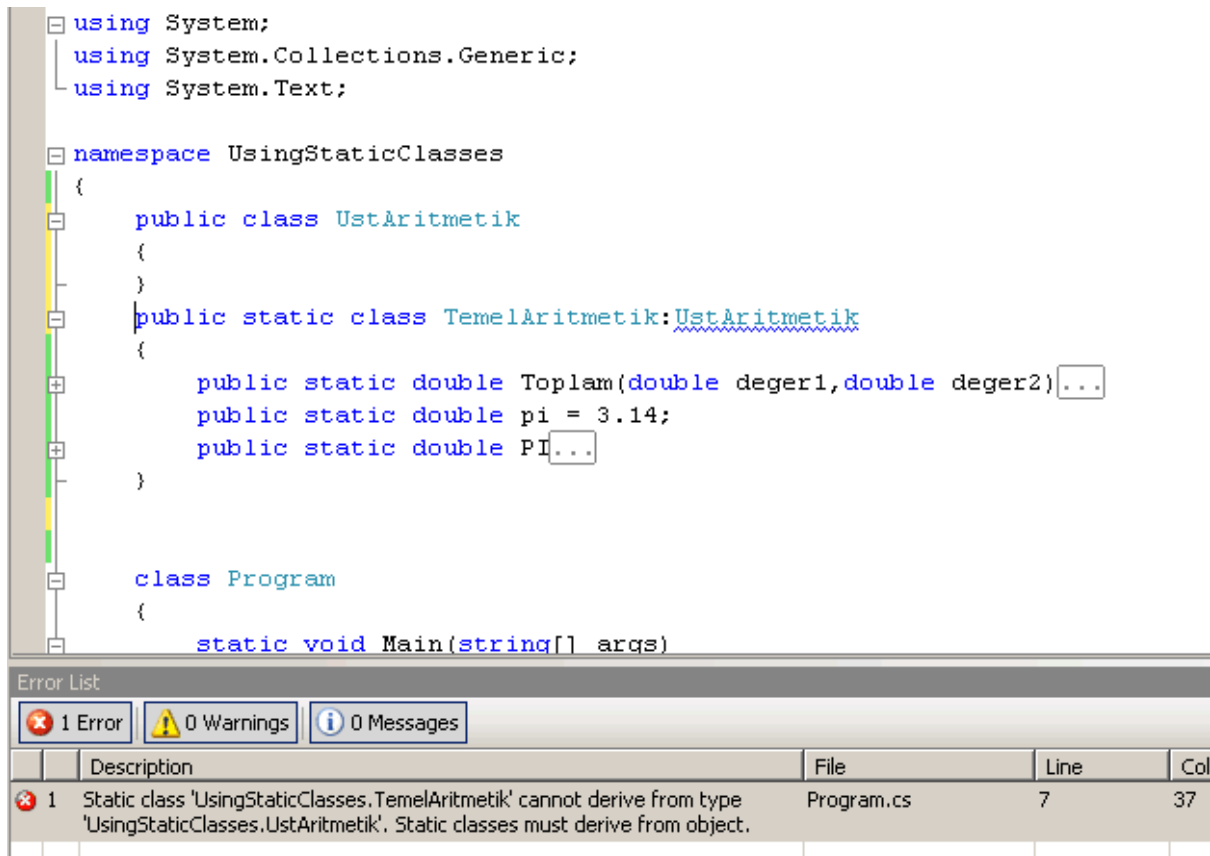
1 Error 0 Warnings 0 Messages

	Description	File	Line	Column
1	'UsingStaticClasses.AltAritmetik': Cannot derive from static class 'UsingStaticClasses.TemelAritmetik'	Program.cs	27	15

Buradan varacağımız sonuç ise,

	<i>Static olarak tanımlanmış sınıflardan <u>başka sınıflar</u> türetilemez.</i>
---	--

Burada dikkate değer başka bir durum daha vardır. Static bir sınıftan başka bir sınıfı türetemeyiz. Peki static sınıfı başka bir sınıftan türetebilir miyiz? Cevap basit;



Ancak bu durum static sınıflarında mutlaka object sınıfından türediği gerçeğini değiştirmez. Her ne kadar static bir sınıfı başka bir sınıftan türetemesekte onun object sınıfından türediği kesindir. Hata mesajımızda zaten bu durumu bize ispatlamaktadır. Bu noktada aklımıza kurnazca bir teknik gelebilir. Static bir sınıfı başka bir static sınıftan türetmeyi deneyebiliriz. Bu tarz bir kodu denediğimizde aşağıdaki hata mesajlarını alırız.

```
using System.Collections.Generic;
using System.Text;

namespace UsingStaticClasses
{
    public static class UstAritmetik
    {
        static double AlanHesabi(double p, double r)
        {
            return p * r * r;
        }
    }
    public static class TemelAritmetik:UstAritmetik
    {
        public static double Toplam(double deger1,double deger2)...
        public static double pi = 3.14;
        public static double PI...
    }
}
```

Error List

2 Errors 0 Warnings 0 Messages

	Description	File	Line
1	'UsingStaticClasses.TemelAritmetik': Cannot derive from static class 'UsingStaticClasses.UstAritmetik'	Program.cs	14
2	Static class 'UsingStaticClasses.TemelAritmetik' cannot derive from type 'UsingStaticClasses.UstAritmetik'. Static classes must derive from object.	Program.cs	14

Buna göre şu sonuca varabiliriz;



Static olarak tanımlanmış sınıflardan başka static sınıflar da türetilemez.

Static olarak tanımlanmış sınıfların sadece static üyeler içereceği kesin olmasına rağmen, üyelerinin static anahtar sözcüğü ile tanımlanıyor olması biraz tuhaf bir durum olarak görülebilir. Öyleki izlediğim bir iki blog sitesinde bu durum biraz alaycı ifadeler ile komik olarak ele alınmış. Tabi şu anda test ettiğimiz static class' ların beta 2 sürümüne ait olduğu düşünülecek olursa piyasaya çıkacak olan sürümde bu durum üzerinde iyileştirmeler yapılabilir.

Durumu özetleyecek olursak, **sadece static üyeler içermesini düşündüğümüz, örneklenmesini ve türetilmesini kesinlikle istemediğimiz sınıfların** tanımlanabilmesi için static anahtar sözcüğünü sınıflar için kullanabiliriz. Bu düşündüğümüz tasarım modelinin uygulamamız için yeterli olacaktır. Geldik bir makalemizin daha sonuna. Bu makalemizde C# 2.0' a gelen yeni özelliklerden brisi olan static sınıfları incelemeye çalıştık. Bir sonraki makalemizde görüşmek dileğiyle hepinize mutlu günler dilerim.

C# 2.0 ve Nullable Değer Tipleri - 22

Haziran 2005 Çarşamba

C# 2, value types, nullable,

Değerli Okurlarım Merhabalar,

C# programlama dilinde bildiğiniz gibi veri türlerini Referans Türleri(Reference Types) ve Değer Türleri (Value Types) olmak üzere iki kısma ayırıyoruz. Bu iki tür arasında bellek üzerinde fiziki tutuluş şekillerinden tutunda birbirleri arasındaki atamalara kadar pek çok farklılık vardır. Bu farklılıklardan birisi de, referans türlerinin null değerleri alabilmelerine karşın, değer türlerinin aynı özelliğe sahip olmayışlarıdır.

Bu durum özellikle veritabanı tablolarında null değer alabilen alanların, dil içerisindeki tip karşılığı değer türüne denk düştüğünde bazı zorluklar çıkartabilmektedir. İşte bu makalemizde, C# 2.0 diliyle gelen yeni özelliklerden birisi olan Nullable Value Types (Null değer alabilen değer türleri) ni incelemeye çalışacağız. Başlangıç olarak C# 1.1 versiyonundaki durumu analiz ederek işe başlayalım. Aşağıdaki kod parçasında bir referans türüne ve bir de değer türüne null değerler atanmaya çalışılmaktadır.

```
using System;
```

```
namespace ConsoleApplication2
{
    class Class1
    {
        static void Main(string[] args)
        {
            string refTuru=null;
            int degerTuru=null;
        }
    }
}
```

Bu uygulamayı derlemeye çalıştığımızda **Cannot convert null to 'int' because it is a value type** hatasını alırız. Aynı durumu kendi tanımladığımız referans ve değer türleri içinde gerçekleştirebiliriz. Aşağıdaki kod parçasında bu durum örneklenmiştir.

```
using System;
```

```
namespace ConsoleApplication2
{
    class Kitap
    {
```

```

    }
    struct Dvd
    {
    }
    class Class1
    {
        static void Main(string[] args)
        {
            Kitap kitap=null;
            Dvd dvd=null;
        }
    }
}

```

Bu kod parçasındada aynı hatayı alırız. Çünkü struct' lar değer türüdür ve bu sebeple null değerler alamazlar. Oysaki aynı istisnai durum class gibi kendi tanımlamış olduğumuz referans türleri için geçerli değildir. Peki değer türlerinin null değer içermeye ihtiyacı ne zaman doğabilir? Bir veritabanı uygulamasını göz önüne alalım. Bu tabloda int, double gibi değer türlerine karşılık gelecek alanların var olduğunu düşünelim. Veri girişi sırasında bu int ve double değişkenleri null olarak tabloya aktarmak isteyebiliriz.

Ya da tablodan veri çekerken, değer türü karşılığı alanların null değer içerip içermediğini anlamak isteyebiliriz. İşte bu gibi durumlarda değer türlerinin null veriler içerebilecek yapıda olması, kodumuzun ölçeklenebilirliğini arttıracak bir yetkinlik olarak düşünülebilir.

Veritabanları için geçerli olan bu senaryoyu göz önüne almadan önce C# 2.0 için değer türlerinin nasıl null veriler taşıyabileceğini incelemeye çalışalım. Değer türlerinin C# 2.0 için iki versiyonu vardır. Nullable değer türleri ve Non-Nullable değer türleri. Bir değer türünün null değerler içerecek tipte olacağını belirtmek için ? tip belirleyicisi kullanılır.

```

using System;
using System.Collections.Generic;
using System.Text;

```

```

namespace TestOfNullableValues
{
    class Program
    {
        static void Main(string[] args)
        {
            int? maas;
            double? pi;
            maas = null;
            pi = null;
        }
    }
}

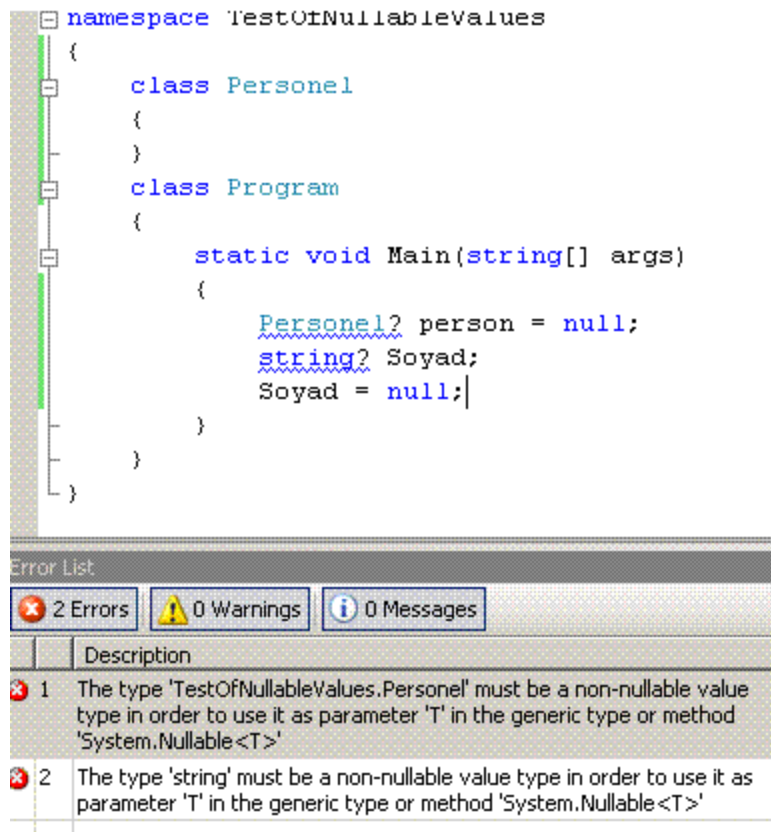
```

Yukarıdaki kod parçası sorunsuz olarak derlenecek ve çalışacaktır. ? ile tanımlanan değer türleri null veriler taşıyabilen değer tipindendir. Aynı durum kendi tanımladığımız bir struct içinde geçerlidir. Aşağıdaki kod parçasında Personel isimli struct' a null değer ataması yapılmıştır.

```
struct Personel
{
}

class Program
{
    static void Main(string[] args)
    {
        Personel? person = null;
    }
}
```

Elbette referans türlerinde bu tarz bir kullanım geçerli olmayacaktır. Örneğin aşağıdaki kod parçasında kendi tanımlamış olduğumuz bir sınıf nesnesine ve string türünden bir değişkene ? belirleyicisi vasıtasıyla null değerler atanmaya çalışılmıştır.



Oysaki referans türleri zaten null değerler alabilmektedir.



Referans türlerine ? tip belirleyicisi uygulayarak null değer ataması yapılamaz.

? tip belirleyicisi aslında tanımlanan değer türünün null veri taşıyabilecek başka bir versiyonunu kullanılacağını belirtmektedir. Bu yeni değer türü versiyonları ise null verileri taşıyabilecek bir yapıda tasarlanmışlardır. ? belirleyicisinin uygulandığı bir değer türünün null değerler içerip içermediğine **HasValue** özelliği ilede bakılabilmektedir. Bu özellik, ilgili değer türü null veri içerdiği sürece false döndürecektir. Örneğin,

```
int? Yas;  
Yas = null; // Yas değer türüne null veri atanıyor.  
if (Yas.HasValue) // false dönecektir  
{  
    Console.WriteLine("Yas null değil...");  
}  
else  
{  
    Console.WriteLine("Yas null..."); // Bu satır işletilir.  
}
```

C# 2.0 diline ? tip belirleyicisinden yola çıkılarak yeni bir operatör eklenmiştir. ?? operatörü. Bu operatör kısaca bir değer türünün içeriğinin null olup olmamasına göre koşullu olarak atama yapmaktadır. Eğer operatörde kullanılan null veri taşıyabilir değer türünün o anki değeri null ise, koşul olarak belirtilen değer ilgili değişkene atanır. Aksi takdirde, değer türünün o anki verisi, ilgili değişkene atanır. Aşağıdaki örnek kod parçasında bu operatörün kullanım şekli gösterilmeye çalışılmıştır.

```
int? Yas; // null değer alabilecek bir değer türü tanımlanıyor.
```

```
Yas = null; // null değer ataması  
int yasi = Yas ?? 0; // Eğer Yas null ise yasi alanına 0 atanır.  
Console.WriteLine(yasi); // 0 yazar
```

```
Yas = 12;  
yasi = Yas ?? 0; // Eğer Yas null değil ise yasi alanına Yas' ın o anki değeri atanır.  
Console.WriteLine(yasi); // 12 yazar
```

İlk kullanımda, Yas değer türünün sahip olduğu veri null olarak belirlenmiştir. ?? operatörü bu durumda yasi alanına 0 değerini atayacaktır. İkinci kullanımda ise Yas değer türünün verisi 12 olarak belirlenmiştir. Buna görede ?? operatörü yasi değişkenine 12 değerini (yani Yas nullable değer türünün o anki verisini) atayacaktır. Burada dikkat ederseniz Yas null değer alabilen bir int tipidir. Bununla birlikte ?? operatörü ile yapılan veri ataması null değerler içermeyen normal bir int tipine doğru yapılmaktadır. Burada söz konusu olan atama işlemini biraz irdelemekte fayda vardır. Aşağıdaki kod parçasını ele alalım.

```
int? pi = 3;  
int m_Pi;  
m_Pi = pi;
```

Bu örnek derlenmeyecektir. Bunun sebebi ise null değer alabilen bir değer türünü, normal bir değer türüne bilinçsiz olarak atamaya çalışmamızdır. Dolayısıyla,



Nullable Değer türleri bilinçsiz olarak normal değer türlerine dönüştürülemez.

Ancak aşağıdaki kod parçasında görülen tür dönüşüm işlemi geçerlidir.

```
int? pi = 3;  
int m_Pi;  
m_Pi = (int)pi;
```

Elbette burada dikkat edilmesi gereken bir durum daha vardır. Eğer o anki değeri null olan bir değer türünü normal bir değer türüne bilinçli olarak atamaya çalışırsak çalışma zamanında `InvalidOperationException` hatası alırız.

```
int? pi = null;  
int m_Pi;  
m_Pi = (int)pi;  
}  
}
```

name	Value
\$exception	{ "Nullable object must have a value." }
[System.InvalidOperationException]	{ "Nullable object must have a value." }

Dolayısıyla bu tip atamalarda ?? operatörünü tercih etmek çok daha akılcı bir yaklaşım olacaktır. Nullable değer türlerine, normal değer türlerinin atanmasında ise bilinçsiz tür dönüşümü de geçerlidir.

```
double e = 2.7;  
double? E;  
E = (double?)e; // Bilinçli tür dönüşümü  
E = e; // Bilinçsiz tür dönüşümü
```

Yukarıdaki kod parçasında E null değerler taşıyabilen bir değer türüdür. e ise normal değer türüdür.



Normal değer türleri, nullable değer türlerine hem bilinçli hem de bilinçsiz olarak dönüştürülebilir.

Makalemizin sonunda bir veritabanı uygulamasında null değerler alabilen değer türlerinin nasıl kullanılabildiğini incelemeye çalışacağız. İlk olarak C# 1.1 versiyonunda aşağıdaki yapıda bir uygulama geliştirelim. Bu örneğimizde, Sporculara ait bir takım temel bilgileri tutan bir tabloyu kullanacağız. Tablomuzda tanımlı olan Sporcu, Boy ve Yaş alanları null değerler içerebilecek şekilde yapılandırılmıştır.

Object Browser		Start Page	Class1.cs	dbo.Sporcular.
	Column Name	Data Type	Length	Allow Nulls
?	ID	int	4	
	Sporcu	nvarchar	50	✓
	Boy	float	8	✓
	Kilo	int	4	✓

Örnek olarak aşağıdaki iki satır verinin var olduğunu düşünelim. Dikkat ederseniz ikinci satırda değer türü olarak ele alacağımız alanlara <Null> değerler atanmıştır.

	ID	Sporcu	Boy	Kilo
▶	1	Michael Jordan	1,96	85
	2	Burak Selim Şenyurt	<NULL>	<NULL>
*				

Uygulama kodlarımız ise başlangıç olarak aşağıdaki gibidir. Basit olarak Sporcuları temsil edecek bir sınıf ve tablo üzerindeki temel veritabanı işlemlerini gerçekleştirecek bir katman sınıfı yer almaktadır. Katman sınıfımız şu an için sadece ve sadece Sporcu tablosundaki verileri okuyup, her bir satır için birer Sporcu nesnesi yaratan ve onun override edilmiş ToString metodu ile bilgilerini ekrana yazdıran bir metoda sahiptir.

```
using System;  
using System.Data;  
using System.Data.SqlClient;
```

```
namespace TestNullableValues  
{  
    class Sporcu  
    {  
        private int m_id;  
        private string m_sporcu;  
        private double m_boy;  
        private int m_kilo;  
        public Sporcu(int id,string sporcu,double boy,int kilo)
```

```

    {
        m_id=id;
        m_sporcu=sporcu;
        m_boy=boy;
        m_kilo=kilo;
    }
    public override string ToString()
    {
        return m_id.ToString()+" "+m_sporcu+" "+m_boy.ToString()+" "+m_kilo.ToString();
    }
}

class SporYonetim
{
    SqlConnection con;
    SqlCommand cmd;
    SqlDataReader dr;

    public SporYonetim()
    {
        con=new SqlConnection("data source=localhost;database=MyBase;user
id=sa;password=");
    }

    public void SporcuListesi()
    {
        cmd=new SqlCommand("Select ID,Sporcu,Boy,Kilo From Sporcular",con);
        con.Open();
        dr=cmd.ExecuteReader(CommandBehavior.CloseConnection);
        while(dr.Read())
        {
            int id=(int)dr["ID"];
            string sporcu=dr["Sporcu"].ToString();
            double boy=(double)dr["Boy"];
            int kilo=(int)dr["Kilo"];
            Sporcu sprc=new Sporcu(id,sporcu,boy,kilo);
            Console.WriteLine(sprc.ToString());
        }
    }
}

class Class1
{
    [STAThread]
    static void Main(string[] args)
    {
        SporYonetim ynt=new SporYonetim();
        ynt.SporcuListesi();
    }
}

```

```

    }
}
}

```

Bizim için önemli olan nokta Boy ve Yas alanlarının değerlerinin double ve int alanlara atıldığı yerdir. Uygulamamızı bu haliyle çalıştırdığımızda aşağıdaki hata mesajını alırız.

```

32 public SporYonetim()
33 {
34     con=new SqlConnection("data source=localhost;...");
35 }
36
37 public void SporcuListesi()
38 {
39     cmd=new SqlCommand("Select ID,Sporcu,Boy,Kilo...");
40     con.Open();
41     dr=cmd.ExecuteReader(CommandBehavior.CloseC...);
42     while(dr.Read())
43     {
44         int id=(int)dr["ID"];
45         string sporcu=dr["Sporcu"].ToString();
46         double boy=(double)dr["Boy"];
47         int kilo=(int)dr["Kilo"];
48         Sporcu sprc=new Sporcu(id,sporcu,boy,kilo);
49         Console.WriteLine(sprc.ToString());
50     }
51 }

```

Name	Value	Type
dr["Boy"]	{System.DBNull}	System.DBNull
dr["Kilo"]	{System.DBNull}	System.DBNull
dr["Sporcu"]	"Burak Selim Şenyurt"	string

Sebebi gayet net ve açıktır. Değer türleri null veri içermeyeceği için cast işlemleri çalışma zamanında InvalidCastException tipinden bir istisna nesnesi fırlatılmasına neden olmuştur. Bu sorunu aşmak için kullanabileceğimiz tekniklerden bir tanesi aşağıdaki gibidir. dr ile okunan tablo alanları object tipinden geriye döndüğünden ve object tipinde referans türü olduğundan null değerler içerebilir. Burada alanın o anki verisinin null olup olmamasına göre uygun atamalar gerçekleştirilmektedir.

```

double boy;
int kilo;
if(dr["Boy"]==System.DBNull.Value)
    boy=0;
else
    boy=(double)dr["Boy"];
if(dr["Kilo"]==System.DBNull.Value)
    kilo=0;
else

```



```
kilo=(int)dr["Kilo"];
Sporcu sprc=new Sporcu(id,sporcu,boy,kilo);
```

Aynı örneği aşağıdaki haliyle C# 2.0 versiyonunda daha farklı bir yaklaşım ile yazabiliriz. Bu kez, Sporcu sınıfımızın yapıcı metodunda yer alan boy ve kilo parametreleri ile private field olarak tanımladığımız m_Boy ve m_Kilo alanlarını null değer içerebilecek şekilde tanımlayarak işe başlayacağız. Yine SqlDataReader ile okuduğumuz alanların null değer içerip içermediğini kontrol edeceğiz. Ancak bu kez null değer içerseler dahi onları taşıyabilecek değer türlerimiz elimizde olacak. Böylece tablomuzda null değere sahip olan alanlarımızı ekrana yazdırabileceğiz.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Collections.Generic;
using System.Text;

namespace UsingNullableValues
{
    class Sporcu
    {
        //Diğer kod satırları

        // Sınıf içi bu iki alanı null değerler taşıyabilecek tipten tanımladık.
        private double? m_boy;
        private int? m_kilo;

        // Diğer kod satırları
    }

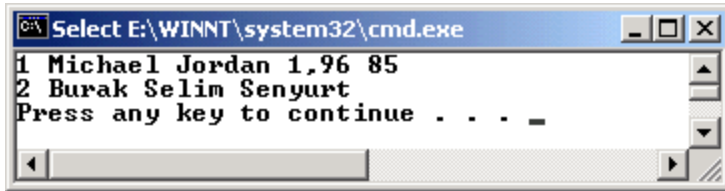
    class SporYonetim
    {
        // Diğer kod satırları

        public void SporcuListesi()
        {
            // Diğer kod satırları
            while (dr.Read())
            {
                // Diğer kod satırları

                double? boy=null;
                int? kilo=null;
                if(dr["Boy"]!=System.DBNull.Value)
                    boy=(double)dr["Boy"];
                if (dr["Kilo"] != System.DBNull.Value)
                    kilo = (int)dr["Kilo"];
            }
        }
    }
}
```

// nullable değişkenleri Sporcu sınıfının yapıcı metoduna parametre olarak gönderiyoruz.

```
Sporcu sprc = new Sporcu(id, sporcu, boy, kilo);
Console.WriteLine(sprc.ToString());
    }
}
}
class Program
{
    static void Main(string[] args)
    {
        SporYonetim ynt = new SporYonetim();
        ynt.SporcuListesi();
    }
}
```



Uygulamamızı bu haliyle çalıştırdığımızda tablodaki tüm satırların ekrana yazdırıldığını görürüz. C# 1.1 de geliştirdiğimiz uygulama ile bu versiyon arasındaki en önemli fark, nullable değer türlerinin kullanılarak null verilerin uygulama içerisinde kullanılabilmesidir. Bir sonraki makalemizde görüşmek dileğiyle hepinize mutlu günler dilerim.

C# 2.0 ile Partial Types (Kısmi Tipler) - 27 Haziran 2005 Pazartesi

C# 2, class, partial class, partial type,

Değerli Okurlarım Merhabalar,

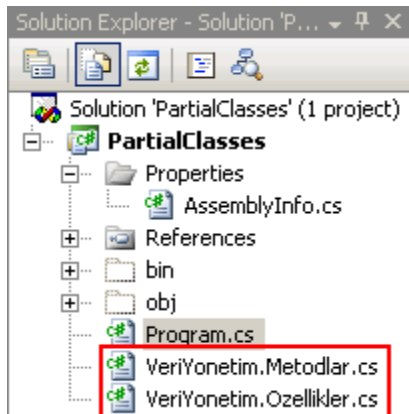
Visual Studio.Net ile windows veya web uygulamaları geliştirirken, kod yazılması sırasında karşılaştığımız güçlüklerden birisi, tasarım kodları ile kendi yazmış olduklarımızın iç içe geçmeleridir. Bu zamanla kodun okunabilirliğini zorlaştıran bir etmendir. Bunun windows uygulamalarını veya asp.net uygulamalarını geliştirirken sıkça yaşamaktayız. Bununla birlikte, özellikle soruce safe gibi ortamlarda farklı geliştiricilerin aynı sınıf kodları üzerinde eş zamanlı olarak çalışması pek mümkün değildir.

Visual Studio.2005 ile birlikte, sınıf (class), arayüz(interface) ve yapı(struct) gibi tipleri mantıksal olarak ayrıştırılabileceğimiz ve farklı fiziki dosyalarda (veya aynı fiziki dosya üzerinde) tutabileceğimiz yeni bir yapı getirilmiştir. Bu yapının kilit noktası tiplerin partial anahtar sözcüğü ile imzalanmasıdır. Partial olarak tanımladığımız tipleri farklı fiziki dosyalarda (veya aynı fiziki dosya içerisinde) tutabiliriz. Burada önemli olan, çalışma zamanında yazmış olduğumuz tipin mutlaka tek bir bütün olarak ele alınıyor olmasıdır.

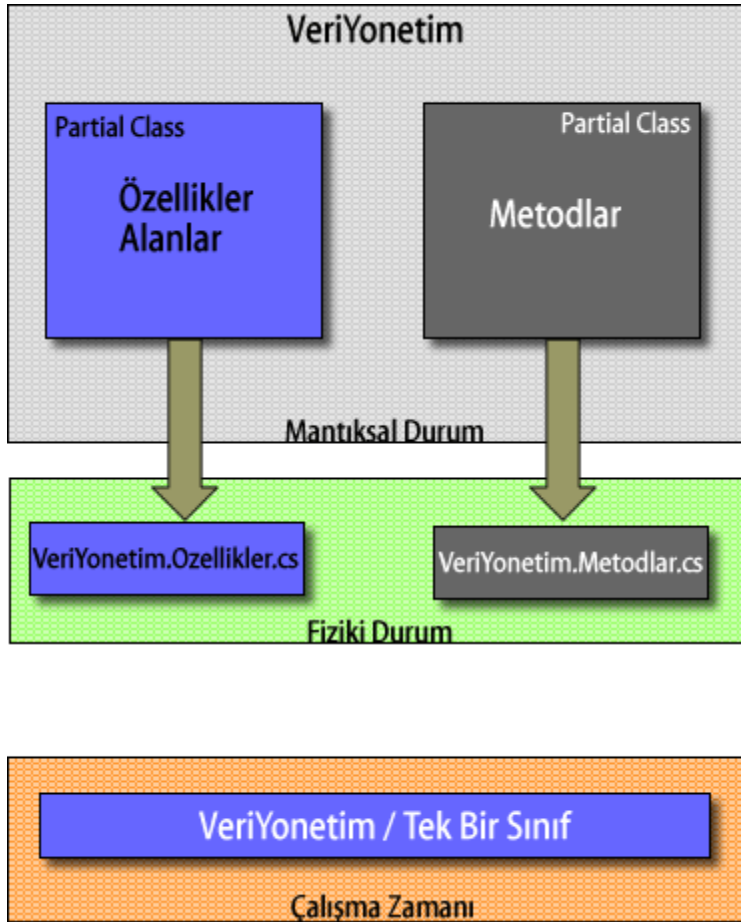


Partial tipler, bir tipin bütünü oluşturur soyutsal parçalardır.

İşte bu makalemizde kısaca partial tiplerin nasıl kullanıldığını incelemeye çalışacağız. Şimdi Visual Studio.2005 ile geliştirdiğimiz aşağıdaki örneği göz önüne alalım. Yeni açtığımız bir Console uygulamasında projemize şekilden de göreceğiniz gibi VeriYonetim.Ozellik ve VeriYonetim.Metod adlı iki kaynak kod dosyası ekledik.



Öncelikle buradaki amacımızdan bahsedelim. Veritabanı ile ilgili yönetim işlerini üstlenecek bir sınıf geliştirmek istiyoruz. Ancak bu sınıfın özelliklerini ve metodlarını ayrı fiziki kaynak kod dosyalarında tutacağız. Bu mantıksal ayrımı yapmamızın çeşitli nedenleri olabilir. Geliştiricilerin aynı sınıfın çeşitli mantıksal parçaları üzerinde bağımsız ama eş zamanlı olarak çalışmalarını isteyebiliriz. Çoğunlukla en temel nedemiz sınıf bütünü mantıksal olarak ayrıştırarak kodlamayı kolaylaştırmaktır. İşte bu amaçla VeriYonetim isimli sınıfımızın bu ayrı parçalarını tutacak iki fiziki sınıfı dosyasını projemize ekledik. Böylece sınıfımızı aslında iki mantıksal parçaya bölmüş olduk. İlk parçada sadece gerekli özellik tanımlamalarını ikinci parçada ise işlevsel metodları barındıracağız. Böylece kod geliştirme safhasında bu mantıksal parçaların iki farklı dosyada tutulmasını sağlamış oluyoruz.



Tasarım zamanında sınıfımızı iki farklı parçaya ayırmış olsakta, kod geliştirme sırasında veya çalışma zamanında tek bir tip olarak ele alınacaktır. Yani sınıfın bütünlüğü aslında hiç bir şekilde bozulmamaktadır. Bu önemli bir noktadır. Şimdi ilk örneğimizin kodlarına kısaca bakalım.

```
VeriYonetim.Ozellikler.cs;
```

```
using System;  
using System.Collections.Generic;
```

```

using System.Text;
using System.Data.SqlClient;

namespace PartialClasses
{
    partial class VeriYonetim
    {
        private SqlConnection m_GuncelBaglanti;
        private SqlCommand m_SqlKomutu;

        public SqlConnection GuncelBaglanti
        {
            get { return m_GuncelBaglanti; }
            set { m_GuncelBaglanti = value; }
        }
        public SqlCommand SqlKomutu
        {
            get { return m_SqlKomutu; }
            set { m_SqlKomutu = value; }
        }
    }
}

```

VeriYonetim.Metodlar.cs;

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Data.SqlClient;

namespace PartialClasses
{
    partial class VeriYonetim
    {
        public VeriYonetim(string baglanti)
        {
            m_GuncelBaglanti = new SqlConnection(baglanti);
        }
        public void KomutHazirla(string sorguCumlesi)
        {
            m_SqlKomutu = new SqlCommand(sorguCumlesi, GuncelBaglanti);
        }
    }
}

```

Uygulama ;

```

using System;

```

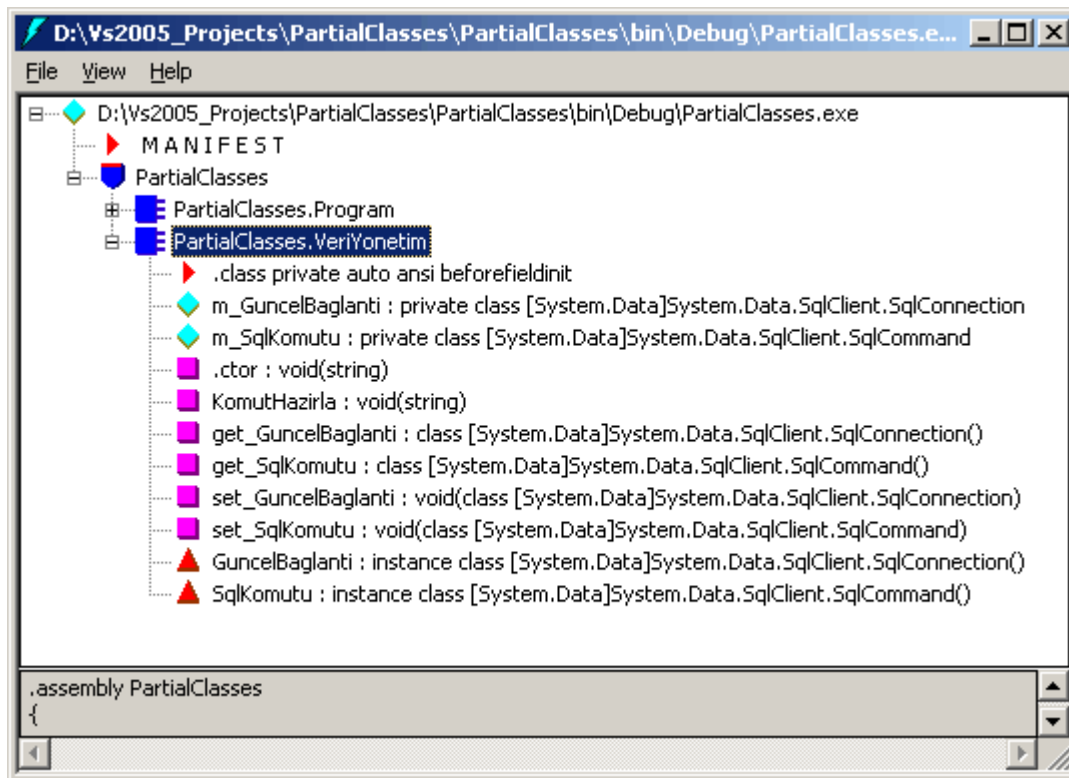
```

using System.Collections.Generic;
using System.Text;

namespace PartialClasses
{
    class Program
    {
        static void Main(string[] args)
        {
            VeriYonetim yonetici = new VeriYonetim("data
source=localhost;database=AdventureWorks;integrated security=SSPI");
            yonetici.KomutHazirla("SELECT * FROM Customers");
        }
    }
}

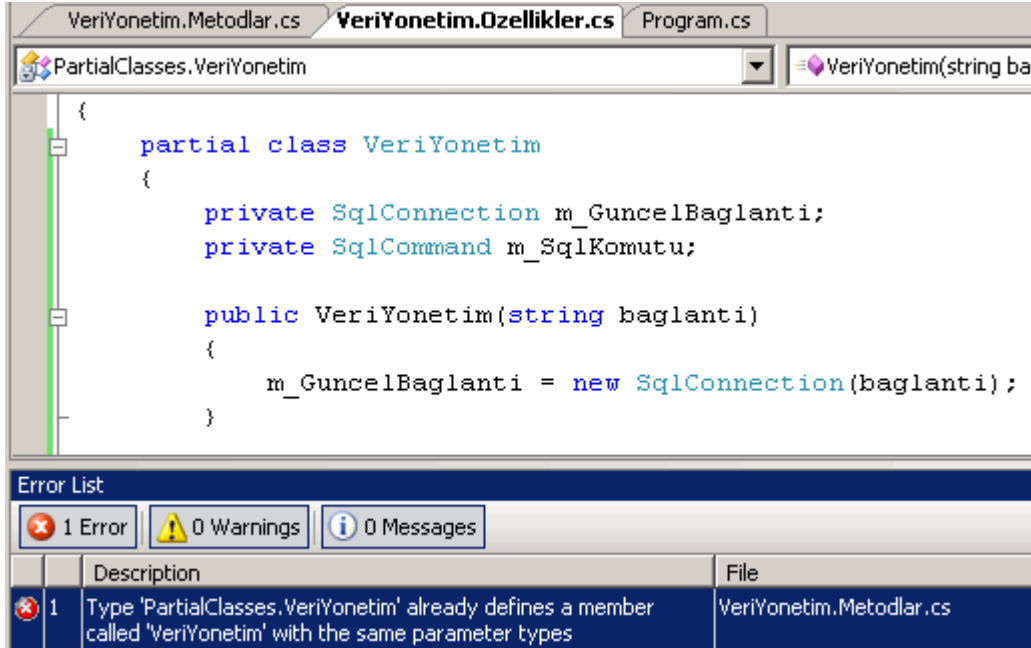
```

VeriYonetim isimli sınıfımızı her iki kaynak kod dosyası içerisinde partial anahtar kelimesi ile tanımlıyoruz. Böylece önceden bahsetmiş olduğumuz mantıksal ayrıştırmayı gerçekleştirmiş olduk. Eğer MS ILDasm (Microsoft Intermediate Language DisAssembler) programı yardımıyla derlediğimiz uygulamamıza göz atarsak ayrı parçalar halinde geliştirdiğimiz VeriYonetim sınıfının tek bir tip olarak yazıldığını görürüz.



Bizim partial sınıflar içerisine böldüğümüz özellik, alan ve metodlar burada tek bir çatı altında toplanmıştır. Partial sınıfları kullanırken elbetteki dikkat etmemiz gereken noktalar vardır. Örneğin partial sınıflar içerisinde aynı elemanları tanımlayamayız. Yukarıdaki örneğimizde özellikleri tuttuğumuz sınıf parçamıza metodları tuttuğumuz sınıf parçasında olan constructor (yapıcı metod) dan bir

tane daha ekleyelim. Özellikle aynı metod imzalarına sahip olmalarına dikkat edelim. Bu durumda uygulamayı derlemeye çalıştığımızda aşağıdaki hata mesajını alırız.



Bu hata bize, partial class' ların aslında tek bir sınıfı oluşturan parçalar olduğunu ispat etmektedir. Bu teoriden yola çıkarak metodların aşırı yüklenmiş (overload) hallerini partial class' lar içerisinde kullanabileceğimizi söyleyebiliriz. Bu sefer özellikleri tuttuğumuz partial sınıfımız içerisine VeriYonetim tipimiz için varsayılan yapıcı metodu (default constructor) ekleyelim.

```
partial class VeriYonetim
{
    public VeriYonetim()
    {
        m_GuncelBaglanti = new SqlConnection("data
source=localhost;database=AdventureWorks;integrated security=SSPI");
    }
    // Diğer kodlar
}
```

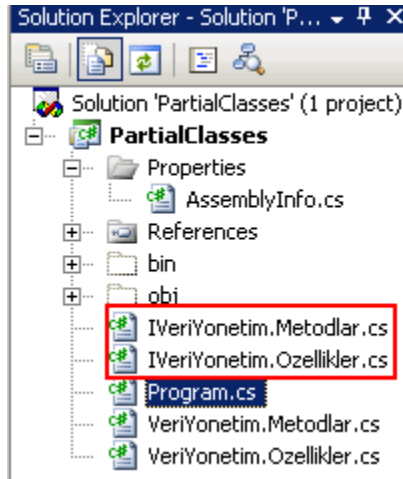
Bu durumda uygulamamız sorunsuz şekilde derlenecek ve çalışacaktır.



Partial sınıflar(classes) tanımlayabildiğimiz gibi partial arayüzler(interfaces) veya yapılarda(structs) tanımlayabiliriz.

Örneğin, parçaları aşağıdaki şekilden de görüldüğü gibi iki farklı fiziki kaynak dosyada tutulacak IVeriYonetim isimli bir arayüz oluşturmak istediğimizi

düşünelim. IVeriYonetim.Metodlar.cs kaynak kod dosyası içerisinde arayüzü uygulayacak olan tiplerin içermesi gereken metodları bildireceğimizi farzedelim. IVeriYonetim.Ozellikler.cs kaynak kod dosyasında ise, bu arayüzü uygulayacak tiplerin içermesi gereken özellik bildirimlerini yapacağımızı varsayalım.



Tek yapmamız gereken arayüzlerimizi mantıksal olarak ayırtırmak ve partial anahtar kelimesini kullanarak kodlamak olacaktır.

IVeriYonetim.Ozellikler.cs dosyası;

```
using System;  
using System.Collections.Generic;  
using System.Text;
```

```
namespace PartialClasses  
{  
    partial interface IVeriYonetim  
    {  
        int MaasArtisOrani  
        {  
            get;  
            set;  
        }  
    }  
}
```

IVeriYonetim.Metodlar.cs dosyası;

```
using System;  
using System.Collections.Generic;  
using System.Text;
```

```
namespace PartialClasses  
{  
    partial interface IVeriYonetim
```



```

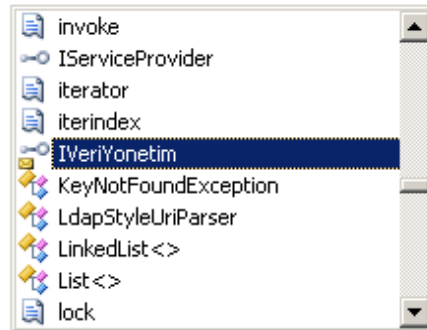
{
    void Bilgilendir();
    int PersonelSayisi(string sqlCumlesi);
}
}

```

Her iki interface tanımlamasıda aslında tek bir interface' in kendi kurduğumuz mantıksal mimari çerçevesinde ayrılmış soyut parçalardır. Keza, IVeriYonetim isimli arayüzümüzü her hangi bir sınıfa uyguladığımızda aşağıdaki şekilden de görüldüğü gibi arayüzümüzün parçaları değil bütünlüğü ele alınacaktır.

```
class VeriYoneticisi:IV
```

```
{
}
```



interface PartialClasses.IVeriYonetim

Sınıflarda olduğu gibi interface' ler içinde IL kodunun verdiği görünüm benzer olacaktır. Arayüzü ilgili sınıfımıza uyguladığımızda partial bölümlerde yer alan tüm elemanların implementasyona dahil edildiğini görürüz.

```
class VeriYoneticisi:IVeriYonetim
```

```
{
```

```
    #region IVeriYonetim Members
```

```
    // IVeriYonetim.Ozellikler.cs kaynak kod dosyası içerisindeki partial kısımdan gelen üyeler.
```

```
    public void Bilgilendir()
```

```
    {
```

```
        //
```

```
    }
```

```
    public int PersonelSayisi(string sqlCumlesi)
```

```
    {
```

```
        //
```

```
    }
```

```
    // IVeriYonetim.Metodlar.cs kaynak kod dosyası içindeki partial kısımdan gelen üyeler.
```

```
    public int MaasArtisOrani
```

```
    {
```

```
        get
```

```
        {
```

```
            //
```

```
        }
```

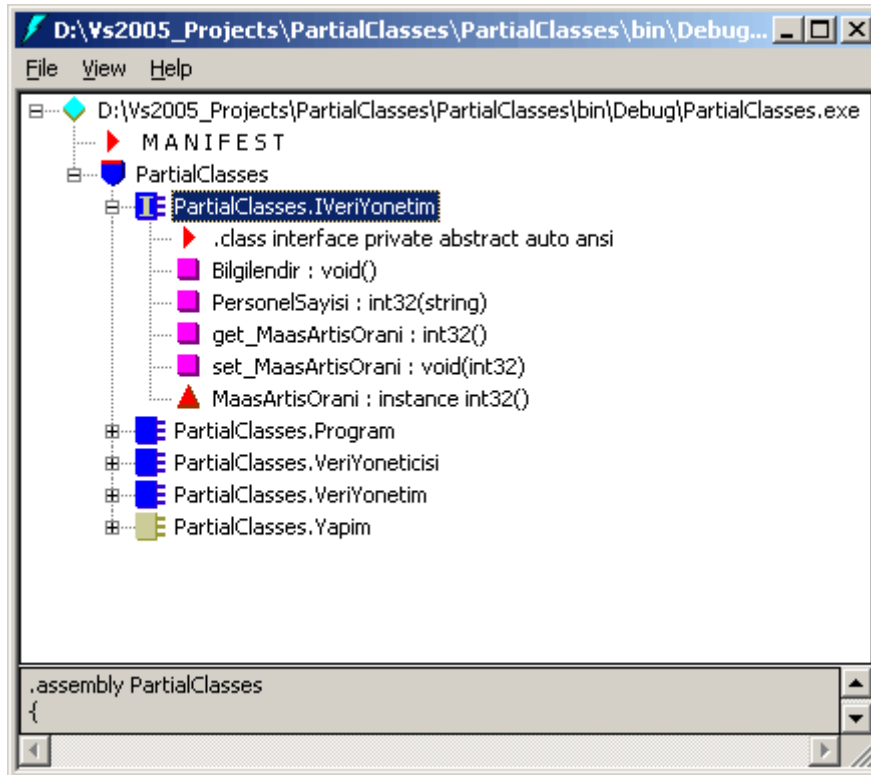
```
    }
```

```

    set
    {
        //
    }
}
#endregion
}

```

Tek bir çatı altında toplanmış kod içerisinde mantıksal olarak ayrı parçalara bölünmüş bir tek arayüz.



Partial kısımların, sturct' larda kullanılış biçimleride sınıflar ve arayüzlerde olduğu gibidir.

Özellikle partial sınıflar için bir takım kurallar vardır. Bu kurallara göre partial sınıflardan her hangi biri abstract veya sealed olarak belirtilirse, söz konusu tipin tamamı bu şekilde değerlendirilir. Örneğin aşağıdaki uygulamada partial sınıflarımızdan birisi sealed olarak tanımlanmıştır. Bunun doğal sonucu olarak söz konusu olan tip sealed olarak ele alınır. Dolayısıyla bu tipten her hangi bir şekilde türetme işlemi gerçekleştirilemez. Örneğin aşağıdaki kod derleme zamanında hataya yol açacaktır.

```

sealed partial class Islemler
{
}
partial class Islemler
{
}

```

```

}
class AltIslemler : Islemler
{
}

```

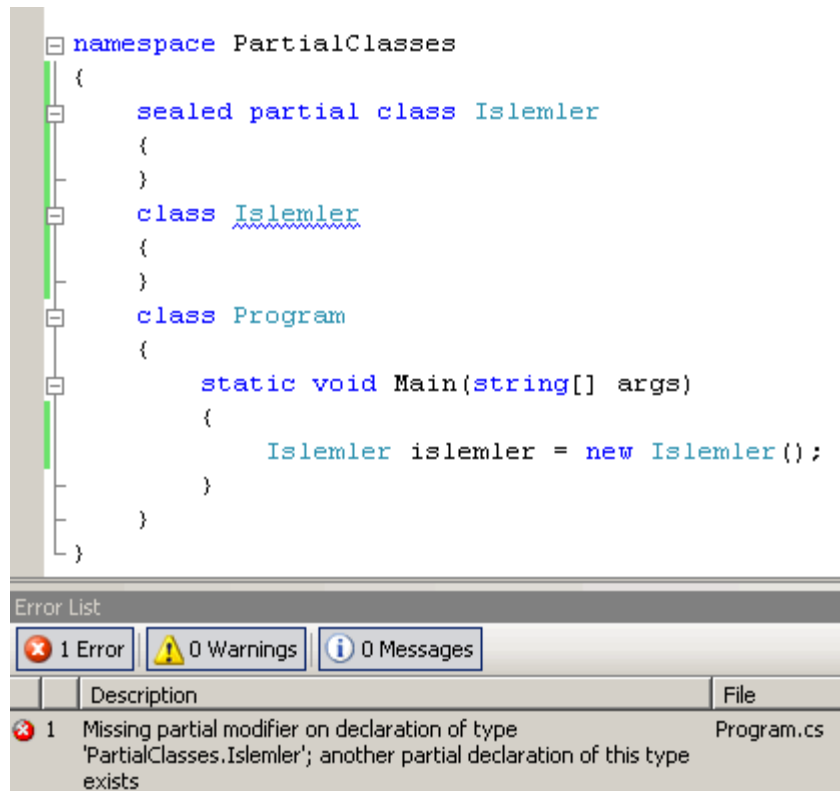
Peki Partial sınıfları birden çok defa sealed tanımlarsak ne olur? Örneğin yukarıdaki kodumuzda Islemler sınıfının her iki parçasınıda sealed anahtar sözcüğünü uyguladığımızı düşünelim.

```

sealed partial class Islemler
{
}
sealed partial class Islemler
{
}

```

Burada ki gibi bir kullanım tamamen geçerlidir. Derleme zamanında her hangi bir hata alınmaz. Parital tiplerin kullanımı ile ilgili bir diğer kısıtlamaya göre partial olarak imzalanmış bir sınıfı partial olamayan bir sınıf olarak tekrardan yazamayız. Yani aşağıdaki ekran görüntüsünde olduğu gibi Islemler isimli sınıfı hem partial olarak hem de normal bir sınıf olarak tanımlayamayız. Böyle bir kullanım sonrasında derleme zamanı hatasını alırız.



Görüldüğü gibi sınıflarımızı, arayüzlerimizi veya yapılarımızı partial olarak tanımlamak son derece kolaydır. Tek yapmanız gereken partial anahtar sözcüğünü kullanmaktır. Asıl zor olan, bu tipleri bölme ihtiyacını tespit edebilmek ve ne şekilde parçalara ayırabileceğimize karar vermektir. Yani bir sınıfı gelişmiş güzel parçalara bölmektense bunun için geçerli bir sebep aramak son derece

önemlidir. Ayrıca mantıksal bölümlemeyi çok iyi analiz etmemiz gerekir. Bu analizin en iyi çözümü sunabilmesi ise sizlerin proje tecrübenize, planlama ve öngörü yeteneklerinize bağlıdır. Böylece geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

C# 2.0 Covariance ve Contravariance Delegates - 30 Haziran 2005 Perşembe

C# 2, covariance generic, contravariance generic, covariance, delegate,

Değerli Okurlarım Merhabalar,

Bildiğiniz gibi temsilciler (delegates) çalışma zamanında metodların başlangıç adreslerini işaret eden tiplerdir. Bu tipleri uygulamalarımızda tanımlarken çalışma zamanında işaret edebilecekleri metodların geri dönüş tipi ve parametrik yapılarını bildirecek şekilde oluştururuz. Ancak özellikle , C# 1.1 ortamında temsilcilerin kullanımında parametre ve dönüş tipleri açısından iki önemli sıkıntımız vardır. Bu sıkıntıların kaynağında birbirlerinden türeyen yani aralarında kalıtımsal (inheritance) ilişkiler olan tipler yer alır.

Nitekim temsilciler C# 1.1 versiyonunda oluşturulacakları zaman, işaret edecekleri metodlar için kesin dönüş ve parametre tipi uyumluluğunu ararlar. Dolayısıyla bu tiplerde kalıtımsal ilişkiler söz konusu olduğunda ortaya çıkan iki temel problem vardır. Bu iki temel sorunumuza bakmadan önce, konunun odağında aralarında kalıtım ilişkisi olan iki sınıf olduğunu göz önüne almalıyız. Örneğin Sekil ve Dortgen isimli iki sınıfımız olduğunu ve Dortgen sınıfının Sekil sınıfından türediğini varsayalım. Bu sınıfların sahip olduğu içeriğin bizim için şu aşamada çok fazla önemi yoktur. İlk olarak covariance' lığa neden olan sorunu ele alalım. Aşağıdaki uygulamamızı dikkatle inceleyelim.

```
using System;
```

```
namespace UsingCovariance
```

```
{
```

```
    public class Sekil
```

```
    {
```

```
        public Sekil()
```

```
        {
```

```
        }
```

```
    }
```

```
    public class Dortgen: Sekil
```

```
    {
```

```
        public Dortgen()
```

```
        {
```

```
        }
```

```
    }
```

```
    public delegate Sekil Temsilci();
```

```
    class Class1
```

```

{
    public static Sekil Metod_1()
    {
        return null;
    }

    public static Dortgen Metod_2()
    {
        return null;
    }

    [STAThread]
    static void Main(string[] args)
    {
        Temsilci temsilci=new Temsilci(Metod_1);
        temsilci=new Temsilci(Metod_2); // Derleme zamanı hatası
    }
}

```

Bu uygulamayı derlediğimizde,

```
temsilci=new Temsilci(Metod_2);
```

satırı için **Method 'UsingCovariance.Class1.Metod_2()' does not match delegate 'UsingCovariance.Sekil UsingCovariance.Temsilci()'** hatasını alırız. Peki burada sorun nedir? Temsilci isimli delegate tipimiz, Sekil sınıfından nesne örneklerini geriye döndüren ve parametre almayan metodları işaret edebilecek şekilde tanımlanmıştır . Bu durumda,

```
Temsilci temsilci=new Temsilci(Metod_1);
```

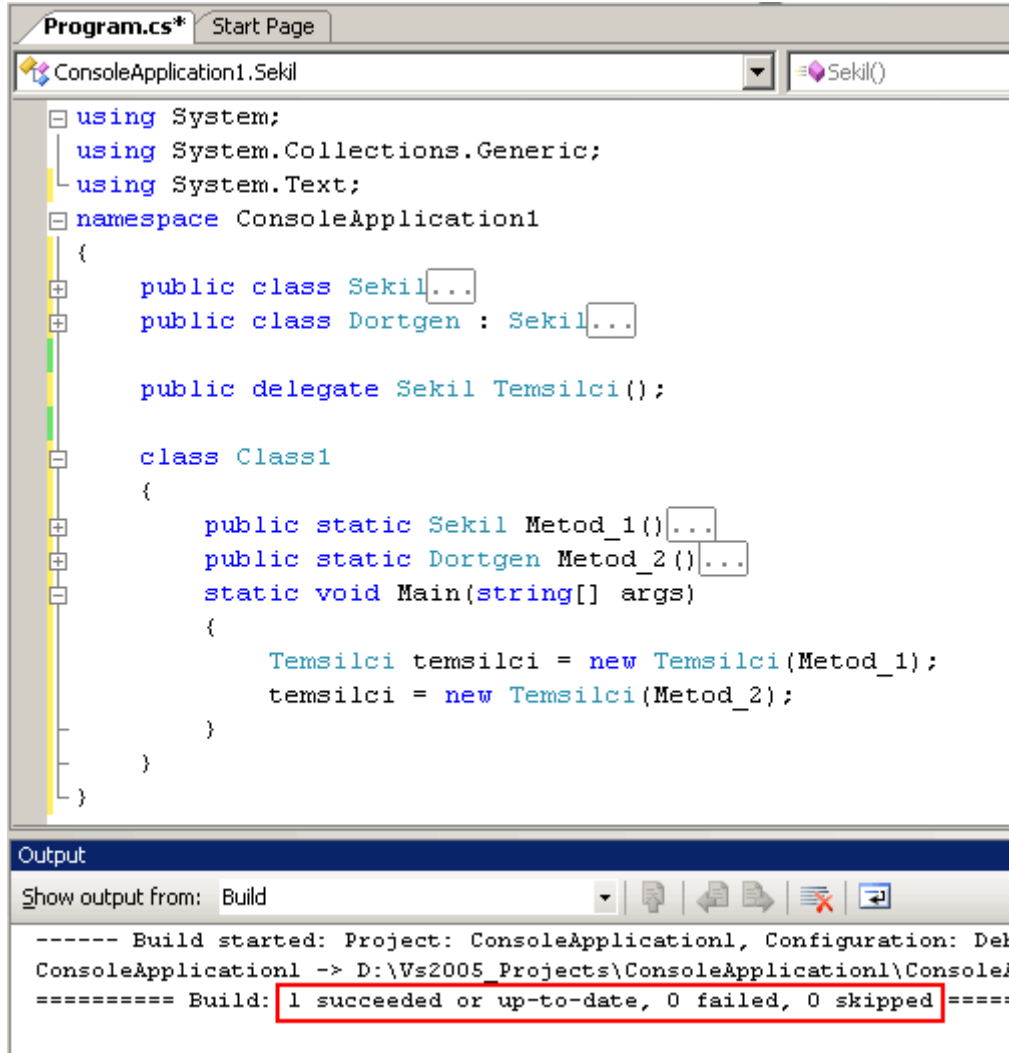
satırı düzgün olarak çalışacaktır. Nitekim Metod_1 delegate tipimizin tanımlamalarına uyan tarzda bir metoddur. Oysaki Metod_2 metodumuzun geriye döndürdüğü değer Dortgen sınıfı tipindendir. Dolayısıyla temsilci nesnemizin tanımladığı bildirimin dışında bir dönüş tipinin dönüşü söz konusudur. Hatırlayın, temsilciler işaret edecekleri metodlar için kesin dönüş tipi ve parametre tipi uyumluluğu ararlar. Oysaki Dortgen ve Sekil sınıfı arasında kalıtsal bir ilişki söz konusudur ve bu sebeple bu tarz bir kullanımın sorunsuz olarak çalışacağı düşünülmektedir. Çünkü kalıtımın doğası gereği bu Dortgen sınıfına ait nesne örnekleri Sekil sınıfına ait nesne örneklerine dönüştürülebilir. Ancak temsilcimiz açısından bu kural ne yazık ki geçerli değildir. Yani temsilcimiz belirtilen tipler için çalışma zamanında çok biçimliliği destekleyememiştir.

Peki bu sorunu nasıl çözebiliriz? Tek yapabileceğimiz Dortgen sınıfına ait nesne örneklerini döndüren Metod_2 isimli metodumuzu işaret edecek yeni bir delegate nesnesini aşağıdaki örnek kod parçasında olduğu gibi tanımlamak ve kullanmak olacaktır. Bu tahmin edeceğiniz gibi hantal bir çözümdür. Bir birleri arasında tür dönüşümü yapılabilecek iki sınıf için gereksiz yere iki ayrı temsilci tipi tanımlamak zorunda kalışımız dahi istenmeyen bir durumdur.

```
public delegate Sekil Temsilci();  
public delegate Dortgen Temsilci2();
```

```
class Class1  
{  
    public static Sekil Metod_1()  
    {  
        return null;  
    }  
  
    public static Dortgen Metod_2()  
    {  
        return null;  
    }  
  
    [STAThread]  
    static void Main(string[] args)  
    {  
        Temsilci temsilci=new Temsilci(Metod_1);  
        Temsilci2 temsilci2i=new Temsilci2(Metod_2);  
    }  
}
```

İşte C# 2.0' da, delegate tipi için söz konusu olan bu sorun ortadan kaldırılmıştır. Yukarıdaki örneğimizin aynısını C# 2.0' da yazdığımızı düşünürsek her hangi bir derleme zamanı hatası almayız. Covariance' ın desteklenebilmesi için C# 2.0' a getirilen ekstra bir kodlama tekniği yoktur.



The screenshot shows a Visual Studio IDE window with a C# file named 'Program.cs*'. The code defines a namespace 'ConsoleApplication1' containing a base class 'Sekil', a derived class 'Dortgen' that inherits from 'Sekil', and a delegate 'Sekil Temsilci()'. A static class 'Class1' implements two static methods, 'Metod_1()' and 'Metod_2()', both of which return a 'Sekil' object. The 'Main' method in 'Class1' creates two instances of the 'Temsilci' delegate, one pointing to 'Metod_1' and the other to 'Metod_2'. Below the code editor, the 'Output' window shows the build results for the 'Build' configuration, indicating that the build was successful: '1 succeeded or up-to-date, 0 failed, 0 skipped'.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    public class Sekil...
    public class Dortgen : Sekil...

    public delegate Sekil Temsilci();

    class Class1
    {
        public static Sekil Metod_1()...
        public static Dortgen Metod_2()...
        static void Main(string[] args)
        {
            Temsilci temsilci = new Temsilci(Metod_1);
            temsilci = new Temsilci(Metod_2);
        }
    }
}
```

Output

Show output from: Build

----- Build started: Project: ConsoleApplication1, Configuration: Debug
ConsoleApplication1 -> D:\Vs2005_Projects\ConsoleApplication1\ConsoleApplication1.exe
===== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped =====

Bu tamamen .net çekirdeğinde delegate tipi üzerinde yapılan bir düzenlemenin sonucudur.



Covariance, temsilcilerin çalışma zamanında işaret etmek istediği metodların, aralarında kalımsal ilişki olan dönüş tipleri arasındaki poliformik uyum sorununu ortadan kaldıran bir özellik olarak nitelendirilebilir.

Temsilci nesnelerimizin işaret edeceği metodların dönüş tiplerinin kalımsal ilişkilere izin verecek şekilde düzenlenmiş olması elbetteki çalışma zamanında bize büyük bir esneklik getirmektedir. Nitekim bu sayede birbirlerinden türemiş n sayıda sınıfa ait nesne örneklerinin dönüş tipi olarak kullanıldığı metodları tek bir delegate nesnesi vasıtasıyla çalışma zamanında işaret edebiliriz.

Gelelim contravariance durumuna. Bu kez kalıtım ilişkisine sahip olan sınıf örnekleri, temsilcilerin işaret edeceği metodların parametrik yapıları içerisinde

kullanılmaktadır. Aşağıdaki örnek kod parçası C# 1.1 için bu durumu örneklemektedir.

```
using System;

namespace UsingContravariance
{
    public class Sekil
    {
        public Sekil()
        {
        }
    }
    public class Dortgen:Sekil
    {
        public Dortgen()
        {
        }
    }

    public delegate int Temsilci(Dortgen dortgen);

    class Class1
    {
        public static int Metod_1(Dortgen dortgen)
        {
            return 0;
        }
        public static int Metod_2(Sekil sekil)
        {
            return 0;
        }

        [STAThread]
        static void Main(string[] args)
        {
            Temsilci temsilci=new Temsilci(Metod_1);
            temsilci=new Temsilci(Metod_2); // Derleme zamanı hatası
        }
    }
}
```

Bu kez delegate tipimiz geriye int tipinden değer döndüren ve parametre olarak Dortgen sınıfı tipinden nesne örneklerini alan metodları işaret edebilecek şekilde tanımlanmıştır. Kodu derlediğimizde *Method 'UsingContravariance.Class1.Metod_2(UsingContravariance.Sekil)' does not match delegate 'int UsingContravariance.Temsilci(UsingContravariance .Dortgen)'* hatasını alırız. Yine delegate tipimiz burada parametrik imza uyumsuzluğundan bahsetmektedir. Sorun,

```
temsilci=new Temsilci(Metod_2);
```

satırında oluşur. Çünkü Metod_2 Dortgen sınıfı tipinden bir nesne örneğini parametre olarak almaktansa Dortgen sınıfının üst sınıfı olan Sekil sınıfından bir nesne örneğini parametre olarak almaktadır. Buradaki problem tersi durum içinde söz konusudur. Yani temsilcimizi tanımlarken metodun alacağı parametrenin, türeyen sınıf yerine temel sınıftan (base class) bir nesne örneğini kullanacak şekilde aşağıdaki kod parçasında görüldüğü gibi tanımlandığını düşünersek;

```
public delegate int Temsilci(Sekil sekil);
```

```
class Class1
```

```
{
    public static int Metod_1(Dortgen dortgen)
    {
        return 0;
    }
    public static int Metod_2(Sekil sekil)
    {
        return 0;
    }

    [STAThread]
    static void Main(string[] args)
    {
        Temsilci temsilci=new Temsilci(Metod_1); // Derleme zamanı hatası
        temsilci=new Temsilci(Metod_2);
    }
}
```

bu sefer Metod_1 için oluşturulan temsilci isimli nesnemiz üzerinden aynı hata mesajını alırız. Doğal olarak, metod parametlerinin üst sınıftan veya türeyen sınıftan olması durumu değiştirmeyecektir.



Contravariance, temsilcilerin çalışma zamanında işaret etmek istediği metodların parametreleri arasında kalıtımsal ilişkiye sahip tiplerin neden olduğu polimorfik uyum sorununu ortadan kaldıran bir özellik olarak nitelendirilebilir.

Parametrelerin uyumsuzluğunun neden olduğu bu sorunu C# 1.1 ile çözmek için, covariance tekniğinde olduğu gibi her bir metod için ayrı temsilci nesnelerini aşağıdaki kod parçasında olduğu gibi tanımlamamız gerekecektir.

```
public delegate int Temsilci(Sekil sekil);
public delegate int Temsilci2(Dortgen dortgen);
```

```
class Class1
```

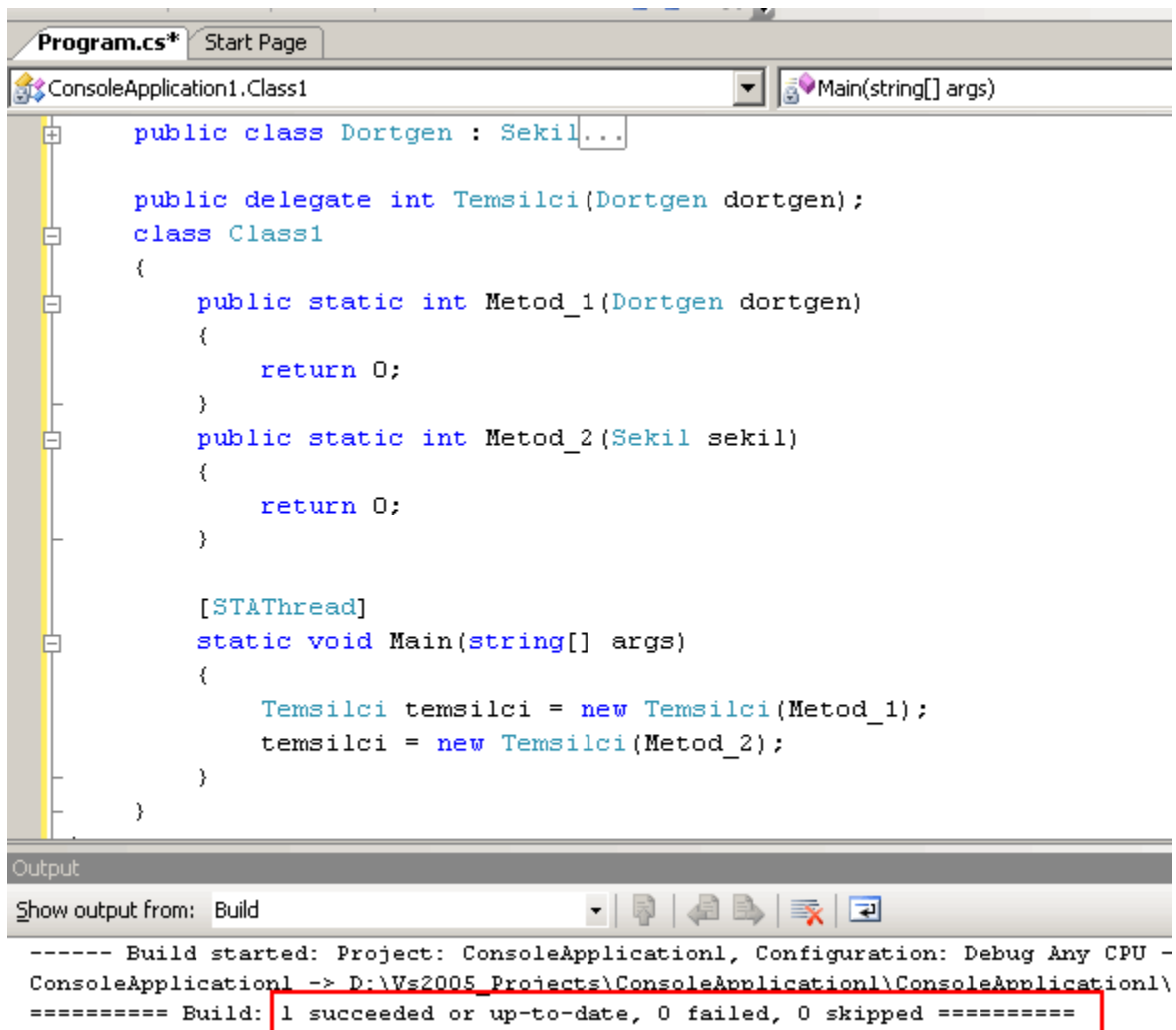
```

{
    public static int Metod_1(Sekil sekil)
    {
        return 0;
    }
    public static int Metod_2(Dortgen Dortgen)
    {
        return 0;
    }

    [STAThread]
    static void Main(string[] args)
    {
        Temsilci temsilci=new Temsilci(Metod_1);
        Temsilci2 temsilci2=new Temsilci2(Metod_2);
    }
}

```

Yukarıdaki örneğimizi C# 2.0 ile derlediğimizde ise her hangi bir sorun olmadığını görürüz.



Parametrelerin arasındaki kalıtımsal ilişki, çalışma zamanında temsilciler oluşturulurken de değerlendirilecek ve nesneler arasındaki tür dönüşümü başarılı bir şekilde parametrelerde yansıyacaktır. Bu covarince probleminde olduğu gibi bize yine büyük bir esneklik sağlar. Aralarında kalıtımsal ilişki olan n sayıda sınıf nesne örneğini kullanan metodları tek bir temsilci nesnesi ile çalışma zamanında işaret edebilme yeteneği. Böylece geldik bir makalemizin daha sonuna, bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

C# 2.0 İçin İterasyon Yenilikleri - 05

Temmuz 2005 Salı

C# 2, yield, iterations,

Değerli Okurlarım Merhabalar,

Bazen kendi yazmış olduğumuz tiplerin dizi bazlı elemanları olur. Uygulamalarımızda, bu elemanlar arasında, elemanların sahibi olan nesne örneği üzerinden ileri yönlü iterasyonlar kurmak isteyebiliriz. Foreach döngüleri belirtilen tip için bu iterasyonu sağlayan bir mekanizmaya sahiptir. Lakin kendi geliştirdiğimiz tiplerin sahip oldukları elemanlar üzerinde, bu tarz bir iterasyonu uygulayabilmek için bir numaratóre ve uygulayıcıya ihtiyacımız vardır. Kısacası, tipimizin elemanları arasında nasıl öteleme yapılabileceğini sisteme öğretmemiz gerekmektedir. Bu işlevselliği kazandırmak için IEnumerable ve IEnumerator arayüzlerini birlikte kullanırız.

Uygulamalarımızda klasik olarak kullandığımız bir iterasyon tekniği vardır. Bu tekniğe göre iterasyon içerisinde kullanılacak olan elemanlar için bir numaratór sağlanır. IEnumerable arayüzünün sağladığı GetEnumerator metodu geriye IEnumerator arayüzü tipinden bir nesne örneği döndürmektedir ve bizim için gerekli olan numaratórün kendisidir. IEnumerator ise çoğunlukla dahili bir sınıfa (inner class) uygulanır ve iterasyon için gerekli asıl metodları sağlar. Bu metodlardan MoveNext bir sonraki elemanın olup olmadığını bool tipinden döndüren bir işleve sahip iken, Current metodu o anki elemanı iterasyona devreder. Bu iki arayüzü kullanarak bir sınıf içindeki elemanlar üzerinde iterasyona izin verecek yapıyı kurmak biraz karmaşıktır. Aslında teknik son derece kolaydır sadece uygulanabilirliği ilk zamanlarda biraz kafa karıştırıcıdır. İlk olarak C# 1.1 için bahsetmiş olduğumuz iterasyon tekniğinin nasıl gerçekleştirildiğini inceleyeceğimiz bir örnek geliştirelim.

Örneğimizde her hangi bir işi simgeleyen bir sınıfımız olacak. Bu sınıfımızın modeli basit olarak aşağıdaki kod parçasında olduğu gibidir. IsBilgisi isimli sınıfımız basit olarak bir işin adını ve sorumlusuna ait bilgileri tutacak şekilde tasarlanmıştır. İşe ait bilgileri tutan iki özelliği ve aşırı yüklenmiş(overload) bir yapıcı metodu(constructor) mevcuttur.

```
public class IsBilgisi
{
    private string m_IsAdi;
    private string m_IsSorumlu;
    public string IsAdi
    {
        get
        {
```

```

        return m_IsAdi;
    }
    set
    {
        m_IsAdi=value;
    }
}
public string IsSorumlu
{
    get
    {
        return m_IsSorumlu;
    }
    set
    {
        m_IsSorumlu=value;
    }
}

public IsBilgisi(string isAdi,string isSorumlu)
{
    m_IsAdi=isAdi;
    m_IsSorumlu=isSorumlu;
}
}

```

Şimdi IsBilgisi sınıfı tipinden 3 elemanı bünyesinde barındıracak bir listeleme sınıfı tasarlayacağız. Amacımız IsBilgisi tipinden dizinin elemanlarına foreach döngüsünü kullanarak IsListesi nesnesi üzerinden erişebilmek. Yani aşağıdaki kodun çalıştırılmasını sağlamak istiyoruz. Burada dikkat ederseniz foreach döngümüz, *"liste isimli IsListesi örneğindeki her bir IsBilgisi tipinden nesne örneği için ilerle"* gibisinden bir iterasyon gerçekleştirmektedir.

```

IsListesi liste=new IsListesi();
foreach(IsBilgisi i in liste)
{
    Console.WriteLine(i.IsAdi+" "+i.IsSorumlu);
}

```

Normal şartlarda eğer IsListesi isimli sınıfımıza IEnumerable ve IEnumerator arayüzlerini kullandığımız iterasyon tekniğini uygulamazsak bu kod derleme zamanında foreach döngüsünün uygulanamayacağına dair hata mesajı verecektir. Yani IsListesi sınıfımızın kodunun başlangıçta aşağıdaki gibi olduğunu düşünersek;

```

public class IsListesi
{
    static IsBilgisi[] Isler=new IsBilgisi[3];
    private void ListeOlustur()

```

```

{
    IsBilgisi is1=new IsBilgisi("Birinci iş","Burak");
    IsBilgisi is2=new IsBilgisi("İkinci iş","Jordan");
    IsBilgisi is3=new IsBilgisi("Üçüncü iş","Vader");
    Isler[0]=is1;
    Isler[1]=is2;
    Isler[2]=is3;
}

public IsListesi()
{
    ListeOlustur();
}
}

```

derleme zamanında aşağıdaki hata mesajını alırız.



*foreach statement cannot operate on variables of type 'Using_Iterators_CSharp_1.IsListesi' because 'Using_Iterators_CSharp_1.IsListesi' **does not contain a definition for 'GetEnumerator'**, or it is inaccessible*

Çünkü foreach döngüsünün IsListesi sınıfı içindeki IsBilgisi nesnelere nasıl erişeceği ve onlar üzerinde ileri yönlü nasıl hareket edeceği bilinmemektedir. Bu nedenle IsListesi sınıfımız aşağıdaki gibi oluşturulmalıdır. Buradaki amacımız C# 1.1 için iterasyon tekniğini incelemek olduğundan ana fikirde IsBilgisi sınıfı tipinden 3 elemanlı bir dizi kullanılmaktadır.

// İterasyonu sağlayabilmek için IEnumerable arayüzünü uyguluyoruz.

```

public class IsListesi:IEnumerable
{
    // IsBilgisi tipinden nesne örneklerini taşıyacak dizimiz tanımlanıyor.
    static IsBilgisi[] Isler=new IsBilgisi[3];
    // Isler isimli diziye dolduracak basit bir metod.
    private void ListeOlustur()
    {
        IsBilgisi is1=new IsBilgisi("Birinci iş","Burak");
        IsBilgisi is2=new IsBilgisi("İkinci iş","Jordan");
        IsBilgisi is3=new IsBilgisi("Üçüncü iş","Vader");
        Isler[0]=is1;
        Isler[1]=is2;
        Isler[2]=is3;
    }

    public IsListesi()
    {

```

// IsListesi nesne örneğimiz oluşturulurken Isler isimli dizimizde IsBilgisi tipinden elemanlar ile dolduruluyor.

```
ListeOlustur();  
}
```

#region IEnumerable Members

// GetEnumerator metodu IsListesi sınıfımızdaki Isler dizisinin elemanlarında hareket edebilmemiz için gerekli numarator nesne örneğini geriye döndürüyor

```
public IEnumerator GetEnumerator()  
{  
    return new Numarator();  
}
```

#endregion

// Isler dizisindeki elemanlarda foreach döngüsünü kullanarak IsListesi sınıfı üzerinden ileri yönlü ve yalnız okunabilir iterasyon yapmamızı sağlayacak inner class' ımızı oluşturuyor ve bu sınıfa IEnumerator arayüzünü uyguluyoruz.

```
private class Numarator:IEnumerator  
{  
    // Dizideki elemanı temsil edecek bir indeks değeri tanımlıyoruz.  
    int indeks=-1;
```

#region IEnumerator Members

```
public void Reset()  
{  
    indeks=-1;  
}
```

// Güncel IsBilgisi elemanını indeks değerini kullanarak Isler dizisi üzerinden object tipinde geriye döndüren bir özellik. Dikkat ederseniz yalnızca get bloğu var. Bu nedenle foreach döngülerinin sağladığı iterasyon içinde elemanlar üzerinde değişiklik yapamıyoruz.

```
public object Current  
{  
    get  
    {  
        return Isler[indeks];  
    }  
}
```

// İterasyonun devam edip etmemesine karar verebilmek için, şu anki elemandan sonra gelen bir elemanın varlığı tespit edilmelidir. Bunu MoveNext metodu bool tipinden bir değer döndürerek foreach mekanizmasına anlatır. Bizim kontrol etmemiz gereken güncel indeks değerinin Isler isimli dizinin uzunluğundan fazla olup olmadığıdır.

```
public bool MoveNext()  
{
```



```

        if(++indeks>=Isler.Length)
            return false;
        else
            return true;
    }

    #endregion
}
}

```

Şimdi bu elemanları kullanan basit bir console uygulamasını çalıştırdığımızda aşağıdakine benzer bir ekran görüntüsü elde ederiz.



Her ne kadar uygulamamız istediğimiz biçimde çalışsada bizim için bir takım zorluklar vardır. İlki kod yazımının bir desen dahilinde de olsa uzun oluşudur. İkinci zorluk foreach döngüsü içinde kullanılan elemanlar için tip güvenliğinin (type-safety) olmayışdır. Dikkat ederseniz IEnumerator arayüzünün sağladığı Current isimli metodumuz object tipinden elemanları geriye döndürmektedir. Oysaki biz sadece IsBilgisi nesnesi tipinden elemanları geriye döndürecek bir Current metodunu pekala isteyebiliriz. Hatta bu bize tip güvenliğinde sağlayacaktır.

İşte C# 2.0 hem uzun kod satırlarının önüne geçen hem de tip güvenliğini sağlayan yeni bir yapı getirmiştir. Yapının temelinde yine IEnumerable arayüzü yer almaktadır. Yeni generic tipleri sayesinde, iterasyonun bizim belirttiğimiz tiplere yönelik olarak gerçekleştirilebilecek olmasıda garanti altına alınmaktadır. Bu da aradığımız tip güvenliğini bize sağlar. Yukarıda geliştirmiş olduğumu yapıyı C# 2.0' da aşağıdaki şekilde düzenleriz.

```

using System;
using System.Collections.Generic;
using System.Text;

namespace UsingIterators
{
    public class IsListesi:IEnumerable<IsBilgisi>
    {
        static IsBilgisi[] Isler = new IsBilgisi[3];
        private void ListeOlustur()
        {

```

```

        IsBilgisi is1 = new IsBilgisi("Birinci iş", "Burak");
        IsBilgisi is2 = new IsBilgisi("İkinci iş", "Jordan");
        IsBilgisi is3 = new IsBilgisi("Üçüncü iş", "Vader");
        Isler[0] = is1;
        Isler[1] = is2;
        Isler[2] = is3;
    }

    public IListesi()
    {
        ListeOlustur();
    }

    #region IEnumerable<IsBilgisi> Members

    IEnumerator<IsBilgisi> IEnumerable<IsBilgisi>.GetEnumerator()
    {
        yield return Isler[0];
        yield return Isler[1];
        yield return Isler[2];
    }

    #endregion

    #region IEnumerable Members

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        throw new Exception("The method or operation is not implemented.");
    }

    #endregion
}

```

Uygulama kodumuza kısaca bir göz gezdirirsek en büyük yeniliklerden birisinin IEnumerable arayüzünün uygulanışı sırasında ki generic tip tanımlaması olduğunu farkedebiliriz.

```
public class IListesi:IEnumerable<IsBilgisi>
```

Bu tanımlama ile basitçe, IListesi nesnesinin uygulayacağı IEnumerable arayüzünün, IsBilgisi tipinden nesne örnekleri için geçerli olacağı belirtilmektedir. Bu tahmin edebileceğiniz gibi, tip güvenliği dediğimiz ihtiyacı karşılamaktadır (type safety).

Diğer önemli bir değişiklik IEnumerator arayüzünü uygulamış olduğumuz her hangi bir dahili sınıfın (inner class) burada yer almasıdır. Son olarak yield anahtar sözcüğünün kullanımda en büyük yeniliktir. yield anahtar sözcüğü C#

2.0 ile gelen yeni anahtar sözcüklerden birisidir. IEnumerator<T> tipinden nesne örneğini geri döndüren metodumuz içerisinde kullanılmaktadır.

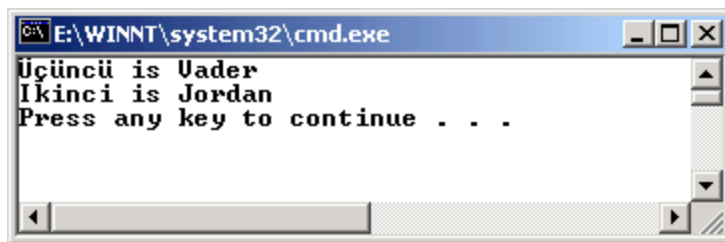
```
IEnumerator<IsBilgisi> IEnumerable<IsBilgisi>.GetEnumerator()  
{  
    yield return Isler[0];  
    yield return Isler[1];  
    yield return Isler[2];  
}
```

Dikkat ederseniz Isler isimli dizimizde yer alan her eleman için yield return söz dizimi kullanılmıştır. Aslında aynı işlevi aşağıdaki kod ilede karşılayabiliriz. Üstelik bu çok daha profesyonel bir yaklaşımdır. Temel olarak anlamamız gereken yield return' ün ilgili dizi içindeki her bir eleman için kullanılıyor olmasıdır.

```
IEnumerator<IsBilgisi> IEnumerable<IsBilgisi>.GetEnumerator()  
{  
    for (int i = 0; i < Isler.Length; i++)  
    {  
        yield return Isler[i];  
    }  
}
```

Kısacası artık yeni iterasyon modelimizde tek yapmamız gereken, foreach döngüsüne dahil olacak elemanların ilgili numarator metodu içinde yield anahtar sözcüğü kullanılarak geri döndürülmesini sağlamaktır. Örneğin aşağıdaki kod parçasının ekran çıktısını göz önüne alalım.

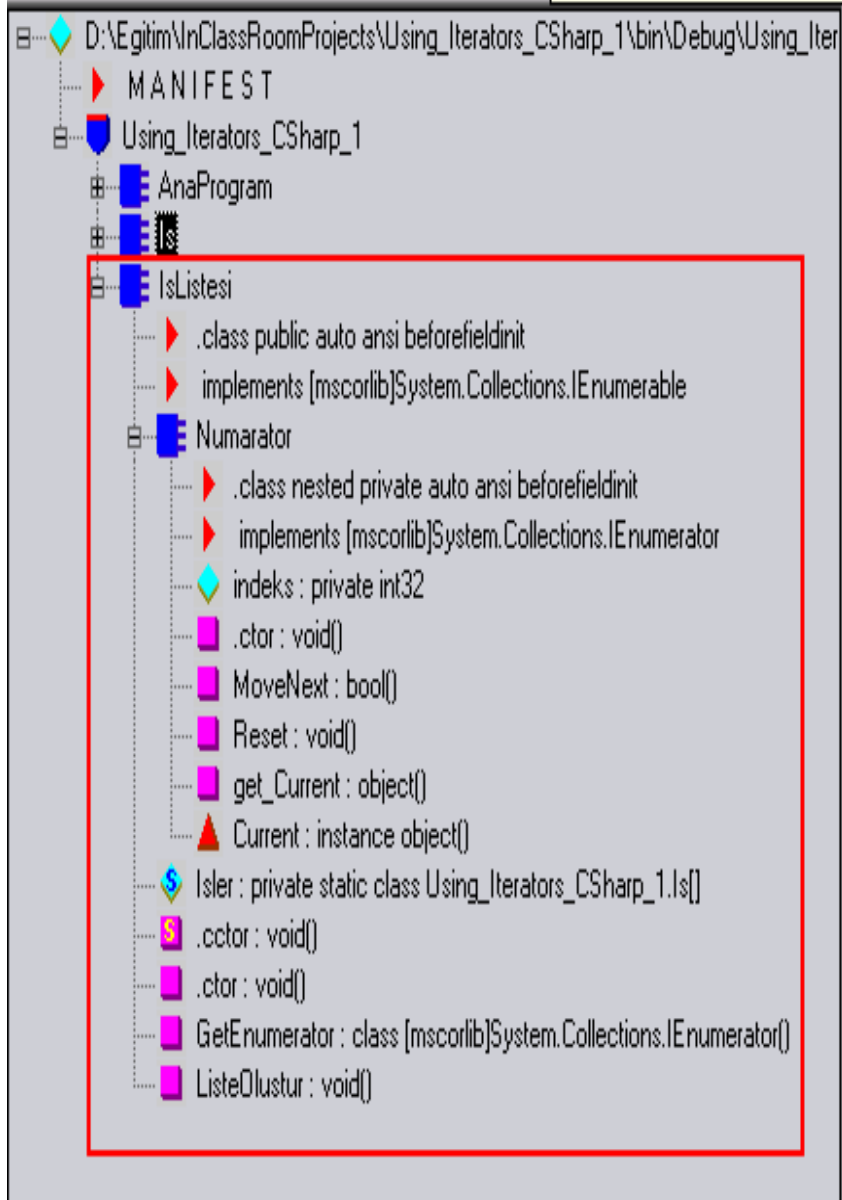
```
IEnumerator<IsBilgisi> IEnumerable<IsBilgisi>.GetEnumerator()  
{  
    yield return Isler[2];  
    yield return Isler[1];  
}
```



Dikkat ederseniz foreach iterasyonumuz yield ile hangi elemanları hangi sırada döndürdüysek ona göre çalışmıştır.

Uygulamadaki gelişimi bir de IL kodu açısından düşünmek lazım. Aslında temelde numaralandırıcının kullanılması için gerekli olan IEnumerator bazlı bir inner class yine kullanılmaktadır. Bu sadece kod yazımında yapılmamaktadır. Öyleki, C# 1.1

için geliřtirdiđimiz örneđin IL koduna ilDasm aracı ile bakarsak ařađıdaki ekran görüntüsünü yakalarız.



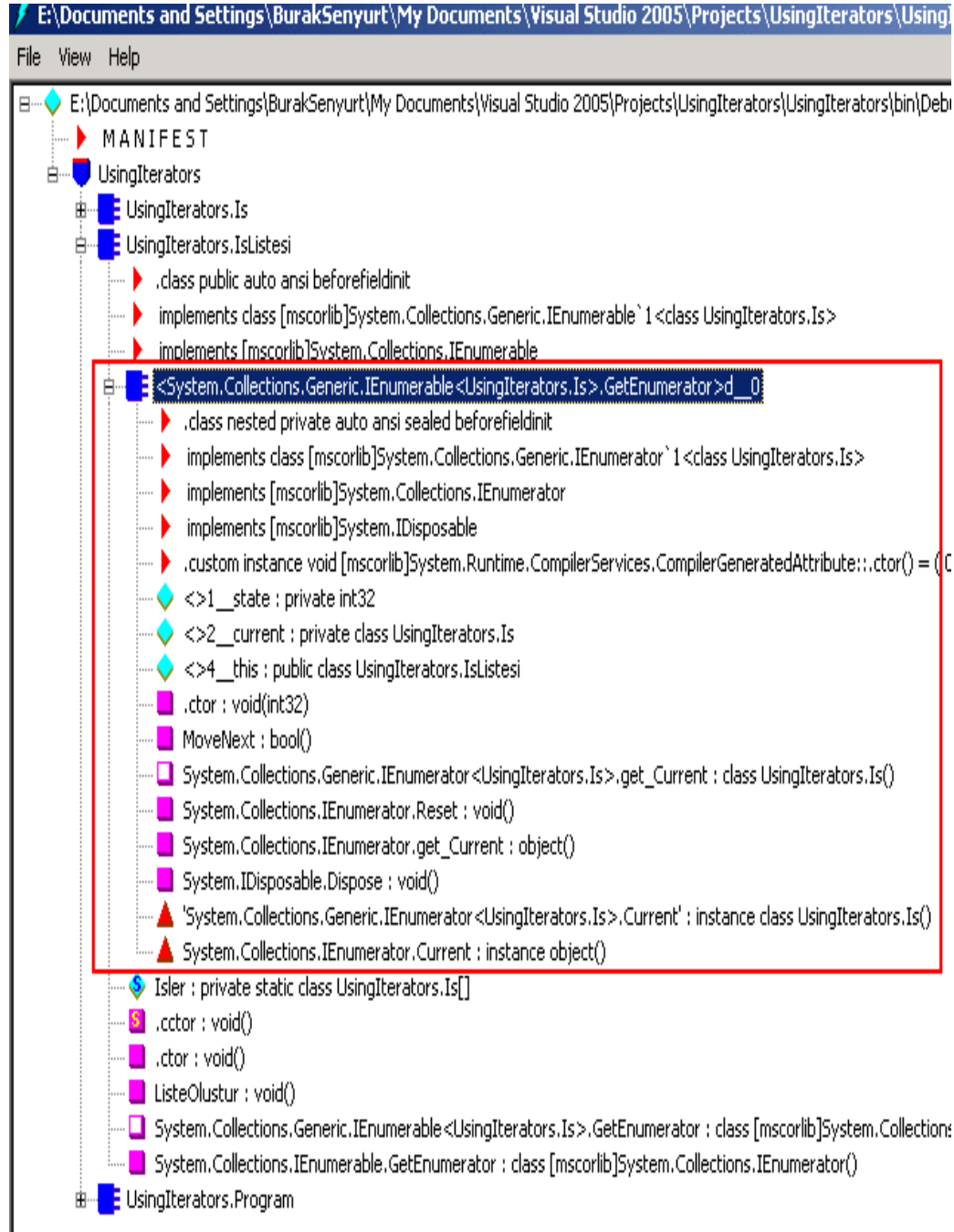
Dikkat ederseniz burada Numarator isimli dahili sınıfımız (inner class) ve ona IEnumerator arayüzü vasıtasıyla uyguladıđımız üyeler görölmektedir. Birde C# 2.0 için geliřtirdiđimiz örneđin IL koduna bir bakalım.



*řu anki beta versiyonuna göre IILDasm aracının fiziki adresi
\\Program Files\\Microsoft Visual Studio
8\\SDK\\v2.0\\Bin\\ildasm.exe dir.*

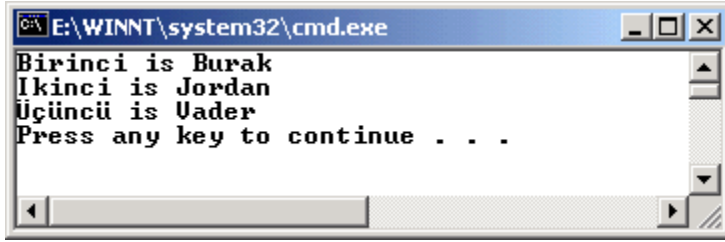
Dikkat ederseniz biz IEnumerator arayüzünü kullanarak bir dahili sınıfı yazmamıř olsakta, IL kodunun detaylarında böyle bir yapının kullanıldıđı görölmektedir.

Tabi burada bir önceki versiyondan farklı olarak, generic tip uyarlamasında mevcuttur.



Artık uygulamamız için aşağıdaki kod parçasını başarılı bir şekilde çalıştırabiliriz.

```
IsListesi isListesi = new IsListesi();
foreach (IsBilgisi i in isListesi)
{
    Console.WriteLine(i.IsAdi + " " + i.IsSorumlu);
}
```



```
E:\WINNT\system32\cmd.exe
Birinci is Burak
Ikinci is Jordan
Uçüncü is Vader
Press any key to continue . . .
```

Gördüğünüz gibi C# 2.0 ile iteratif işlemlerin alt yapısının oluşturulması C# 1.1 ' e göre daha az kod yazarak kolayca gerçekleştirilebiliyor. Böylece geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

Dizilere(Arrays) İlişkin Üç Basit Öneri - 14 Temmuz 2005 Perşembe

C#, arrays,

Değerli Okurlarım Merhabalar,

Hepimiz dizileri (Arrays) bilir ve kullanırız. Her ne kadar günümüzde koleksiyonlar, xml kaynakları ve tablo yapıları veri saklamak amacıyla daha çok kullanılıyor olsalar da, dizilerde yoğun şekilde yararlanmaktayız. Örneğin kendi tasarladığımız bir sınıfa ait nesneler topluluğunu pekala bir dizi şeklinde ifade edebilir hatta serileştirebiliriz (serializable). Lakin dizileri kullanırken tercih edeceğimiz ve bize performans açısından avantaj sağlayacak teknikleri çoğu zaman göz ardı ederiz. İşte bu makalemizde dizileri kullanırken işimize yarayacak performans kriterlerinden bahsedeceğiz.



Öneri 1 ; Dizi elemanları arasında gezinirken kullanacağımız **for döngüleri, foreach döngülerine nazaran** daha süratli çalışır.

İki döngü yapısı arasındaki farkı daha net olarak görebilmek için aşağıdaki basit console uygulamasını göz önüne almakta fayda var. Uygulamamızda basit olarak double tipinden bir dizi tanımlıyoruz. Bu dizinin boyutunu dikkat ederseniz yüksek verdik. Hem for döngüsünü hem de foreach döngüsünü sırasıyla çalıştırıyoruz. İlk olarak for döngüsü ile indeksleyici operatörünü kullanarak dizimizin elemanları üzerinde ilerliyor ve bir toplam değeri alıyoruz. Aynı işlemi daha sonra bir foreach döngüsü ile gerçekleştiriyoruz. (*Aldığımız toplama işleminin bizim için özel bir önemi veya anlamı yok.*) Döngülerin çalışma sürelerini yaklaşık olarak hesaplayabilmek amacıyla da TimeSpan ve DateTime sınıflarından faydalanıyoruz. Bu hesaplamalar bize yaklaşık olarak döngülerin çalışma sürelerini verecektir.

```
using System;
```

```
namespace UsingArrays
```

```
{
```

```
    class DizilerTest
```

```
    {
```

```
        static void DiziTest_1()
```

```
        {
```

```
            double[] dizi=new double[50000000];
```

```
            double toplam=0;
```

```

for(int i=0;i<dizi.Length;i++)
{
    dizi[i]=i;
}

#region for döngüsü ile
DateTime dtBaslangic=DateTime.Now;
for(int i=0;i<dizi.Length;i++)
{
    toplam+=dizi[i];
}
DateTime dtBitis=DateTime.Now;
TimeSpan tsFark=dtBitis-dtBaslangic;
Console.WriteLine(tsFark.TotalSeconds+" saniye...");
#endregion

#region foreach döngüsü ile
toplam=0;
dtBaslangic=DateTime.Now;
foreach(int i in dizi)
{
    toplam+=i;
}
dtBitis=DateTime.Now;
tsFark=dtBitis-dtBaslangic;

Console.WriteLine(tsFark.TotalSeconds+" saniye...");
#endregion

Console.ReadLine();
}

static void Main(string[] args)
{
    DiziTest_1();
}
}

```

Kodumuzu bu haliyle çalıştırdığımızda aşağıdakine benzer bir ekran görüntüsü elde ederiz. Ben kendi test ortamımda (*windows 2000, 512 mb Ram, P4 2.4 ghz cpu tabanlı bir pc*) uygulamayı iki kez farklı zamanlarda çalıştırdığımda aşağıdaki sonuçları aldım. Yaklaşık değerler olmasına rağmen, belirgin derecede fark olduğu hemen gözlemlenmektedir. Tabiki uygulamanın verdiği sonuçlar sistemin o anki işlem yoğunluğuna göre değişiklik gösterebilir. Ancak sonuç olarak for döngüleri söz konusu olan iterasyonu foreach döngülerine göre daha hızlı tamamlamaktadır.

İlk çalıştırılış;


```
C:\ "D:\Egitim\InClassRoomProjects\Console
1,301872 saniye...
3,0744208 saniye...
```

İkinci çalıştırılış;

```
C:\ "D:\Egitim\InClassRoomProjects\O
1,1716965 saniye...
3,0143645 saniye...
```

Elbetteki dizimizdeki eleman sayısının azalması halinde iki döngünün çalışma süreleri arasındaki fark azalacaktır hatta bir birlerine daha da yaklaşacaktır. Örneğin dizimizi 10 milyon elemanlı olarak tanımlarsak uygulamanın çalışmasının sonucu aşağıdaki gibi olacaktır. Döngülerin çalışma süreleri arasındaki fark azalmış olmasına rağmen yine de for döngüsü iterasyonu çok daha önce tamamlamıştır.

İlk çalıştırılış;

```
C:\ "D:\Egitim\InClassRoomProjec...
0,240348 saniye...
0,60087 saniye...
```

İkinci çalıştırılış;

```
C:\ "D:\Egitim\InClassRoomProje
0,2503625 saniye...
0,6108845 saniye...
```

Gelelim diziler ile ilgili ikinci önemli öneriye. Bu öneri, geriye dönüş değerleri dizi olan metodlar ile ilgilidir.



Öneri 2 ; Metodlardan geriye dizi referansları dönüyorsa, dönüş değeri olmayacak koşullu durumlarda null değer yerine 0 elemanlı bir dizinin döndürülmesi uygulamada daha az kontrol yapmamızı sağlar.

Bu öneriyi anlayabilmek için aşağıdaki kod parçasında ye alan basiy console uygulamasını göz önüne alalım.

```

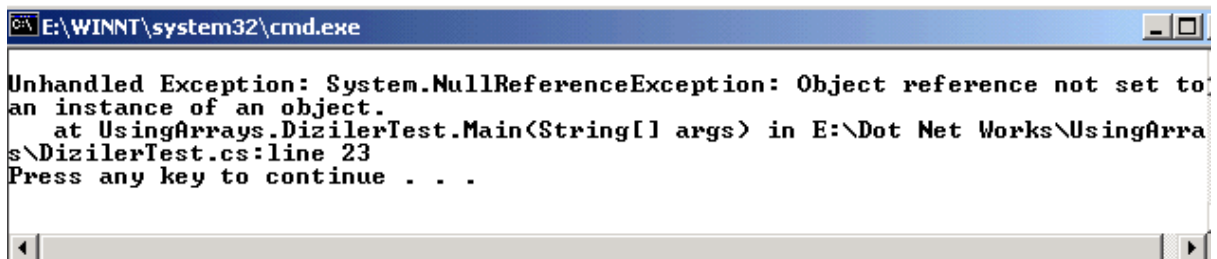
using System;

namespace UsingArrays
{
    class DizilerTest
    {
        static double[] DiziTest_2(int elemanSayisi)
        {
            if (elemanSayisi <= 0)
            {
                return null;
            }
            else
            {
                return new double[elemanSayisi];
            }
        }

        static void Main(string[] args)
        {
            double[] dizi;
            dizi = DiziTest_2(-1);
            for (int i = 0; i < dizi.Length; i++)
            {
                Console.WriteLine(dizi[i]);
            }
        }
    }
}

```

Bu örnekte DiziTest_2 isimli metodumuz diziyi içeride oluşturup referansını geri döndürmek üzere tasarlanmıştır. Eğer bu metoda aktardığımız parametre değerimiz 0 veya negatif ise, metodumuz geriye null değer döndürmektedir. Bu mümkündür çünkü diziler bildiğiniz gibi referans türünden elemanlardır. Ana kodumuzda ise for döngüsü yardımıyla oluşturulan dizinin elemanları ekrana yazdırılmaktadır. Uygulamamızı bu haliyle derlediğimizde aşağıdaki ekran görüntüsünde olduğu gibi çalışma zamanında **NullReferenceException** istisnası ile karşılaşırız. Bu sonuç for döngüsü yerine foreach döngüsünü kullandığımız takdirde de kaçınılmazdır.



Sorunu çözebilmek için döngünün çalıştırılmasından hemen önce dizi referansının null olup olmadığının kontrolünü yapmamız gerekir. Dolayısıyla kodumuzu aşağıdaki gibi düzenlemeliyiz.

```
double[] dizi;  
dizi = DiziTest_2(-1);  
if (dizi != null)  
{  
    foreach (double eleman in dizi)  
    {  
        Console.WriteLine(eleman);  
    }  
}
```

Ancak basit bir teknik ile bu kontrolü atlayabiliriz. Tek yapmamız gereken DiziTest_2 isimli metodumuzda null değer yerine 0 elemanlı bir dizi referansı döndürmek olacaktır. Yani;

```
static double[] DiziTest_2(int elemanSayisi)  
{  
    if (elemanSayisi <= 0)  
    {  
        return new double[0];  
    }  
    else  
    {  
        return new double[elemanSayisi];  
    }  
}
```

Bu tekniğin performans açısından kazanımı olup olmadığı tartışılacak bir konudur. Ancak bizi istisna yakalama ve if kontrolünden kurtarmıştır.

Diziler ile ilgili üçüncü önerimiz ise özellikle iki boyutlu dizileri ilgilendirmektedir.



Öneri 3 ; İki boyutlu dizilerde iç içe kullanılan döngülerde en dıştan içe doğru gerçekleştirilen ötelemelerde, dizinin boyutlarındaki sıraya göre işlem yapılması performansı olumlu yönde etkiler.

Ne kastetmek istediğimizi anlayabilmek için aşağıdaki kod parçasında verilen örneği göz önüne alalım.

```
static void DiziTest(int x)  
{  
    int[,] dizi = new int[x, x];  
    DateTime dtBaslangic, dtBitis;  
    TimeSpan tsFark;
```

```

dtBaslangic = DateTime.Now;
for (int i = 0; i < x; i++)
    for (int j = 0; j < x; j++)
        dizi[i, j] = i + j;
dtBitis = DateTime.Now;
tsFark = dtBitis - dtBaslangic;
Console.WriteLine("Geçen süre...{0} saniye.", tsFark.TotalSeconds);

dtBaslangic = DateTime.Now;
for (int i = 0; i < x; i++)
    for (int j = 0; j < x; j++)
        dizi[j, i] = i + j;
dtBitis = DateTime.Now;
tsFark = dtBitis - dtBaslangic;
Console.WriteLine("Geçen süre...{0} saniye.", tsFark.TotalSeconds);
}

static void Main(string[] args)
{
    for (int testSayisi = 0; testSayisi < 5; testSayisi++)
    {
        DiziTest(6000);
        Console.WriteLine("-----");
    }
}

```

Bu kod parçasında, iki boyutlu bir diziye eleman atamışları için iki adet iç-içe geçmiş döngü kullandığımızı görüyorsunuz. Bu döngüler arasında ki tek fark dizi elemanlarına erişim şekli. İlk döngüde i,j sırası kullanılırken ikinci döngümüzde j,i sırası kullanılıyor. İkinci iç - içe döngümüzde aslında i,j sırasında bir iterasyon yapılmasına rağmen dizi elemanlarına j,i sırasında erişilmekte. Bu hepimizin gözden kaçırabileceği bir nokta. Dalgınlık sonucu dahi bu tarz bir kod satırı yazma ihtimalimiz var. Peki ya bu tarz bir yazımın sonuçları ne olabilir? Döngülerin çalışmalarının sonucu aynı olsa da aşağıdaki şekil aradaki farkı açıkça ortaya koymaktadır. Sistemden siteme süreler fark gösterebilir, ancak iki döngü mekanizmasının arasındaki süre farkı çok açık ve belirgindir.

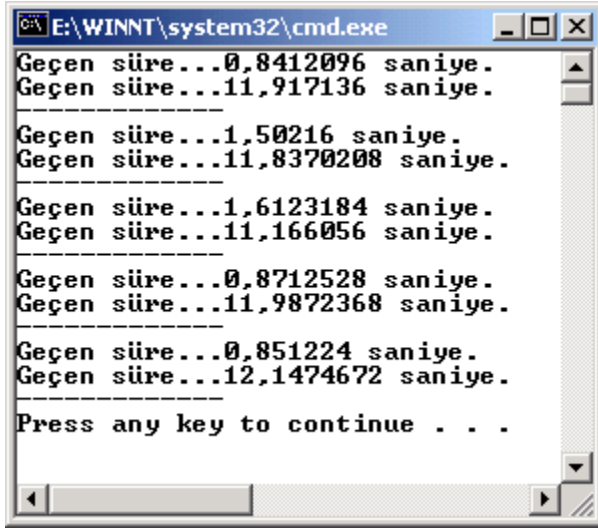
Elbette aynı durum boyutları farklı bir dizi içinde geçerlidir. Örneğimizdeki döngü yapılarını aşağıdaki gibi değiştirelim ve tekrar deneyelim.

```
static void DiziTest(int x,int y)
{
    int[,] dizi = new int[x, y];
    DateTime dtBaslangic, dtBitis;
    TimeSpan tsFark;

    dtBaslangic = DateTime.Now;
    for (int i = 0; i < x; i++)
        for (int j = 0; j < y; j++)
            dizi[i, j] = i + j;
    dtBitis = DateTime.Now;
    tsFark = dtBitis - dtBaslangic;
    Console.WriteLine("Geçen süre...{0} saniye.", tsFark.TotalSeconds);

    dtBaslangic = DateTime.Now;
    for (int i = 0; i < y; i++)
        for (int j = 0; j < x; j++)
            dizi[j, i] = i + j;
    dtBitis = DateTime.Now;
    tsFark = dtBitis - dtBaslangic;
    Console.WriteLine("Geçen süre...{0} saniye.", tsFark.TotalSeconds);
}

static void Main(string[] args)
{
    for (int testSayisi = 0; testSayisi < 5; testSayisi++)
    {
        DiziTest(6000,5000);
        Console.WriteLine("-----");
    }
}
```



```
E:\WINNT\system32\cmd.exe
Geçen süre...0,8412096 saniye.
Geçen süre...11,917136 saniye.
-----
Geçen süre...1,50216 saniye.
Geçen süre...11,8370208 saniye.
-----
Geçen süre...1,6123184 saniye.
Geçen süre...11,166056 saniye.
-----
Geçen süre...0,8712528 saniye.
Geçen süre...11,9872368 saniye.
-----
Geçen süre...0,851224 saniye.
Geçen süre...12,1474672 saniye.
-----
Press any key to continue . . .
```

Bu kez süreler biraz daha uzamış görünüyor. Özellikle ikinci iç-içe döngüde. Burada en büyük etkenlerden birisi x-y sırası yerine y-x sırasını tercih ettiğimizdir. Bunu yapmamızın sebebi tahmin edeceğimiz gibi `IndexOutOfRangeException` istisnasından kurtulmaktır. Bu küçük hileye rağmen dizi boyutlarına ters sırada erişilmeye çalışılması, performansı olumsuz yönde etkiler. Elbetteki geliştirdiğimiz uygulamalarda çoğu zaman bu tip çok boyutlu döngüleri kullanmayabiliriz. O sebepten her iki kullanımda derleme zamanı hatası vermeyeceği gibi düzgün bir biçimde çalışacaktır. Özellikle küçük boyutlu dizilerde bu farklar çok ama çok azdır. Yine de siz siz olun ve profesyonel bir yazılım geliştirici olarak bu öneriyi ve daha öncekilerini dikkate alın. Böylece geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

Tip Güvenli (Type Safety) Koleksiyonlar Oluşturmak - 1 - 23 Temmuz 2005 Cumartesi

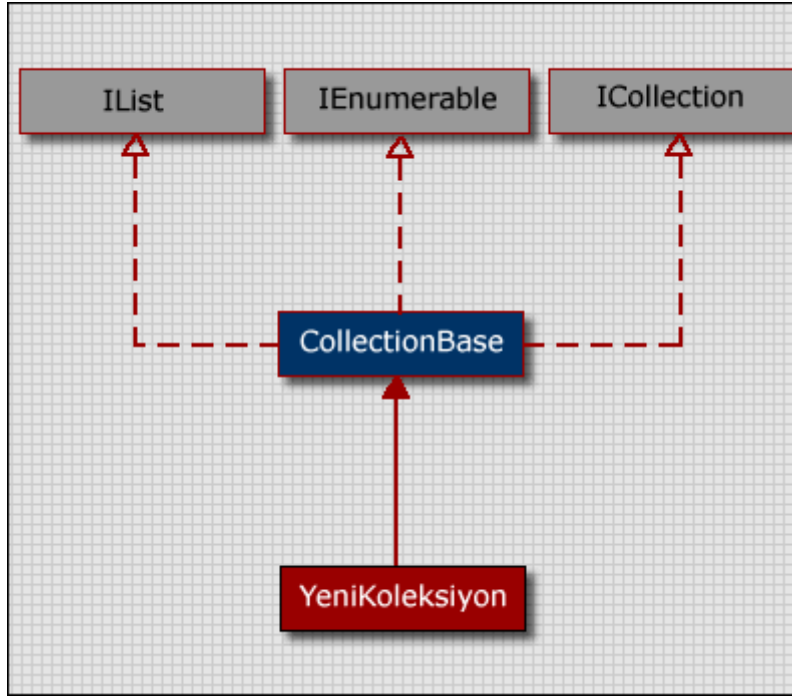
C#, type safety, type safed collections, collections,

Değerli Okurlarım Merhabalar,

Tip güvenliğini sağlamak her zaman için önemli unsurlardan birisidir. Koleksiyon tabanlı nesneleri kullanırken çoğu zaman istediğimiz tip güvenliğini sağlayamayabiliriz. Buradaki en büyük etken, koleksiyon tabanlı nesnelerin object tipinden referanslar taşıyor olmasıdır. Bazen kendi belirlediğimiz tip dışında, başka her hangi bir tip barındırmasına izin vermek istemediğimiz yapıda koleksiyon nesnelere ihtiyacımız olur. Böyle bir koleksiyon nesnesinin en büyük avantajı az önce bahsettiğimiz tip güvenliğini sağlamasıdır.

C# 2.0 versiyonunda koleksiyon nesnelere ilişkin olarak tip güvenliği generic yapıları ile kazandırılmıştır. Peki 1.1 versiyonunda bu işleri nasıl gerçekleştirebiliriz? İki alternatif yolumuz vardır. Bunlardan birisi var olan bir koleksiyon nesnesini türetmektir. Diğer yol ise CollectionBase veya DictionaryBase tipinden türetme yaparak bir koleksiyon nesnesi tanımlamaktır. Biz bu makalemizde CollectionBase sınıfı yardımıyla tip güvenli koleksiyon sınıflarını nasıl oluşturabileceğimizi inceleyeceğiz. Bu sınıflar aynı zamanda strongly-typed collections (kuvvetle türendirilmiş koleksiyonlar) olarakta adlandırılmaktadır.

İlk olarak CollectionBase sınıfını kısacada olsa tanımakta fayda var. CollectionBase sınıfı abstract bir sınıftır. Bu kendisine ait bir nesne örneğinin oluşturulamayacağı anlamına gelmektedir. Ancak CollectionBase sınıfı IList, ICollection ve IEnumerable arayüzlerini implemente etmektedir. Buda tipik olarak bir koleksiyon nesnesi için gerekli üyeleri sağladığını gösterir. Aşağıdaki şekilde görülen YeniKoleksiyon strongly-typed koleksiyon sınıfımızdır. YeniKoleksiyon, CollectionBase sınıfından türetilmiştir. CollectionBase sınıfımız ise bir koleksiyonun taşınması gereken özellikleri sunan üyelerin bildirimini içeren üç temel arayüzden türemektedir.



CollectionBase sınıfı kendi içerisinde IList tipinden bir tip döndüren protected yapıda List isimli bir özelliğe de sahiptir; ki bu özellik sayesinde CollectionBase sınıfından türeteceğimiz sınıflar içerisinde IList tipindeki koleksiyona erişebilir ve doğal olarak ekleme, çıkarma gibi temel koleksiyon işlevselliklerini sağlayabiliriz. List özelliğinin yanı sıra InnerList isimli özellik doğrudan bir ArrayList nesne referansına erişilmesini sağlar. Her iki özellikten de faydalanabilirsiniz. Biz bu makalemizde List özelliği yardımıyla koleksiyon aktivitelerini sağlayacağız. Şimdi gelin kendi strongly-typed koleksiyon sınıfımızı yazalım. Örneğimizde bir Dvd' ye ait bilgileri barındıran nesneler dizisini tutacak şekilde bir koleksiyon tasarlayacağız. İlk olarak bu koleksiyon içerisinde tutmak istediğimiz Dvd nesnelerini temsil edecek sınıfımızı geliştirelim. Dvd sınıfının prototipi aşağıdaki şekilde olduğu gibidir.

Dvd
+ Baslik: string + Fiyat: int + YapimYili: DateTime
+ Dvd(baslik,fiyat,yapimYili) + ToString():string

Dvd sınıfımız

```
using System;
```

```
namespace StrongCollections
{
    public class Dvd
    {
```



```

private string m_Baslik;
private int m_Fiyat;
private DateTime m_YapimYili;

public string Baslik
{
    get{return m_Baslik;}
    set{m_Baslik=value;}
}
public int Fiyat
{
    get{return m_Fiyat;}
    set{m_Fiyat=value;}
}
public DateTime YapimYili
{
    get{return m_YapimYili;}
    set{m_YapimYili=value;}
}
public Dvd(string baslik,int fiyat,DateTime yapimYili)
{
    Baslik=baslik;
    Fiyat=fiyat;
    YapimYili=yapimYili;
}
public Dvd()
{
}
public override string ToString()
{
    return this.Baslik+" "+this.Fiyat+" "+this.YapimYili.ToShortDateString();
}
}
}

```

DvdKoleksiyon sınıfımızı ise yapısı içerisinde bir koleksiyon için ihtiyaç duyulabilecek bir iki metod içerecek şekilde tasarlayacağız. Temel olarak bir Dvd nesnesini koleksiyona ekleme ve çıkarma gibi işlemleri üstlenen metodlara sahip olacak. Ancak önemli bir özellik olarak koleksiyon içerisindeki elemanlara indeksler üzerinden erişebilmemizi sağlayacak bir indeksleyicide yer alacak. Koleksiyon sınıfımızın yapısını aşağıdaki şekilden daha net olarak görebilirsiniz.



DvdKoleksiyon sınıfı

```

using System;
using System.Collections;

namespace StrongCollections
{
    public class DvdKoleksiyon:CollectionBase
    {
        public void Ekle(Dvd dvd)
        {
            this.List.Add(dvd);
        }
        public void Cikart(Dvd dvd)
        {
            this.List.Remove(dvd);
        }
        public Dvd this[int indeks]
        {
            get{return (Dvd)this.List[indeks];}
            set{this.List[indeks]=value;}
        }
        public void Cikart(int indeks)
        {
            this.List.RemoveAt(indeks);
        }
        public void Ekle(int indeks,Dvd dvd)
        {
            this.List.Insert(indeks,dvd);
        }
        public DvdKoleksiyon()
        {

```

```
}  
}  
}
```

Ekle isimli metodumuzun iki versiyonu vardır. Bunlardan birisi koleksiyonun sonuna bir Dvd nesnesini eklerken diğeri, parametre olarak belirtilen indekse ekleme işlemini gerçekleştirir. Cıkart isimli metodumuzda iki versiyona sahiptir. Bunlardan birisi koleksiyondan parametre olarak aldığı Dvd nesnesini çıkartırken, ikincisi belirtilen indeks üzerindeki Dvd nesnesini koleksiyondan çıkartmaktadır. Indeksleyicimiz ise, DvdKoleksiyonu içerisindeki Dvd tipinden nesnelere indeks değerleri üzerinden erişmemizi sağlar.

Dikkat ederseniz tüm bu üyeler sadece ve sadece Dvd tipinden nesneler üzerinden iş yapmaktadır. Örneğin koleksiyona nesne ekleme ve koleksiyondan nesne çıkartmak için kullandığımız metodlar parametre olarak mutlaka bir Dvd nesne örneğini referans ederler. Benzer şekilde indeksleyicimizde sadece Dvd tipinden nesneler üzerinden çalışır. Şimdi yazdığımız koleksiyon sınıfımızı deneyeceğimiz bir uygulama geliştirelim.

```
using System;  
using System.Collections;  
  
namespace StrongCollections  
{  
    class Uygulama  
    {  
        static DvdKoleksiyon dvdCol;  
  
        static void Listele()  
        {  
            foreach(Dvd dvd in dvdCol)  
            {  
                Console.WriteLine(dvd.ToString());  
            }  
            Console.WriteLine("-----");  
        }  
  
        static void KoleksiyonOlustur()  
        {  
            dvdCol.Ekle(new Dvd("Gladiator",10,new DateTime(2000,1,1)));  
            dvdCol.Ekle(new Dvd("Star Wars 3",20,new DateTime(2005,1,4)));  
            dvdCol.Ekle(new Dvd("Crow",15,new DateTime(1997,3,9)));  
        }  
  
        static void Main(string[] args)  
        {  
            dvdCol=new DvdKoleksiyon();  
            KoleksiyonOlustur();  
        }  
    }  
}
```

```

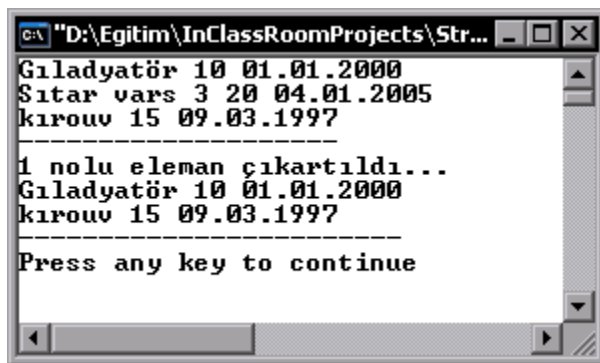
for(int i=0;i<dvdCol.Count;i++)
{
    Console.WriteLine(dvdCol[i].ToString());
}
Console.WriteLine("-----");
dvdCol.Cikart(1);

Console.WriteLine("1 nolu eleman çıkartıldı...");
Listele();

// dvdCol.Ekle(123); // type - safety sağlanır.
}
}
}

```

Uygulamayı çalıştırdığımızda aşağıdaki sonucu elde ederiz.



Aslında sanki normal bir koleksiyondan, örneğin bir ArrayList ile yaptığımız işlemlerden pek bir farkı yokmuş gibi görünüyor. Farkı anlayabilmek için yorum satırımızı koda katarak uygulamayı yeniden derleyelim.

```
26 static void Main(string[] args)
27 {
28     dvdCol=new DvdKoleksiyon();
29     KoleksiyonOlustur();
30
31     for(int i=0;i<dvdCol.Count;i++)
32     {
33         Console.WriteLine(dvdCol[i].ToString());
34     }
35     Console.WriteLine("-----");
36     dvdCol.Cikart(1);
37
38     Console.WriteLine("1 nolu eleman çıkartıldı...");
39     Listele();
40
41     dvdCol.Ekle(123); // type - safety sağlanır.
42 }
```

Task List - 2 Build Error tasks shown (filtered)

!	Description
	Click here to add a new task
!	Argument '1': cannot convert from 'int' to 'StrongCollections.Dvd'

Burada problem Ekle metoduna sayısal bir değerin parametre olarak girilmeye çalışılmasıdır. DvdKoleksiyon sınıfımızın Ekle metodlarını kısaca bir hatırlarsak;

```
public void Ekle(Dvd dvd)
{
    this.List.Add(dvd);
}
public void Ekle(int indeks,Dvd dvd)
{
    this.List.Insert(indeks,dvd);
}
```

parametre olarak sadece Dvd tipinden nesneleri aldıklarını görebiliriz. Bu nedenle kod derleme zamanında hata vererek geliştiricinin çalışma zamanında bir hataya düşmesini engellemiştir. İşte bu tip güvenliğini sağlar. Benzer şekilde koleksiyondan okuduğumuz bir Dvd nesnesini farklı bir nesne tipine de atayamayız. Örneğin aşağıdaki kod parçasını göz önüne alalım.

```
Kitap ktp=new Kitap();
ktp=(Kitap)dvdCol[0];
```

```
double dbl=(double)dvdCol[0];
```

Bu durumda derleme zamanında aşağıdaki hata mesajlarını alırız.

```

29 KoleksiyonOlustur();
30
31 for(int i=0;i<dvdCol.Count;i++)
32 {
33     Console.WriteLine(dvdCol[i].ToString());
34 }
35 Console.WriteLine("-----");
36 dvdCol.Cikart(1);
37
38 Console.WriteLine("1 nolu eleman çıkartıldı...");
39 Listele();
40
41 Kitap ktp=new Kitap();
42 ktp=(Kitap)dvdCol[0];
43
44 double dbl=(double)dvdCol[0];
45

```

Task List - 2 Build Error tasks shown (filtered)

!	Description
	Click here to add a new task
!	Cannot convert type 'StrongCollections.Dvd' to 'double'
!	Cannot convert type 'StrongCollections.Dvd' to 'StrongCollections.Kitap'


Burada var olan bir tipe ve bizim yazdığımız tipe atamalar yapılmaya çalışılmıştır. Ancak koleksiyonumuz sadece Dvd tipinden nesne örneklerini geriye döndüren bir indeksleyiciye sahiptir. Bunu indeksleyiciyi kullanırken de görebilirsiniz.

```

double dbl=(double)dvdCol[0];
StrongCollections.Dvd DvdKoleksiyon [int indeks]

```

Oysaki, bir ArrayList göz önüne alındığında geriye dönen değer her zaman object tipinden olacaktır.

	<p><i>CollectionBase tipinden türettiğimiz nesnelerde, koleksiyonda işlenecek olan nesne tipi ne ise onu kullanmalıyız. Bu tip güvenliğini (type - safety) sağlayabilmemizi olanaklı kılar.</i></p>
---	--

CollectionBase' den türettiğimiz sınıfların sağladığı tip güvenliği daha iyi anlayabilmek için Dvd nesnelerini taşıyacak bir ArrayList koleksiyonunun kullanıldığı aşağıdaki örneği göz önüne almakta fayda var.

```
ArrayList alDvd=new ArrayList();
```

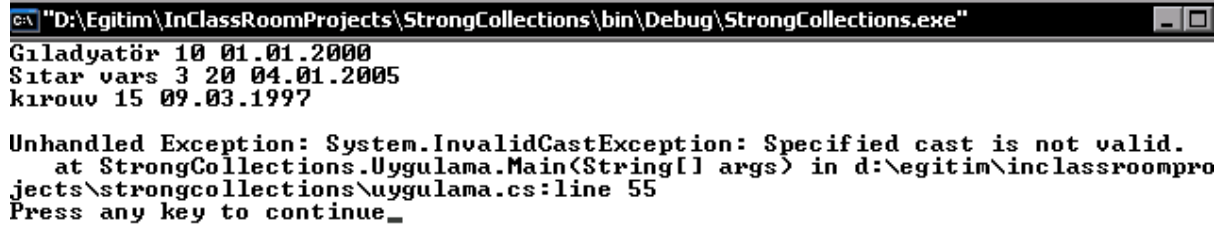
```

alDvd.Add(new Dvd("Gladyatör",10,new DateTime(2000,1,1)));
alDvd.Add(new Dvd("Sitar vars 3",20,new DateTime(2005,1,4)));
alDvd.Add(new Dvd("Kırouv",15,new DateTime(1997,3,9)));
alDvd.Add(12);

```

```
foreach(Dvd dvd in alDvd)
{
    Console.WriteLine(dvd.ToString());
}
```

Yazılan kod son derece masumane görünmektedir. Üstelik derleme zamanında hiç bir hata mesajı vermez. Yani çalışır bir koddur. Ancak uygulamayı bu haliyle çalıştırdığımızda aşağıdaki hata mesajını alırız.




```
C:\> "D:\Egitim\InClassRoomProjects\StrongCollections\bin\Debug\StrongCollections.exe"
Gıladıatör 10 01.01.2000
Sıtar vars 3 20 04.01.2005
kırouv 15 09.03.1997
Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
at StrongCollections.Uygulama.Main(String[] args) in d:\egitim\inclassroompro
jects\strongcollections\uygulama.cs:line 55
Press any key to continue_
```

Sorun foreach döngüsünde açıkça görülmektedir. foreach döngüsü sadece Dvd tipinden elemanlar üzerinde bir öteleme gerçekleştirmek isterken koleksiyonun sonuna eklenen sayısal değer bu durumu bozmaktadır. Eğer foreach döngüsünü terk edip aşağıdaki gibi bir for döngüsünü tercih ederseniz durum biraz daha ilginç bir hal alacaktır.

```
for(int i=0;i<alDvd.Count;i++)
{
    Console.WriteLine(alDvd[i].ToString());
}
```

Kodda herhangi bir derleme zamanı hatası alınmaz. Çalışma zamanında da bir hata alınmaz. Uygulama başarılı bir şekilde çalışır. Ancak bu kez de programın mantıksal bütünlüğü bozulmuştur.



```
C:\> "D:\Egitim\InClassRoomProjects\StrongCollections\bin\D..."
Gıladıatör 10 01.01.2000
Sıtar vars 3 20 04.01.2005
kırouv 15 09.03.1997
12
Press any key to continue
```

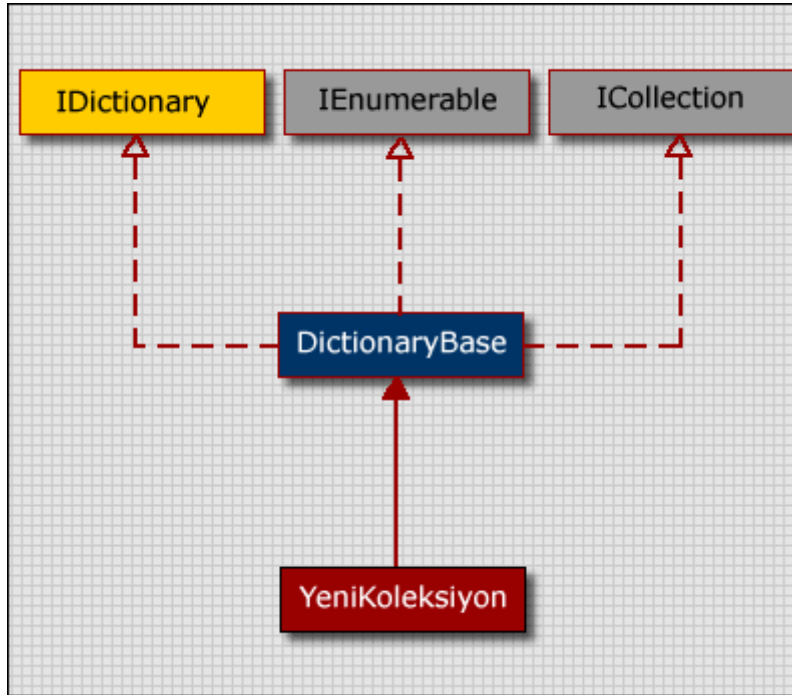
İşte bu tip bir durumla karşılaştığımızda geliştirici olarak tip güvenliğini ve uygulamanın mantıksal bütünlüğünü korumak amacıyla kendi koleksiyon sınıflarımızı kullanmayı tercih ederiz. Yazımızın başında hatırlarsanız kendi yazacağımız koleksiyon sınıflarını `CollectionBase` yoluyla veya `DictionaryBase` yoluyla geliştirebildiğimizi söylemiştik. Bu makalemizde `CollectionBase` sınıfı ile bu işi nasıl yapacağımızı gördük. Bir sonraki makalemizde ise `DictionaryBase` sınıfını inceleyemeye çalışacağız. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

Tip Güvenli (Type Safety) Koleksiyonlar Oluşturmak - 2 - 31 Temmuz 2005 Pazar

C#, strongly typed collections, collections,

Değerli Okurlarım Merhabalar,

Bir önceki makalemizde tip güvenli koleksiyon nesnelerimizi CollectionBase sınıfı yardımıyla nasıl oluşturabileceğimizi incelemiştik. CollectionBase bize ArrayList benzeri koleksiyon sınıflarını yazma fırsatı vermektedir. Diğer yandan Hashtable koleksiyonunda olduğu gibi key (anahtar) - value (değer) çiftlerinden oluşacak tip güvenli bir koleksiyon sınıfı yazmak isteyebiliriz. Bu durumda, DictionaryBase sınıfından yararlanabiliriz. DictionaryBase sınıfı CollectionBase sınıfı gibi abstract yapıdadır. Yani kendisini örnekleyemeyiz. Temel olarak DictionaryBase key-value çiftlerine sahip bir koleksiyonun kullanması gereken üyeleri sunan arayüzlerden türemiştir. Yani IDictionary, IEnumerable ve ICollection arayüzlerini uyarlamaktadır. Dikkat ederseniz CollectionBase sınıfında türediği IEnumerable ve ICollection arayüzleri DictionaryBase içinde söz konusudur.



CollectionBase sınıfı gerekli fonksiyonelliği sağlamak için nasıl ki bir ArrayList koleksiyonunu çevreliyorsa (encapsulate), DictionaryBase sınıfı da bir Hashtable koleksiyonunu çevreler. DictionaryBase sınıfının protected özellikleri (Dictionary, InnerHashtable) yardımıyla oluşturduğumuz koleksiyon içindeki key-value çiftlerine erişilebilir ve eleman ekleme, silme, arama vb. gibi pek çok var olan aksiyonu gerçekleştirebiliriz. Örneğimizi incelediğimizde DictionaryBase yardımıyla tip güvenli bir koleksiyon nesnesi oluşturmanın, CollectionBase

kullanıldığında gerçekleştirilen uyarılama ile neredeyse aynı olduğunu göreceksiniz. Tek fark, içeride erişilen nesnenin bir Dictionary nesnesi olması ve key-value çiftlerinin söz konusu olmasıdır.



DictionaryBase sınıfı key-value (anahtar - değer) çiftlerinin kullanıldığı Hashtable tipi koleksiyon sınıflarını yazmamızı ve kendi tip güvenliğimizi oluşturabilmemizi sağlar.

Şimdi gelin bir örnek üzerinden bu konuyu incelemeye çalışalım. Bu sefer anahtar-değer çiftleri yapısına uygun bir koleksiyon nesnesi olarak ISBN numarasına sahip Kitapları göz önüne alacağız. ISBN numaraları bizim için key (anahtar) olacak. Bunun karşılığında ise bir Kitap nesnesini value(değer) olarak tutacağız. Bu elbette tipik olarak bir Hashtable koleksiyonu ile de yapılabilir. Ancak bizim amacımız anahtarların(keys) mutlaka integer ve değerlerin (values) mutlaka Kitap tipinden nesne örnekleri olmasıdır. Normal bir Hashtable key-value çiftlerini object tipinden tuttuğu için, herhangi bir tipi bu çiftlere atayabiliriz. İşte burada tip güvenliği bizim esas olan amacımız olmaktadır. Bu nedenle DictionaryBase sınıfı yardımıyla kendi koleksiyon sınıfımızı yazacağız. İlk olarak Kitap nesnemize ait sınıfımızı oluşturalım. Sınıfımızın basit yapısı aşağıdaki gibidir.

Kitap
+ ISBN:int + Baslik:string + Fiyat:int
+ Kitap(isbn,baslik,fiyat) + ToString():string

Sınıfımız kodları;

using System;

namespace StrongCollections

{

public class Kitap

{

private int m_ISBN;

private string m_Baslik;

private int m_Fiyat;

public int ISBN

{

get{return m_ISBN;}

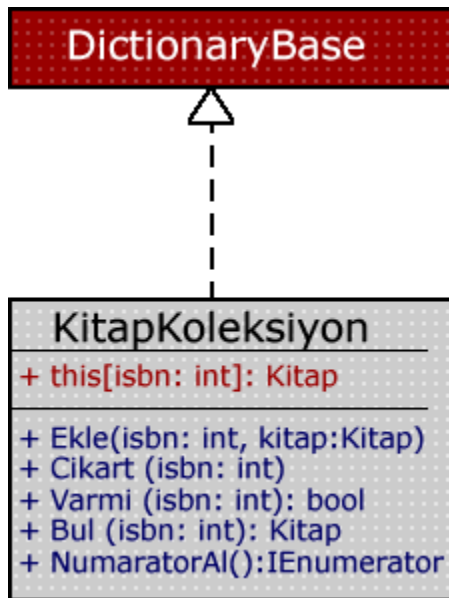
set{m_ISBN=value;}

```

    }
    public string Baslik
    {
        get{return m_Baslik;}
        set{m_Baslik=value;}
    }
    public int Fiyat
    {
        get{return m_Fiyat;}
        set{m_Fiyat=value;}
    }
    public Kitap(int isbn,string baslik,int fiyat)
    {
        m_ISBN=isbn;
        m_Baslik=baslik;
        m_Fiyat=fiyat;
    }
    public override string ToString()
    {
        return m_ISBN.ToString()+" "+m_Baslik+" "+m_Fiyat.ToString();
    }
    public Kitap()
    {
    }
}
}

```

Şimdi kendi tip güvenli koleksiyon sınıfımızı yazalım. Sınıfımızı DictionaryBase' den türettikten sonra içeriden Dictionary özelliğine erişerek eleman ekleme, çıkarma, arama, bulma gibi işlemleri yapacağımız metodlar ile bir indeksleyici ve anahtarlar(keys) üzerinde öteleme yapabileceğimiz bir numarator metod ekleyeceğiz. Basit olarak sınıfımızın içeriği aşağıdaki gibi olacak.



Sınıf kodlarımız;

```

using System;
using System.Collections;

namespace StrongCollections
{
    public class KitapKoleksiyon:DictionaryBase
    {
        public void Ekle(int isbn,Kitap kitap)
        {
            Dictionary.Add(isbn,kitap);
        }
        public void Cikart(int isbn)
        {
            Dictionary.Remove(isbn);
        }
        public bool Varmi(int isbn)
        {
            return Dictionary.Contains(isbn);
        }
        public Kitap Bul(int isbn)
        {
            return (Kitap)Dictionary[isbn];
        }
        public Kitap this[int isbn]
        {
            get{return (Kitap)Dictionary[isbn];}
            set{Dictionary[isbn]=value;}
        }
        public IEnumerator NumaratorAl()
    }
}
  
```

```

    {
        return Dictionary.Keys.GetEnumerator();
    }
    public KitapKoleksiyon()
    {
    }
}

```

Ekle metodumuz iki parametre almaktadır. Tahmin edeceğimiz gibi ilk parametremiz Hashtable koleksiyonu için gerekli key (anahtar), ikinci parametremiz ise value(değer) dir. Bu metod sayesinde, belli bir isbn numarasının karşılığı olarak bir Kitap nesne örneğini koleksiyonumuza eklemiş oluruz. Cıkart metodu, parametre olarak verilen isbn değerini Dictionary içerisinde bulur ve listeden çıkartır. Varmi metodumuz geriye bool tipinden bir değer döndürür. Amacı belirtilen isbn değerinin koleksiyonda yer alıp almadığını belirlemektir. Bunun için Dictionary özelliği üzerinden Contains metodunu çağırırız.

Bul metodumuz ise, parametre olarak girilen isbn' i Dictionary üzerinde arar ve sonucu geriye bir Kitap nesne örneği şeklinde döndürür. Bu dönüştürme gereklidir çünkü Dictionary geriye object tipinden bir nesne örneğini döndürecektir. Indeksleyicimizin indeks değerleri bu sefer key (anahtar) lardır. Dikkat ederseniz tüm metodlarımız int-Kitap tipinden anahtar-değer çiftlerini kullanmaktadır. Böylece aslında tip güvenliğinde sağlamış oluyoruz. Son olarak eklediğimiz NumaratorAl isimli metodumuz Dictionary üzerinde Key değerleri için bir IEnumerator nesne örneğini geriye döndürüyor. Bunu çalışma zamanında koleksiyonumuz içindeki anahtarlar üzerinden öteleme yaparak Kitap nesnelerini elde etmek için kullanabiliriz. Kısacası bir listemele işlemi için kullanabiliriz. Şimdi koleksiyonumuzu kullanacağımız bir örnek uygulama yazalım.

```

using System;
using System.Collections;

namespace StrongCollections
{
    class Uygulama
    {
        static KitapKoleksiyon kitapCol;

        static void Listele()
        {
            IEnumerator numarator=kitapCol.NumaratorAl();
            while(numarator.MoveNext())
            {
                Console.WriteLine(kitapCol[Convert.ToInt32(numarator.Current)].ToString());
            }
        }
    }
}

```

```

static void KoleksiyonOlustur()
{
    kitapCol.Ekle(1000,new Kitap(1000,"Her Yönüyle C#",1));
    kitapCol.Ekle(1001,new Kitap(1001,"Thinking in C#",1));
    kitapCol.Ekle(1002,new Kitap(1002,"Truva",1));
    kitapCol.Ekle(1003,new Kitap(1003,"Java in a Nuthshell",1));
}

static void Main(string[] args)
{
    // KitapKoleksiyon nesne örneğimizi oluşturuyoruz.
    kitapCol=new KitapKoleksiyon();
    // Koleksiyonumuza bir kaç örnek Kitap elemanını ekliyoruz.( key-value çifti olarak)
    KoleksiyonOlustur();
    // Koleksiyonumuzdaki elemanları listeliyoruz.
    Listele();
    Console.WriteLine("-----");
    // Koleksiyonda belirtilen isbn değerine sahip Kitap nesnesi olup olmadığına
    bakıyoruz. (Sonuçlar bool tipinden)
    Console.WriteLine("ISBN: 1000 var mı? {0}",kitapCol.Varmi(1000));
    Console.WriteLine("ISBN: 9999 var mı? {0}",kitapCol.Varmi(9999));
    Console.WriteLine("-----");
    // Koleksiyonumuzdan key değeri 1001 olan key-value çiftini çıkartıyoruz.
    kitapCol.Cikart(1001);
    Console.WriteLine("ISBN: 1001 çıktı");
    // Koleksiyonumuzdaki elemanların son halini listeliyoruz. (Artık 1001 numaraları
    eleman yok)
    Listele();
    Console.WriteLine("-----");
    // Koleksiyonumuzda 1003 isbn değerine sahip key-value çiftini buluyoruz.
    Kitap bulunanKitap=kitapCol.Bul(1003);
    // Bulunan Kitap nesnesinin içeriğini override ettiğimiz ToString metodu ile ekrana
    yazdırıyoruz.
    Console.WriteLine(bulunanKitap.ToString());
}
}
}

```

Uygulamamızı çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.

```
C:\D:\Egitim\InClassRoomProjects\S...
1000 Her Yönüyle C# 1
1003 Java in a Nuthshell 1
1002 Truva 1
1001 Thinking in C# 1
-----
ISBN: 1000 var mı? True
ISBN: 9999 var mı? False
-----
ISBN: 1001 çıktı
1000 Her Yönüyle C# 1
1003 Java in a Nuthshell 1
1002 Truva 1
-----
1003 Java in a Nuthshell 1
Press any key to continue
```

Görüldüğü gibi KitapKoleksiyon isimli koleksiyonumuza eleman eklemek istediğimizde bizden bir key-value çifti beklenmektedir. Öyleki key integer tipinde, value ise Kitap tipinde olmak zorundadır. Bunu tasarım zamanında koleksiyonumuza eleman eklerken kolayca görebiliriz.

```
static void KoleksiyonOlustur()
{
    kitapCol.Ekle(
        void KitapKoleksiyon.Ekle (int isbn, StrongCollections.Kitap kitap) C#, 1));
    kitapCol.Ekle(1001,new Kitap(1001,"Thinking in C#",1));
    kitapCol.Ekle(1002,new Kitap(1002,"Truva",1));
    kitapCol.Ekle(1003,new Kitap(1003,"Java in a Nuthshell",1));
}
```

İşte bu bizim için tip güvenliğini sağlamaktadır. Çünkü KitapKoleksiyon tipinden nesne örneğimize int-Kitap tipi dışında bir anahtar-değer çifti ekleyemiz. Böylece geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

SortedList ve Hashtable İçin 2 Basit Öneri - 06 Ağustos 2005 Cumartesi

C#, sorted list, hashtable,

Değerli Okurlarım Merhabalar,

SortedList ve Hashtable koleksiyonları, anahtar-değer (key-value) çiftlerini esas alır. Hashtable koleksiyonu özellikle sahip olduğu elemanlar ile ilgili işlemlerde kullandığı hash algoritmali teknik sayesinde en hızlı çalışan koleksiyon olma özelliğininide gösterir. Diğer yandan SortedList anahtar-değer çiftlerinin, anahtar değerine göre her zaman sıralandığı bir koleksiyon tipidir.

Yani SortedList koleksiyonuna eklediğimiz elemanların sırasına bakılmaksızın, yeniden yapılan bir sıralama söz konusudur. Bu avantajlı bir durum olsa bile, özellikle SortedList' in çok daha yavaş çalışan bir koleksiyon olmasına neden olmaktadır. Her iki koleksiyon hakkında söylenebilecek pek çok konu vardır. Biz bu makalemizde özellikle dikkat etmemiz gereken 2 teori üzerinde duracağız. İlk teorimiz ile işe başlayalım.



*Bir SortedList oluştururken **doğrudan eleman eklemek yerine**, elemanları önce bir Hashtable koleksiyonuna ekleyip, **SortedList' i bu Hashtable üzerinden oluşturmak** daha hızlıdır.*

Kulağa biraz galip geliyor değil mi? Bir SortedList koleksiyonunu anahtar-değer çiftleri ile doldururken doğrudan SortedList' i kullanmak yerine bir Hashtable' in kullanılması... Her ne kadar ilginç gibi görünsede aşağıdaki basit örnek ile bu durumu analiz edebiliriz.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace UsingSortedList
{
    class Tester
    {
        private SortedList sl;
        private SqlConnection con;
        private SqlCommand cmd;
        private SqlDataReader dr;

        private void Hazirla()
        {
            con = new SqlConnection("data
source=BURAKS;database=AdventureWorks;integrated security=SSPI");
            cmd = new SqlCommand("SELECT NationalIDNumber,Title From
HumanResources.Employee", con);
        }

        public void Olustur()
        {
            sl = new SortedList();
            con.Open();
            dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
            while (dr.Read())
```

```

        {
            sl[dr["NationalIDNumber"]] = dr["Title"];
        }
        dr.Close();
    }

    public void OlusturHt()
    {
        Hashtable ht = new Hashtable();
        con.Open();
        dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
        while (dr.Read())
        {
            ht[dr["NationalIDNumber"]] = dr["Title"];
        }
        sl = new SortedList(ht);
        dr.Close();
    }

    public Tester()
    {
        Hazirla();
    }
}

class Program
{
    static void Main(string[] args)
    {
        Tester tester = new Tester();
        DateTime dtBaslangic, dtBitis;
        TimeSpan ts;

        #region SortedList ile

        dtBaslangic = DateTime.Now;
        tester.Olustur();
        dtBitis = DateTime.Now;
        ts = dtBitis - dtBaslangic;
        Console.WriteLine(ts.TotalMilliseconds);

        #endregion

        #region Hashtable ile

        dtBaslangic = DateTime.Now;
        tester.OlusturHt();
        dtBitis = DateTime.Now;
    }
}

```



```

ts = dtBitis - dtBaslangic;
Console.WriteLine(ts.TotalMilliseconds);

#endregion
}
}
}

```

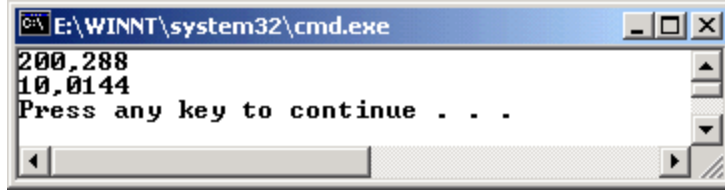
Örneğimizde test işlemlerimiz için Tester isimli bir sınıf kullanıyoruz. Diğer teorimiz için de bu sınıfı kullanacağız. Sınıfımızda Sql Server 2005 üzerinde yer alan AdventureWorks isimli veritabanını kullanacağız. Burada Employee isimli tabloyu göz önüne alacağız. Amacımız bir Employee' un NationalIDNumber alanlarını anahtar olarak, Title alanlarını ise değer olarak SortedList koleksiyonumuza eklemek. Tablonun select edilen içeriğine bakalım olursak NationalIDNumber alanlarının düzensiz(unsorted) sırada olduğunu görürüz.

Employee (HumanResources.Employee) - Start Page - Program.cs						
	EmployeeID	NationalIDNumber	ContactID	LoginID	ManagerID	Title
1		14417807	1209	adventure-work...	16	Production Tech...
2		253022876	1030	adventure-work...	6	Marketing Assist...
3		509647174	1002	adventure-work...	12	Engineering Man...
4		112457891	1290	adventure-work...	3	Senior Tool Desi...
5		480168528	1009	adventure-work...	263	Tool Designer
6		24756624	1028	adventure-work...	109	Marketing Manager
7		309738752	1070	adventure-work...	21	Production Supe...
8		690627818	1071	adventure-work...	185	Production Tech...
9		695256908	1005	adventure-work...	3	Design Engineer
10		912265825	1076	adventure-work...	185	Production Tech...
11		998320692	1006	adventure-work...	3	Design Engineer
12		245797967	1001	adventure-work...	109	Vice President of...
13		844973625	1072	adventure-work...	185	Production Tech...
14		233069302	1067	adventure-work...	21	Production Supe...
15		132674823	1073	adventure-work...	185	Production Tech...
16		446466105	1068	adventure-work...	adventure-works\jeffrey...	Production Supe...
17		565090917	1074	adventure-work...	185	Production Tech...
18		494170342	1069	adventure-work...	21	Production Supe...
19		9659517	1075	adventure-work...	185	Production Tech...

Elbetteki burada **basit bir Order By ile** NationalIDNumber alanına göre sıralama yaptırabiliriz. Ancak SortedList ile ilgili teorimize bakmak için bize sırasız(unsorted) ve benzersiz(unique) anahtar(key) değerleri gerekiyor. Hazır elimizde var iken kullanmakta fayda var. İlk metodumuz (Olustur() metodu) anahtar-değer çiftlerini SortedList' e doğrudan ekliyor. İkinci metodumuz (Olusturht() metodu) ise anahtar-değer çiftlerini önce bir Hashtable koleksiyonuna ekliyor ve daha sonra SortedList koleksiyonunu aşağıdaki yapıcı metod prototipi ile oluşturuyor.

```
public SortedList(IDictionary d);
```

Hashtable koleksiyonu IDictionary arayüzünü implemente ettiği için, SortedList' imizi bu şekilde oluşturabilmemiz son derece doğaldır. Uygulamayı çalıştırdığımızda aşağıdakine benzer bir sonuç elde ederiz.



Aslında sonuçlar milisaniye cinsinden olduğu için çok önemsiz görünebilir. Kaldı ki uygulamanın kısa süreli sonraki çalıştırılışlarında veritabanı kaynaklarının yeniden kullanımında etkisiyle bu süre dahada aşağılara inecektir. Ancak gerçek hayat problemlerinde çok daha fazla satıra sahip (çoğunlukla buradaki gibi 290 satırlık bir veri seti değil) tablolarda benzer işlemleri kullanabiliriz. Sonuç itibarıyla teorik olarak bir SortedList koleksiyonunu oluştururken bir Hashtable koleksiyonundan yararlanmak performansı olumlu yönde etkilemektedir. Gelelim ikinci dikkate değer teoriye.



*İster SortedList ister Hashtable olsun, anahtar-değer çiftine sahip koleksiyonların elemanları arasında ileri yönlü iterasyon kullanırken **DictionaryEntry nesneleri üzerinden hareket etmek**, anahtarlar üzerinden hareket etmekten **daha hızlıdır**.*

Dictionary bazlı bir koleksiyonda (çoğunlukla Hashtable ve SortedList) foreach döngüsünü kullanarak yaptığımız iterasyonlarda genellikle kullandığımız iki desen vardır. Bu desenlerden birisinde **Keys** özelliği kullanılır. Keys özelliği ile koleksiyon içerisindeki her bir anahtar üzerinde ileri yönlü hareket sağlanır. Bir anahtara karşılık gelen değeri koleksiyon içerisinden almak için, güncel anahtar koleksiyonun indeksleyicisine parametre olarak verilir.

```
foreach (object anahtar in sl.Keys)
{
}
```

Diğer yöntemde ise DictionaryEntry nesneleri kullanılmaktadır ve deseni aşağıdaki gibidir. DictionaryEntry nesneleri o anki anahtar-değer çiftlerine erişebilmemizi sağlayan Key ve Value özelliklerine sahiptir.

```
foreach (DictionaryEntry dicEnt in sl)
{
}
```

Şimdi bu teoriyi örneğimizde uygulayarak oluşan süre farklarını değerlendirmeye çalışalım. Tester isimli sınıfımıza aşağıdaki iki metodu ekleyerek işe başlıyoruz.

```
public void Dolas_1()
```

```

{
    OlusturHt();
    foreach (object anahtar in sl.Keys)
    {
        object deger = sl[anahtar];
    }
}

public void Dolas_2()
{
    OlusturHt();
    foreach (DictionaryEntry dicEnt in sl)
    {
        object deger = dicEnt.Value;
    }
}

```

Metodlarımızda SortedList koleksiyonunu kullandık. İlk metodumuzda değerlere erişmek için indeksleyici üzerinden anahtarları kullanıyoruz. Yani foreach döngümüz koleksiyon içindeki her bir anahtar için öteleme yapıyor. İkinci döngümüz ise koleksiyon içerisindeki her bir DictionaryEntry nesnesini ele alarak öteleme yapıyor. Her iki tekniği kullanan kodlarımızı ise Main metodumuza aşağıdaki gibi ekleyelim.

#region Key üzerinden döngü

```

dtBaslangic = DateTime.Now;
tester.Dolas_1();
dtBitis = DateTime.Now;
ts = dtBitis - dtBaslangic;
Console.WriteLine(ts.TotalMilliseconds);

```

#endregion

#region DictionaryEntry üzerinden döngü

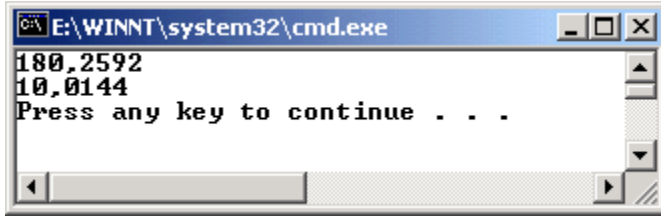
```

dtBaslangic = DateTime.Now;
tester.Dolas_2();
dtBitis = DateTime.Now;
ts = dtBitis - dtBaslangic;
Console.WriteLine(ts.TotalMilliseconds);

```

#endregion

Uygulamamızı çalıştırdığımızda aşağıdakine benzer bir sonuç elde ederiz.



Bu teori Hashtable koleksiyonları içinde geçerlidir. Her iki teoriyide incelediğimiz örneklerin doğrduğu sonuçlar kullandığınız sisteme nazaran görecelidir. Farklı sonuçlar oluşabilir. Özellikle milisaniye cinsinden değerler söz konusu olduğundan çalışma zamanında bu farklar çok önemsizdir. Ancak yinede profesyonel stilde kod yazarken kullanabileceğimiz tekniklerdir.

Özetle SortedList koleksiyonunun kullanım amacı elemanlarının her zaman anahtarlarına göre sıralı tutuluyor oluşudur. Hashtable koleksiyonu ise elemanlarını içeride hash algoritması ile oluşturduğu indekslere göre tutar ve bulur. Hash algoritmasının doğası gereği Hashtable koleksiyonları son derece hızlıdır. Her iki koleksiyonunda ortak noktası IDictionary arayüzlerini uygulamış olmaları ve bu sebepten DictionaryEntry tipinden nesneleri taşımalarıdır. Bu yüzden her iki koleksiyonda bünyesinde key-value çiftlerini barındırır. İşte bu ortak özelliklerden yola çıkarak yukarıdaki iki teori ortaya atılmıştır. Biz de bu makalemizde bunları incelemeye çalıştık. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

Numaralandırıcıları Kullanmak İçin Bir Sebep - 28 Ağustos 2005 Pazar

C#, enum, enumerations, datarow,

Değerli Okurlarım Merhabalar,

Bildiğiniz gibi numaralandırıcılar (enum sabitleride diyebiliriz) yardımıyla sayısal değerleri kod içerisinde daha anlamlı isimlendirmelerle ifade edebiliriz. Uygulama geliştirirken çoğunlukla framework'ün parçası olan pek çok enum sabitini kullanmaktayız. Örneğin veritabanı uygulamalarında sıkça kullandığımız CommandBehavior, DataRowState, DataRowVersion sabitleri gibi. Bu sistemin temel amacı, bu tiplerin sahip oldukları değerlerin sayısal karşılıklarına ihtiyacımızın olmasıdır.

Öyle ki, uygulama içerisinde yer alan her hangi bir fonksiyonun davranışı için sayısal bir karşılaştırma yapmamız gereken yerlerde, bu sayısal değerlerin karşılığı olan bir ismi kullanmak çok daha mantıklıdır. Bu geliştirme açısından zaman kazandırıcı bir tekniktir de ötedir. Çoğu zaman projelerimizde kendi numaralandırıcı tiplerimizi tanımlama ihtiyacı duyarız. İşte bu günkü makalemizde bizi numaralandırıcı kullanmaya itecek bir nedeni incelemeye çalışacağız.



Numaralandırıcılar sayısal değerleri anlamlı isimler ile ifade etmemize yardımcı olurken, gerek kodlama zamanında gerekse geliştirme aşamasında programcıya büyük kolaylık ve esneklik sağlarlar.

İlk olarak numaralandırıcıları kullanmamız için gerekli senaryomuzdan kısaca bahsedelim. Örneğimizi bir web uygulaması olarak geliştireceğiz. Bu web uygulamasında AdventureWorks2000 (Sql Server 2000 Reporting Service ile birlikte gelen) isimli veritabanında yer alan Products ve ProductSubCategory tablolarından faydalanacağız. Temel olarak bir dataGrid kontrolü üzerinde veri gösterme ve güncelleme işlemlerini ele alacağız. Uygulamamızı en başından itibaren tasarlayacağız. İlerleyen kısımlarında ise numaralandırıcı ihtiyacımızı keşfedecek ve uygun çözüm yollarını geliştirmeye çalışacağız. İlk olarak uygulamamızın tablolar ile ilgili temel işlemlerini yapacak yönetici sınıfını yazalım. DBYonetici sınıfımız şu an için sadece veri çekme işlemlerini üstlenecek. İlerleyen kısımlarda diğer bazı işlevsellikleri de ekleyeceğiz.

DBYonetici.cs

```
using System;  
using System.Data;
```

```

using System.Data.SqlClient;

namespace UsingEnumerators
{
    public class DBYoneticisi
    {
        private SqlConnection con;
        private SqlDataAdapter da;

        public DBYoneticisi()
        {
            con=new SqlConnection("data
source=localhost;database=AdventureWorks2000;integrated security=SSPI");
        }

        public DataSet SelectCategories()
        {
            DataSet resultSet=new DataSet();
            string sql=@"SELECT ProductSubCategoryID, Name FROM ProductSubCategory
Order By ProductSubCategoryID";
            da=new SqlDataAdapter(sql,con);
            da.Fill(resultSet);
            return resultSet;
        }

        public DataSet SelectProducts()
        {
            DataSet resultSet=new DataSet();
            string sql=@"SELECT TOP 100 ProductID, Name, ProductNumber, StandardCost,
ListPrice, StandardCost / ListPrice * 100 AS IncRate, Class, DealerPrice, SellStartDate,
Size, Weight,ProductSubCategoryID, (SELECT Name FROM ProductSubCategory WHERE
ProductSubCategoryID = P.ProductSubCategoryID) AS ProductSubCategory FROM
Product P WHERE (StandardCost IS NOT NULL) AND (ListPrice IS NOT NULL) ORDER BY
ProductID";
            da=new SqlDataAdapter(sql,con);
            da.Fill(resultSet);
            return resultSet;
        }
    }
}

```

DBYoneticisi isimli sınıfımızdaki metotlardan kısaca bahsetmekte yarar var. Uygulamamızda AdventureWorks2000 veritabanında yer alan Products isimli tabloyu kullanıyoruz. Bu tablodaki bir kaç alanı ele alacağız. Products tablosuna ilişkin sorgumuza dikkat ederseniz, standart maliyet ile liste fiyatı arasındaki oranı ifade eden ek bir alan olduğunu farkedebilirsiniz. Biz bu alandaki değere bakarak dataGrid' imiz üzerinde renklendirme işlemlerini yapmaya çalışacağız.

SelectCategories isimli metodumuz ise ürünlerimizin ait olabileceği kategorileri elde etmemizi sağlıyor. Aslında bu metodu, dataGrid üzerinde her hangibir satır için güncelleme işlemi yapacağımız zaman, bir ListBox kontrolünü dinamik olarak doldurmak amacıyla kullanacağız. Şimdi DBYoneticici sınıfımızı web uygulamamızda kullanalım. İlk olarak WebForm'umuz üzerinde yer alacak dataGrid kontrolümüzün içeriğini oluşturacağız.

```
<asp:datagrid id="dgProducts" runat="server" Font-Size="Smaller" Font-Families="Verdana"
AutoGenerateColumns="False" DataKeyField="ProductID" AllowPaging="True">
  <AlternatingItemStyle BackColor="LightSteelBlue"></AlternatingItemStyle>
  <ItemStyle BackColor="White"></ItemStyle>
  <HeaderStyle Font-Bold="True" ForeColor="Brown" BackColor="Gold"></HeaderStyle>
  <Columns>
    <asp:BoundColumn Visible="False" DataField="ProductID"></asp:BoundColumn>
    <asp:TemplateColumn HeaderText="Urun Adı">
      <ItemTemplate>
        <%#DataBinder.Eval(Container.DataItem,"Name")%>
      </ItemTemplate>
      <EditItemTemplate>
        <asp:TextBox ID="txtName" Runat="server" Width="100"
Text='<%#DataBinder.Eval(Container.DataItem,"Name")%>'></asp:TextBox>
      </EditItemTemplate>
    </asp:TemplateColumn>
    <asp:TemplateColumn HeaderText="Urun Numarası">
      <ItemTemplate>
        <%#DataBinder.Eval(Container.DataItem,"ProductNumber")%>
      </ItemTemplate>
      <EditItemTemplate>
        <%#DataBinder.Eval(Container.DataItem,"ProductNumber")%>
      </EditItemTemplate>
    </asp:TemplateColumn>
    <asp:TemplateColumn HeaderText="Maliyet">
      <ItemTemplate>
        <%#DataBinder.Eval(Container.DataItem,"StandardCost",{0:C})%>
      </ItemTemplate>
      <EditItemTemplate>
        <asp:TextBox ID="txtStandardCost" Runat="server" Width="100"
Text='<%#DataBinder.Eval(Container.DataItem,"StandardCost")%>'></asp:TextBox>
      </EditItemTemplate>
    </asp:TemplateColumn>
    <asp:TemplateColumn HeaderText="Liste Fiyatı">
      <ItemTemplate>
        <%#DataBinder.Eval(Container.DataItem,"ListPrice",{0:C})%>
      </ItemTemplate>
      <EditItemTemplate>
        <asp:TextBox ID="txtListPrice" Runat="server" Width="100"
Text='<%#DataBinder.Eval(Container.DataItem,"ListPrice")%>'></asp:TextBox>
```

```

        </EditItemTemplate>
    </asp:TemplateColumn>
    <asp:TemplateColumn HeaderText="Artis Orani">
        <ItemTemplate>
            <asp:Label ID="lblIncRate" Runat="Server"
Text='<#DataBinder.Eval(Container.DataItem,"IncRate","{0:F}")%>'></asp:Label>
        </ItemTemplate>
        <EditItemTemplate>
            <#DataBinder.Eval(Container.DataItem,"IncRate","{0:F}")%>
        </EditItemTemplate>
    </asp:TemplateColumn>
    <asp:TemplateColumn HeaderText="Alt Kategori ID" Visible="False">
        <ItemTemplate>
            <#DataBinder.Eval(Container.DataItem,"ProductSubCategoryID")%>
        </ItemTemplate>
        <EditItemTemplate>
            <asp:Label ID="lblProductSubCategoryID"
Text='<#DataBinder.Eval(Container.DataItem,"ProductSubCategoryID")%>'
runat="server"></asp:Label>
        </EditItemTemplate>
    </asp:TemplateColumn>
    <asp:TemplateColumn HeaderText="Alt Kategori">
        <ItemTemplate>
            <#DataBinder.Eval(Container.DataItem,"ProductSubCategory")%>
        </ItemTemplate>
        <EditItemTemplate>
            <asp:DropDownList ID="lstProductSubCategory" runat="server" DataSource =
"<# AltKategorileriAl() %>" DataTextField="Name"
DataValueField="ProductSubCategoryID" />
        </EditItemTemplate>
    </asp:TemplateColumn>
    <asp:TemplateColumn HeaderText="Boyutu">
        <ItemTemplate>
            <#DataBinder.Eval(Container.DataItem,"Size","{0:F}")%>
        </ItemTemplate>
        <EditItemTemplate>
            <#DataBinder.Eval(Container.DataItem,"Size","{0:F}")%>
        </EditItemTemplate>
    </asp:TemplateColumn>
    <asp:TemplateColumn HeaderText="Agirlik">
        <ItemTemplate>
            <#DataBinder.Eval(Container.DataItem,"Weight","{0:F}")%>
        </ItemTemplate>
        <EditItemTemplate>
            <asp:TextBox ID="txtWeight" Runat="server" Width="100"
Text='<#DataBinder.Eval(Container.DataItem,"Weight")%>'></asp:TextBox>
        </EditItemTemplate>
    </asp:TemplateColumn>

```



```

        <asp:TemplateColumn>
            <ItemTemplate>
                <asp:Button id="btnDuzenle" CommandName="Duzenle" runat="server"
Text="Düzenle"></asp:Button>
            </ItemTemplate>
            <EditItemTemplate>
                <asp:Button id="btnGuncelle" CommandName="Guncelle" runat="server"
Text="Güncelle"></asp:Button>
                <asp:Button id="btnVazgeç" CommandName="Vazgeç" runat="server"
Text="Vazgeç"></asp:Button>
            </EditItemTemplate>
        </asp:TemplateColumn>
        <asp:TemplateColumn>
            <ItemTemplate>
                <asp:Button id="btnDelete" CommandName="Sil" runat="server"
Text="Sil"></asp:Button>
            </ItemTemplate>
        </asp:TemplateColumn>
    </Columns>
    <PagerStyle Mode="NumericPages"></PagerStyle>
</asp:datagrid>

```

Şu anda aklınızdan geçenleri duyar gibiyim. Bu kadar uzun kodlardan sonra numarandırıcıları nerede kullanacağımızı merak ediyorsunuz. Ama biraz daha sabrederim. İlk olarak yukarıdaki dataGrid kontrolümüz hakkında kısaca bilgi vermekte fayda var. DataGrid kontrolümüz ekranda Products tablosuna ait verileri gösterdiği gibi, satırlar üzerinde veri güncelleme ve silme işlemlerine de izin verecek şekilde oluşturuldu. Bu sebepten üzerinde güncelleme yapılmasını istediğimiz alanlara ait EditItemTemplate kısımlarında TextBox veya DropDownList gibi kontrollere yer verdik. Konumuz dataGrid kontrolünün özelliklerini incelemek olmadığı için çok fazla detaya girmiyorum. Şimdi sayfamızın kodlarını yazalım.

```
private static DBYoneticisi yoneticisi;
```

```
private void Page_Load(object sender, System.EventArgs e)
{
    yoneticisi=new DBYoneticisi();
}

```

```
private void btnYukle_Click(object sender, System.EventArgs e)
{
    dgProducts.DataSource=yoneticisi.SelectProducts();
    dgProducts.DataBind();
}

```

```
private void dgProducts_ItemDataBound(object sender, DataGridItemEventArgs e)
{

```

```

        if((e.Item.ItemType==ListItemType.Item) ||
        (e.Item.ItemType==ListItemType.AlternatingItem))
        {
            double incRate;
            Label lbl=(Label)e.Item.Cells[5].Controls[1];
            incRate=Convert.ToDouble(lbl.Text);
            if(incRate>=100)
            {
                e.Item.Cells[5].BackColor=Color.Red;
                e.Item.Cells[5].ForeColor=Color.White;
            }
        }
    }
}

```

```

private void dgProducts_PageIndexChanged(object source,
DataGridPageChangedEventArgs e)
{
    dgProducts.CurrentPageIndex=e.NewPageIndex;
    dgProducts.DataSource=yonetici.SelectProducts();
    dgProducts.DataBind();
}

```

```

public DataSet AltKategorileriAl()
{
    return yonetici.SelectCategories();
}

```

```

private void dgProducts_ItemCommand(object source, DataGridCommandEventArgs e)
{
    switch (e.CommandName)
    {
        case "Duzenle":
            dgProducts.EditItemIndex=e.Item.ItemIndex;
            dgProducts.DataSource=yonetici.SelectProducts();
            dgProducts.DataBind();
            break;

        case "Sil":
            break;

        case "Vazgec":
            dgProducts.EditItemIndex=-1;
            dgProducts.DataSource=yonetici.SelectProducts();
            dgProducts.DataBind();
            break;

        default :
            break;
    }
}

```

```
}  
}
```

Dilerseniz kodlarımızda neler yaptığımıza kısaca değinelim. Sayfamız yüklenirken dataGrid kontrolü, Products tablosu için çalıştırdığımız sorgudan dönen sonuç kümesi ile dolduruluyor. Bu sırada dataGrid kontrolünün ItemDataBound olayında her bir satır için IncRate isimli alanın değerini kontrol ediyoruz. Bunun sonucuna göre eğer artış oranı 100'den büyük ise o hücrenin arka plan rengini ve font rengini değiştirerek kullanıcıyı uyarıyoruz.

Kullanıcı dataGrid kontrolünde sayfalama işlemi yapabilir. Sayfalama işlemi sağlamak için her zamanki gibi PageIndexChanged isimli olay metodumuzu kullanıyoruz. Her hangibir satır için güncelleme işlemi ve silme işlemi de yapabilir. Güncelleme ve Silme işlemleri için yine DataGrid kontrolümüzün ItemCommand metodunu kullanıyoruz. Eğer düzenleme modu seçilirse bu durumda ürünlerin dahil olduğu kategorileri listelemek için DropDownList kontrolümüzü dolduran SelectProducts metodumuzu çağırıyoruz. Böylece sistemde kayıtlı olan ürünlerin kullanıcı tarafından bir combo kontrolü içerisinde seçilebilmesini sağlamış oluyoruz. Bu hızlı açıklamalardan sonra geldiğimiz noktayı aşağıdaki şekilden görebilirsiniz.

Ürün Fiyat Listesi									
Yükle									
Ürün Adı	Ürün Numarası	Maliyet	Liste Fiyatı	Artış Oranı	Alt Kategori	Boyutu	Ağırlık		
LL Bottom Bracket	BB-7421	55,0000 TL	55,0000 TL	101,87	Bib-Short	15	223	Güncelle Vazgeç	Sil
ML Bottom Bracket	BB-8107	74,92 TL	101,24 TL	74,00	Bottom Bracket	12	168,00	Düzenle	Sil
HL Bottom Bracket	BB-9108	100,00 TL	121,49 TL	82,31	Bottom Bracket	10	170,00	Düzenle	Sil
Mountain-500 Black, #0	BK-M18B-40	600,00 TL	539,99 TL	111,11	Mountain Bike	40	27,35	Düzenle	Sil
Mountain-500 Black, #2	BK-M18B-42	600,00 TL	539,99 TL	111,11	Mountain Bike	42	27,77	Düzenle	Sil
Mountain-500 Black, #4	BK-M18B-44	600,00 TL	539,99 TL	111,11	Mountain Bike	44	28,13	Düzenle	Sil
Mountain-500 Black, #8	BK-M18B-48	610,00 TL	539,99 TL	112,96	Mountain Bike	48	28,42	Düzenle	Sil

Şimdi bu uygulamada odaklanmamız gereken bir nokta var. O da, IncRate' in rengini belirlediğimiz ItemDataBound metodu. Gelin bu metodu mercek altına alalım.

```
private void dgProducts_ItemDataBound(object sender, DataGridItemEventArgs e)
{
    if((e.Item.ItemType==ListItemType.Item) ||
    (e.Item.ItemType==ListItemType.AlternatingItem))
    {
        double incRate;
        Label lbl=(Label)e.Item.Cells[5].Controls[1];
        incRate=Convert.ToDouble(lbl.Text);
        if(incRate>=100)
        {
            e.Item.Cells[5].BackColor=Color.Red;
            e.Item.Cells[5].ForeColor=Color.White;
        }
    }
}
```

Bu metodda dikkat ederseniz if döngümüz içerisinde 5 numaralı hücrenin içerisinde ki 1 numaralı kontrolü ele alıyoruz. Peki bu 5 numaralı hücredeki 1 numaralı kontrol' de neyin nesi oluyor. Aslında dataGrid nesnemizin aspx sayfasındaki kodlarını göz önüne aldığınızda 5 numaralı hücrenin, IncRate alanına ait hücre olduğunu ve 1 numaralı kontrolünde bu hücredeki Label kontrolü olduğunu kolayca görebilirsiniz.

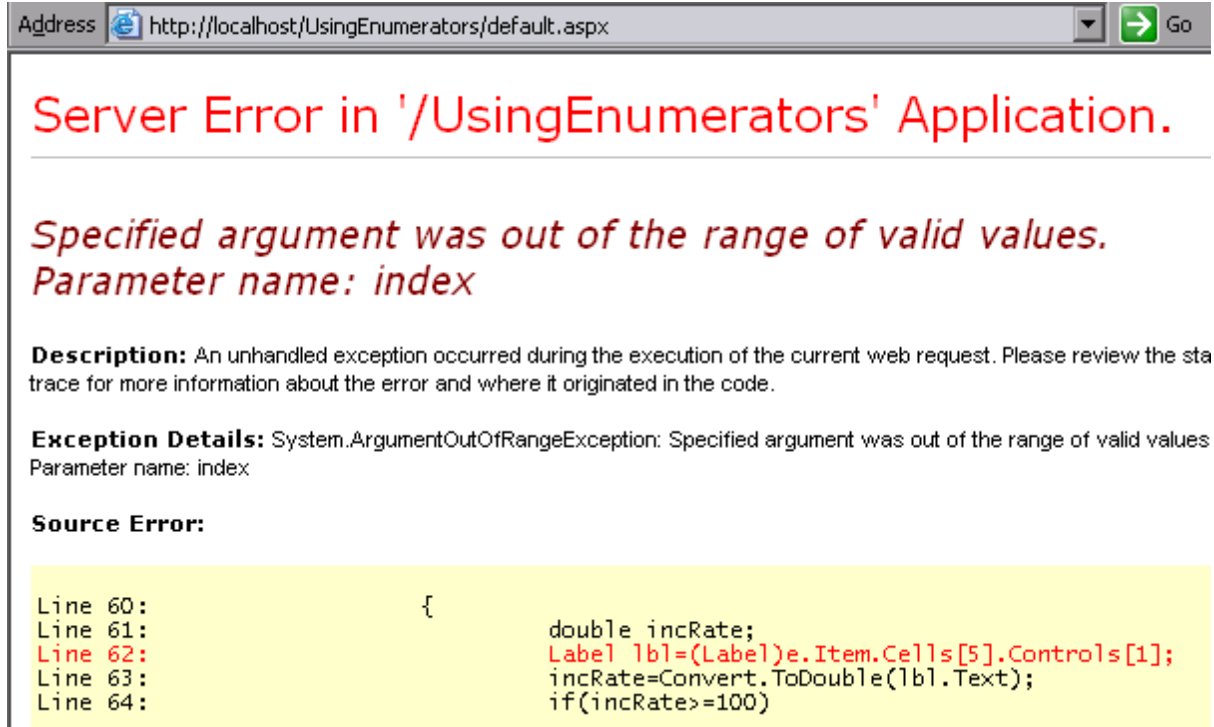
```
<asp:TemplateColumn HeaderText="Artis Orani">
    <ItemTemplate>
        <asp:Label ID="lblIncRate" Runat="Server"
Text='<%=#DataBinder.Eval(Container.DataItem,"IncRate","{0:F}")%>'></asp:Label>
    </ItemTemplate>
    <EditItemTemplate>
        <%=#DataBinder.Eval(Container.DataItem,"IncRate","{0:F}")%>
    </EditItemTemplate>
</asp:TemplateColumn>
```

Herşey buraya kadar gayet güzel. Şimdi default.aspx sayfamızda IncRate alanını dataGrid içerisinde kullandığımız ItemTemplate takısının hemen önüne yeni bir tane daha ekleyelim. Bu sefer query' den çektiğimiz Class alanını buraya ekliyoruz. Aynen aşağıdaki kod parçasında olduğu gibi.

```
<asp:TemplateColumn HeaderText="Sınıf">
    <ItemTemplate>
        <%=#DataBinder.Eval(Container.DataItem,"Class","{0:F}")%>
    </ItemTemplate>
    <EditItemTemplate>
        <%=#DataBinder.Eval(Container.DataItem,"Class","{0:F}")%>
    </EditItemTemplate>
```

</asp:TemplateColumn>

Şimdi uygulamamızı tekrardan çalıştıralım. Bu sefer sizinde benim de beklediğimiz gibi bir şeylerin yolunda gitmemesi gerekiyor. Aşağıdaki ekran görüntüsü korktuğumuzun başımıza geldiği andır.



Hata gayet açık ve net. Artık 5 numaralı hücremiz IncRate alanına ait hücre değil. Burada şimdi Class isimli alanımız yer almakta. Ayrıca 5 numaralı alandaki 1 numaralı kontrolümüzde bir Label kontrolü değil. Hatanın nedenide doğru hücre üzerinde olmayışımız. Daha da kötüsü olabilir elbette. Burada çalışma zamanında aldığımız hata ile sorununun ne olduğunu anlayabildik. Oysaki, yeni eklediğimiz hücre pekala sayısal bir değer içeren bir Label kontrolü taşıyabilirdi. Bu durumda uygulama çalışacaktı ama hatalı sonuçlar verecekti.

Peki ya çözüm? Çözüm sonunda bu makalenin başından beridir bahsetmeye çalıştığımız numaralandırıcılardan geçiyor. Sorgudan dönen alanların 'dataGrid' de denk geldiği indeks numaralarını temsil edecek bir numaralandırıcımız olsaydı daha iyi olmaz mıydı? En azından yeni bir hücre eklediğimizde tek yapmamız gereken enum sabitimizde araya bir eleman daha eklemek olacaktı. Çünkü kodun kaç yerinde aynı numaralı indeksi kullandığımızı bilemeyebilirdik. Her ne kadar arama-bulma yöntemi ile 5' lerin geçtiği yerleri 6 yapabilecek olsakta bu hiçde profesyonelce bir çözüm olmazdı. İşte basit bir enum sabitini kullanarak çok daha profesyonel bir çözüm üretebilirdik.

Gördüğünüz gibi enum sabitlerini kullanmak için son derece güzel bir sebebimiz oldu. Şimdi bu düşüncemizi uygulamamız ile bütünleştirmemiz gerekiyor. İlk olarak yapmamız gereken enum sabitimizi oluşturmak. Eğer bir solution içerisinde pek çok proje ile birlikte çalışıyorsanız bu durumda grid başlıkları gibi

indekslere ait enum sabitlerini saklayabileceğiniz genel bir katman dahi oluşturmayı düşünebilirsiniz. Bizim uygulamamız son derece küçük olduğundan geçerli isim alanımız altında bir enum sabiti yapmamız yeterli olacaktır.

enum sabitimiz ProductsGridHeaders

```
namespace UsingEnumerators
{
    enum ProductsGridHeaders
    {
        PRODUCTID,
        NAME,
        PRODUCTNUMBER,
        STANDARDCOST,
        LISTPRICE,
        CLASS,
        INCRATE,
        PRODUCTSUBCATEGORYID,
        PRODUCTSUBCATEGORY,
        SIZE,
        WEIGHT
    }
}
```

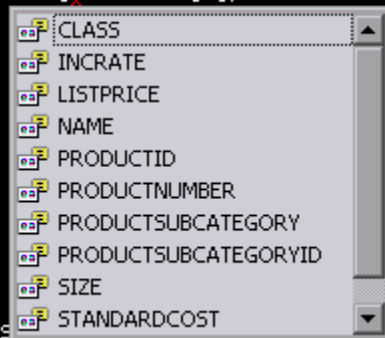
İşte altın yumruğu vurduğumuz an. Artık tek yapmamız gereken, enum sabitimizi DataGridView nesnemizin ItemDataBound olay metoduyla aşağıdaki gibi kullanmak.



Numaralandırıcı içindeki elemanları isimlendirirken gerçekten anlamlı, bir şeyleri kulağımıza çağrıştıran isimler vermeye özen göstermeliyiz.

```
private void dgProducts_ItemDataBound(object sender, DataGridViewItemEventArgs e)
{
    if((e.Item.ItemType==ListItemType.Item) || (e.Item.ItemType==ListItemType.AlternatingItem))
    {
        double incRate;
        Label lbl=(Label)e.Item.Cells[ProductsGridHeaders.INCRATE].Controls[1];
        incRate=Convert.ToDouble(lbl.Text);
        if(incRate>=100)
        {
            e.Item.Cells[5].BackColor=Color.Red;
            e.Item.Cells[5].ForeColor=Color.White;
        }
    }
}

private void dgProducts_PageIndexChanged(object sender, EventArgs e)
```



```
private void dgProducts_ItemDataBound(object sender, DataGridViewItemEventArgs e)
```

```

{
    if((e.Item.ItemType==ListItemType.Item) ||
(e.Item.ItemType==ListItemType.AlternatingItem))
    {
        double incRate;
        Label lbl=(Label)e.Item.Cells[(int)ProductsGridHeaders.INCRATE].Controls[1];
        incRate=Convert.ToDouble(lbl.Text);
        if(incRate>=100)
        {
            e.Item.Cells[(int)ProductsGridHeaders.INCRATE].BackColor=Color.Red;
            e.Item.Cells[(int)ProductsGridHeaders.INCRATE].ForeColor=Color.White;
        }
    }
}

```

Artık uygulamamız sorunsuz şekilde çalışacaktır ve istediğimiz yeni satırı dataGrid kontrolümüze ekleyebiliriz. Unutmamamız gereken tek şey, yeni bir satır eklendiğinde bunu ilgili enum sabitinde yansıtmaktır. Üstelik kodun okunurluğuda inanılmaz derecede kolaylaşmıştır. Ancak elbette ki kodumuzda enum sabitlerini kullanacağımız tek yer burası değil.

Örneğin DropDownList kontrolümüz doldurulduğunda o an aktif olan satıra ait ürün kategori adını seçili olarak gelmesi gerekmektedir. Şu anki kodlarımıza göre DropDownList kontrolümüz herhangi bir satır edit modunda açıldığında ürünleri başarılı bir şekilde listeliyor ama her zaman için ilk elemana konumlanıyor. Bu problemi aşmak için düzenleme moduna geçildiğinde DropDownList kontrolünün SelectedValue özelliğini uygun değere (ki bu değer ProductSubCategoryId olacaktır) atamamız yeterli olacaktır. Dolayısıyla dataGrid kontrolümüzün ItemDataBound olay metodunda aşağıdaki düzenlemeyi yapmamız bu ihtiyacımızı karşılayacaktır. Artık kodu yazarken, hangi indisli hücreyi düşünmemize gerekte yoktur. Çünkü numaralandırıcımız anlamlı isimleri ile bunu bize söylemektedir.

```

if(e.Item.ItemType==ListItemType.EditItem)
{
    Label
    lblKatId=(Label)e.Item.Cells[(int)ProductsGridHeaders.PRODUCTSUBCATEGORYID].Controls[1];
    DropDownList
    lstKategoriler=(DropDownList)e.Item.Cells[(int)ProductsGridHeaders.PRODUCTSUBCATEGORYID].Controls[1];
    lstKategoriler.SelectedIndex=Convert.ToInt16(lblKatId.Text)-1;
}

```

Oluşturduğumuz enum sabiti özellikle güncellenecek satır için yazacağımız kod satırlarında da çok işe yarayacaktır. Son olarak güncelleme işlemimizi numaralandırıcımızı kullanarak nasıl gerçekleştirdiğimizi görelim. Uygulamamıza bu fonksiyonelliği kazandırmak için yine

ItemDataBound olay metodunu kullanacağız. Bu sefer CommandName özelliğinin Guncelle değerini alması halinde gerçekleştireceğimiz işlemler var.

Biz numaralandırıcımızı Kaydetme işleminin gerçekleştirileceği metodu çağırmadan önce, gridde seçili olan satırdaki kontroller üzerindeki değerleri okuduğumuz satırlarda kullanıyoruz. Örneğin dataGrid kontrolünde Name alanının gösterildiği hücrede yer alan TextBox kontrolüne girilen değeri almak için, ProductsGridHeaders numaralandırıcısının Name elemanının işaret ettiği integer değere sahip olan hücredeki 1 numaralı kontrolü ele alıyoruz. Daha sonra bu kontrolü bir TextBox nesnesine dönüştürerek güncel değerini ilgili kaydetme işlemi gerçekleştirecek metodumuza taşıyoruz. Bu değer alma işlemi diğer kontrollerimiz içinde yapıyoruz. Ancak önemli olan e parametresi üzerinden dataGrid kontrolündeki ilgili alanlara nasıl eriştiğimiz. Yani numaralandırıcımızı nasıl kullandığımız.

```
private void Kaydet(DataGridCommandEventArgs e)
{
    int productID=Convert.ToInt32(dgProducts.DataKeys[e.Item.ItemIndex]);
    TextBox txtName=(TextBox)e.Item.Cells[(int)ProductsGridHeaders.NAME].Controls[1];
    TextBox
txtStandardCost=(TextBox)e.Item.Cells[(int)ProductsGridHeaders.STANDARDCOST].Contr
ols[1];
    TextBox
txtListPrice=(TextBox)e.Item.Cells[(int)ProductsGridHeaders.LISTPRICE].Controls[1];
    TextBox txtWeight=(TextBox)e.Item.Cells[(int)ProductsGridHeaders.WEIGHT].Controls[1];
    DropDownList
lstSubCategories=(DropDownList)e.Item.Cells[(int)ProductsGridHeaders.PRODUCTSUBCA
TEGORY].Controls[1];

    string Name=txtName.Text;
    double StandardCost=Convert.ToDouble(txtStandardCost.Text);
    double ListPrice=Convert.ToDouble(txtListPrice.Text);
    double Weight=Convert.ToDouble(txtWeight.Text);
    int SCategoryID=Convert.ToInt16(lstSubCategories.SelectedValue);

    yoneticici.UpdateProducts(productID,Name,StandardCost,ListPrice,Weight,SCategoryID);
}

private void dgProducts_ItemCommand(object source, DataGridCommandEventArgs e)
{
    switch (e.CommandName)
    {
        case "Duzenle":
            dgProducts.EditItemIndex=e.Item.ItemIndex;
            dgProducts.DataSource=yoneticici.SelectProducts();
            dgProducts.DataBind();
            break;

        case "Guncelle":
            Kaydet(e);
    }
}
```



```

        dgProducts.EditItemIndex=-1;
        dgProducts.DataSource=yonetici.SelectProducts();
        dgProducts.DataBind();
        break;

    case "Sil":
        break;

    case "Vazgeç":
        dgProducts.EditItemIndex=-1;
        dgProducts.DataSource=yonetici.SelectProducts();
        dgProducts.DataBind();
        break;

    default :
        break;
}
}

```

DbYoneticisi sınıfımızda güncelleme işlemini gerçekleştiren metodumuz ise aşağıdaki gibidir.

```

public int UpdateProducts(int productID, string name, double standardCost, double listPrice,
double weight, int subCategoryID)
{
    string sql=@"UPDATE Product SET Name=@Name,StandardCost=@StandardCost,
ListPrice=@ListPrice,Weight=@Weight, ProductSubCategoryID=@SubCategoryID WHERE
ProductID=@ProductID";
    SqlCommand cmd=new SqlCommand(sql,con);
    cmd.Parameters.Add("@Name",name);
    cmd.Parameters.Add("@StandardCost",standardCost);
    cmd.Parameters.Add("@ListPrice",listPrice);
    cmd.Parameters.Add("@Weight",weight);
    cmd.Parameters.Add("@SubCategoryID",subCategoryID);
    cmd.Parameters.Add("@ProductID",productID);
    con.Open();
    int result=cmd.ExecuteNonQuery();
    con.Close();
    return result;
}

```

Gördüğünüz gibi enum sabitlerini kullanarak kod geliştirmek daha da kolaylaşmaktadır. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

[Örnek Kodlar İçin Tıklayınız.](#)

Web Uygulamalarında Custom Paging -

03 Eylül 2005 Cumartesi

asp.net, paging, custom paging,

Değerli Okurlarım Merhabalar,

Geliştirdiğimiz web uygulamalarında özellikle DataGrid kontrollerini kullandığımızda sayfalama işlemini sıkça kullanırız. Genellikle sayfalama işlemlerini var sayılan hali ile kullanırız. Bu modele göre grid üzerinde sayfalama yapabilmek için PageIndexChanged olayını ele almamız gerekir. Burada grid kontrolüne yeni sayfa numarasını DataGridPageChangedEventArgs parametresinin NewPageIndex değeri ile verir ve bilgilerin tekrardan yüklenmesini sağlayacak uygulama kodlarımızı yürütürüz. Tipik olarak bu tarz bir kullanım aşağıdaki kod parçasında olduğu gibi yapılmaktadır.

```
private void Doldur()
{
    //Bağlantının açılması, verilerin çekilmesi ve çekilen verilerin DataGrid kontrolüne bağlanması
}

private void DataGrid1_PageIndexChanged(object source,
DataGridPageChangedEventArgs e)
{
    DataGrid1.CurrentPageIndex=e.NewPageIndex;
    Doldur();
}
```

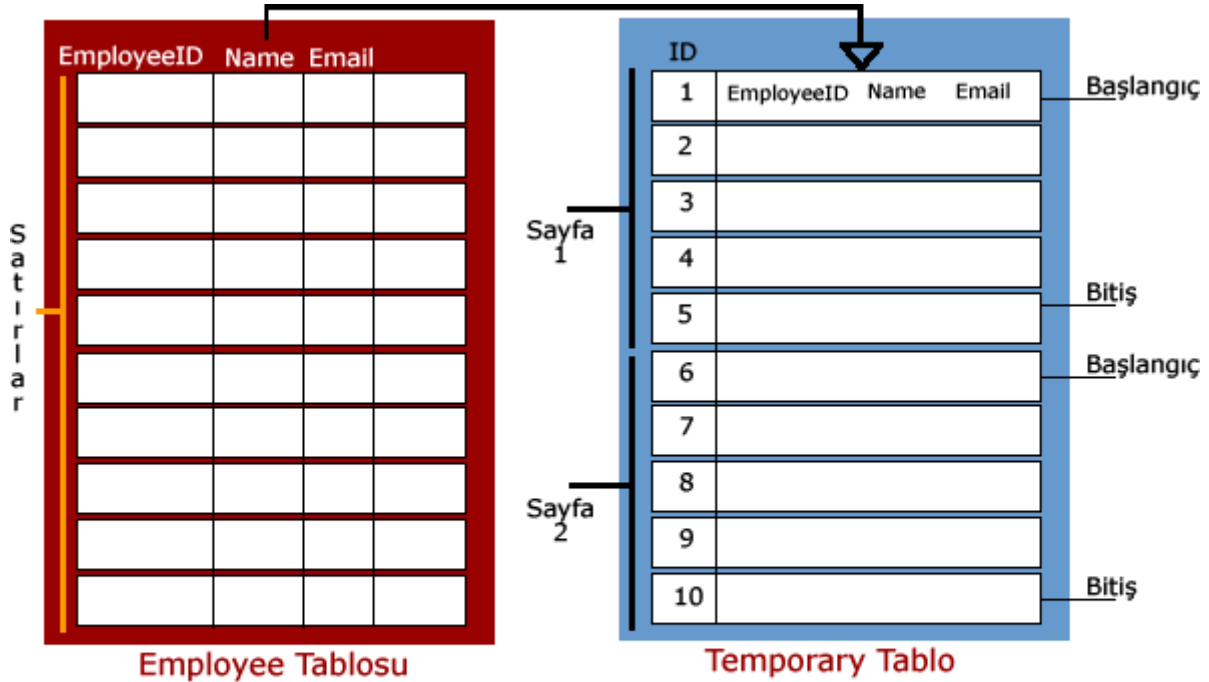
Buradaki yaklaşım gerçekten işe yaramaktadır. Ancak aslında performans açısından bazı kayıplar söz konusudur. Çünkü bu tip sayfalama tekniğini kullanırken, sayfa linklerine her basışımızda ilgili veri kaynağındaki tüm veriler çekilmekte ve çekilen veri kümesi üzerinde ilgili sayfaya gidilmektedir. Bu elbetteki büyük boyutlu veri kümeleri ile çalışırken dikkate alınması gereken bir durumdur. Nitekim performansı olumsuz yönde etkileyecektir. Her ne kadar caching (tampon belleğe almak) teknikleri ile sorunun biraz olsun üstesinden gelinebilecek olsa da daha etkili çözümler üretebiliriz.



Büyük boyutlu veri kümeleri ile çalışırken uygulanan varsayılan sayfalama tekniği hız kaybına neden olarak performansı olumsuz yönde etkileyebilir.

İşte bu makalemizde özel sayfalama tekniklerinden bir tanesini incelemeye çalışacağız. Bu teknikte yer alan parçalardan en önemlisi şablon bir tablonun (temporary) kullanılmasıdır. İlk olarak asıl veri kümesini ele alacağımız bir stored

procedure (saklı yordam) geliştireceğiz. Bu saklı yordamımız içerisinde asıl veri kümesinin satırlarını alıp temporary bir tabloya aktaracağız. Temporary tablomuzun en büyük özelliği identity tipinde 1 den başlayan ve 1' er artan bir alana sahip olmasıdır. Biz bu alanın değerinden faydalanarak temp tablosu üzerinde sayfalama işlemini uygulayacağız. Buradaki ana fikri daha iyi anlamak için aşağıdaki şekile bir göz atalım.



$$\text{Başlangıç} = ((\text{Sayfa Numarası} - 1) * \text{Gösterilecek Kayıt Sayısı}) + 1$$

$$\text{Bitiş} = (\text{Sayfa Numarası} * \text{Gösterilecek Kayıt Sayısı}) + 1$$

Senaryomuzda AdventureWorks2000 veritabanı içerisinde yer alan Employee isimli tabloyu kullanacağız. Bu tablo üzerinde aşağıdaki select sorgusu için sayfalama işlemini gerçekleştireceğiz.

```
SELECT EmployeeID,FirstName,LastName,NationalIDNumber,Title,BirthDate,EmailAddress
FROM Employee
```

Teorimizi şekil üzerinden açıklamaya çalışalım. Önce bir temp tablosu oluşturacağız. Temp tablomuzun alanları yukarıdaki select sorgusundaki alanları karşılayacak şekilde olacak. Bir de ekstradan identity alanımız olacak. Sonra select sorgumuzdaki verileri, temp tablomuz içerisine insert edeceğiz. Ardından temp tablomuz üzerinden yeni bir select sorgusu çalıştıracağız. Ancak bu sefer, Where koşulumuz olacak ve burada identity alanımız için bir aralık belirleyeceğiz. İşte bu aralık sayfanın başlangıç ve bitiş satırlarını belirleyerek istediğimiz sayfaya ait verileri elde etmemizi sağlayacak.

Örneğin, verilerimizi 5' er satırlık sayfalara bölmek istediğimizi düşünelim. Bu durumda 2nci sayfadaki ilk satırın id değerini Baslangic isimli formülümüzden 6 olarak bulabiliriz. Yine 2nci sayfanın bitis satırının değerinde Bitis isimli formülü

kullanarak 10 olarak bulabiliriz. Gördüğünüz gibi teori son derece basit. Sayfalamayı gerçekleştirebilmek için aslında temp tablodaki identity alanlarını kullanıyoruz. Son olarak bu işlemlerin hepsini bir stored procedure(saklı yordam) içerisinde barındırarak işlemlerin doğrudan sql sunucusu üzerinde ne hızlı şekilde gerçekleştirilmesini sağlıyoruz. İşte sp kodlarımız.

```
CREATE PROCEDURE WorkSp_Paging
```

```
@SayfaNo INT,  
@GosterilecekSatirSayisi INT
```

```
AS
```

```
DECLARE @BaslangicID AS INT  
DECLARE @BitisID AS INT  
DECLARE @SelectSorgusu AS NVARCHAR(255)
```

```
-- Önce Temporary tablomuzu oluşturuyoruz. ID alanı önemli.
```

```
CREATE TABLE #TempOfEmployee
```

```
(  
ID INT IDENTITY(1,1),  
EmployeeID INT,  
FirstName NVARCHAR(50),  
LastName NVARCHAR(50),  
NationalIDNumber NVARCHAR(15),  
Title NVARCHAR(50),  
BirthDate DATETIME,  
EmailAddress NVARCHAR(50)  
)
```

```
-- Employee tablosundaki verileri temporary tablomuzla aktarıyoruz.
```

```
SET @SelectSorgusu='SELECT
```

```
EmployeeID,FirstName,LastName,NationalIDNumber,Title,BirthDate,EmailAddress FROM  
Employee'
```

```
INSERT INTO #TempOfEmployee EXEC (@SelectSorgusu)
```

```
-- Başlangıç ve bitiş satırlarının ID alanlarının değerlerini belirlemek için formülasyonumuzu  
kullanıyoruz. SayfaNo ve GösterilecekSatirSayisi sp mize dışarıdan gelen parametreler.
```

```
SET @BaslangicID=((@SayfaNo-1)*@GosterilecekSatirSayisi)
```

```
SET @BitisID=(@SayfaNo*@GosterilecekSatirSayisi)+1
```

```
-- Temporary tablomuz üzerinden ilgili ID aralığındaki veri setini çekiyoruz.
```

```
SELECT  
ID,EmployeeID,FirstName,LastName,NationalIDNumber,Title,BirthDate,EmailAddress  
FROM #TempOfEmployee  
WHERE ID>@BaslangicID AND ID<@BitisID
```

```
-- Son olarak sistemde bir karmaşıklığa yer vermemek için temporary tablomuzu kaldırıyoruz.  
DROP TABLE #TempOfEmployee  
GO
```

Şimdi sp' mizi asp.net uygulamamızda kullanalım. Özel sayfalama yaptığımız için artık PageIndexChanged olayını kullanamayacağız. Dolayısıyla sayfa linklerini manuel olarak oluşturmamız gerekiyor. Bu durumda DataGrid kontrolümüze ait AllowPaging ve AllowCustomPaging özelliklerinin değerlerini false olarak bırakabiliriz. Eğer sadece ilk, önceki, sonraki, son tarzında linkler oluşturacak isek işimiz kolay. İlgili metodumuza sayfa numarasını ve göstereceğimiz satır sayısını parametre olarak göndermemiz yeterli olacaktır. Ancak sayfa numalarını link olarak sunmak istiyorsak biraz daha fazla çabalamamız gerekecek. Öncelikle asıl işi yapan metodumuzu aşağıdaki gibi oluşturalım. Bu metodumuzda tanımlamış olduğumuz sp' mizi bir SqlCommand nesnesi yardımıyla yürütüyor ve elde ettiğimiz sonuç kümesini DataGrid kontrolümüze bağlıyoruz.

```
private void Doldur(int sayfaNo,int gosterilecekSatirSayisi)  
{  
    string sql=@"WorkSp_Paging";  
    cmd=new SqlCommand(sql,con);  
    cmd.CommandType=CommandType.StoredProcedure;  
    cmd.Parameters.Add("@SayfaNo",sayfaNo);  
    cmd.Parameters.Add("@GosterilecekSatirSayisi",gosterilecekSatirSayisi);  
    da=new SqlDataAdapter(cmd);  
    dt=new DataTable();  
    da.Fill(dt);  
    DataGrid1.DataSource=dt;  
    DataGrid1.DataBind();  
}
```

Şimdi linklerimizi oluşturalım. Burada kodlama tekniği açısından tamamen serbestsiniz. Ben aşağıdaki gibi kodlamayı tercih ettim. Öncelikle Employee tablosundaki satır sayısını buluyoruz. Sonra sayfalarda gösterilecek satır sayısı ile bunu oranlayarak sayfa sayısını buluyoruz. Sayfa sayısını bulurken dikkat etmemiz gereken nokta artık satırlar için sayfa numarasını bir arttırmamız gerektiğidir. Bunu tespit edebilmek için toplam satır sayısını sayfada gösterilecek satır sayısına bölerken mod operatörü (%) yardımıyla kalan değeri hesaplıyoruz. Eğer kalan değer 0 ise problem yok. Sayfa sayısı tamdır. Ancak 0 değil ise bu durumda sayfa sayısını bir arttırmalıyız ki kalan satırlarda en sondaki sayfada gösterebilelim.

Bu teknik yardımıyla sayfa sayısını tespit etmemizin ardından her bir sayfa numarası için birer LinkButton kontrolü oluşturuyoruz. Bu kontrollerin Text ve ID özelliklerine ilgili sayfa numarasını set ettikten sonra sayfadaki placeHolder kontrolüne ekliyoruz. Ayrıca LinkButton' lar arasında birer boşluk olmasını sağlamak için Label kontrollerini kullanabiliriz. Şimdi burada önemli olan nokta LinkButton nesnelerinden birisine tıklandığında ilgili sp' mizi çalıştıracak olan

metodumuzu çağırabilmek. Bunun için her bir LinkButton nesnesini döngü içerisinde oluştururken aynı Click olay metoduna yönlendiriyoruz. Bu olay metodu içerisinde yaptığımız iş ise ilgili LinkButton kontrolünün ID değerini almak ve Doldur isimli metoda göndermek. Böylece ilgili linke tıklandığında doğruca sp' miz çalıştırılacak ve ilgili sayfaya ait veri seti ekrandaki grid kontrolümüze dolacak.

```
private void SatirSayisiniBul()
{
    cmd=new SqlCommand("SELECT COUNT(*) FROM Employee",con);
    con.Open();
    int toplamSatirSayisi=Convert.ToInt32(cmd.ExecuteScalar());
    con.Close();
    ViewState.Add("TSS",toplamSatirSayisi);
}
```

```
private void LinkleriOlustur(int gosterilecekSatirSayisi)
{
    if(ViewState["TSS"]!=null)
    {
        SatirSayisiniBul();
    }

    int kalanSatirSayisi=toplamSatirSayisi%gosterilecekSatirSayisi;
    int sayfaSayisi;
    if(kalanSatirSayisi==0)
        sayfaSayisi=(toplamSatirSayisi/gosterilecekSatirSayisi);
    else
        sayfaSayisi=(toplamSatirSayisi/gosterilecekSatirSayisi)+1;

    for(int i=1;i<sayfaSayisi;i++)
    {
        LinkButton link=new LinkButton();
        Label lbl=new Label();

        link.Text=i.ToString();
        link.ID=i.ToString();
        link.Click+=new EventHandler(link_Click);
        lbl.Text=" ";
        plhLinkler.Controls.Add(link);
        plhLinkler.Controls.Add(lbl);
    }
}
```

```
private void link_Click(object sender, EventArgs e)
{
    LinkButton currLink=(LinkButton)sender;
```

```
int sayfaNo=Convert.ToInt16(currLink.ID);
Doldur(sayfaNo,5);
}
```

Son olarak sayfamız yüklenirken Load olayında olmasını istediğimiz kodları da aşağıdaki gibi ekleyelim.

```
private void Page_Load(object sender, System.EventArgs e)
{
    con=new SqlConnection("data
source=localhost;database=AdventureWorks2000;integrated security=SSPI");
    LinkleriOlustur(5);
    if(!Page.IsPostBack)
    {
        Doldur(1,5);
    }
}
```

Uygulamamızı çalıştıracak olursak sayfalar arasında başarılı bir şekilde dolaşabildiğimizi görürüz. Görüldüğü gibi özel sayfalama işlemi biraz meşakkatli bir yol gerektirse de performans açısından oldukça iyi verim sunacaktır. Buradaki performans farkını iyi anlayabilmek için normal sayfalama ve özel sayfalama tekniklerini iyice kavramak gerekir. Bir kere daha özetleyecek olursak, normal sayfalama tekniğinde her bir sayfada tüm veri seti tekrardan ilgili bağlantısız katman kontrolüne doldurulmaktadır.

Bizim kullandığımız teknikte de dikkat ederseniz buna benzer bir yaklaşım söz konusudur. Çünkü bizde sp' miz içerisinde tüm veri setini çekip bir temp tablo içerisine alıyoruz. Yanlız biz bu işlemi sql sunucusu üzerinde gerçekleştiriyoruz. Oysaki normal sayfalamada hakikaten tüm veri kümesi bağlantısız katman nesnesine doldurulmaktadır. Bizim tekniğimizde ise sadece ilgili sayfadaki belirtilen satır sayısı kadarlık bir veri seti bağlantısız katman nesnesine doldurulmaktadır. İşte aradaki en büyük fark budur. Ki bu fark bize performans sağlamaktadır. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

[Örnek Kodlar İçin Tıklayınız.](#)

Callback Tekniđi ile Asenkron Metod Yürütmek - 09 Eylül 2005 Cuma

C#, async, asynchronous programming, delegate,

Deđerli Okurlarım Merhabalar,

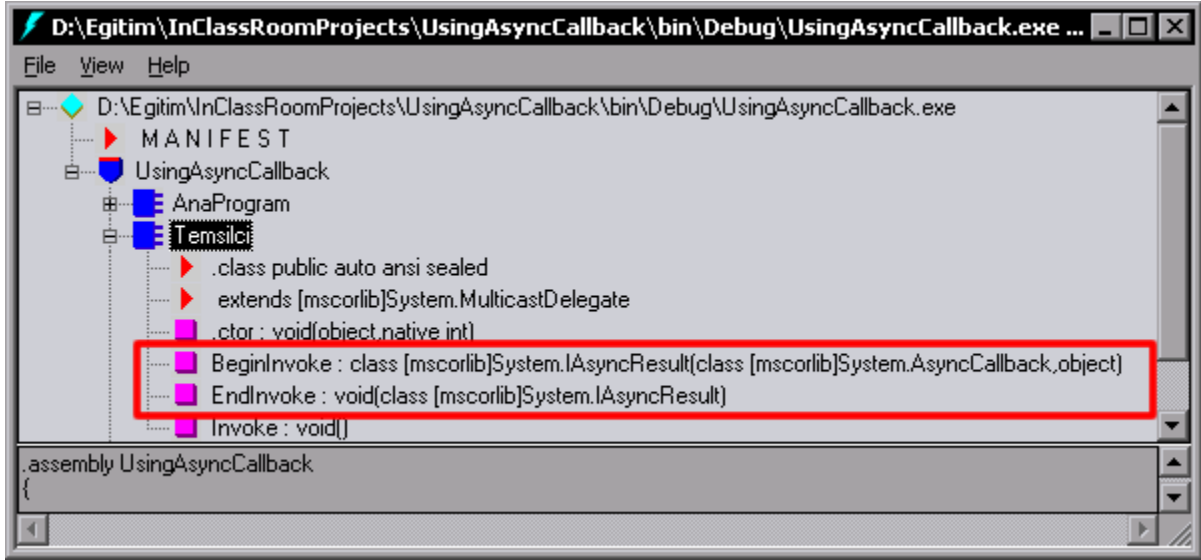
Çođu zaman projelerimizde, çalışmakta olan uygulamaları uzun süreli olarak duraksatacak işlevlere yer veririz. Özellikle görsel tabanlı uygulamalarda veritabanlarına ait kapsamlı sorguların yer aldığı işlemlerde bu sorunla sıkça karşılaşılmaktadır. En büyük problem var sayılan olarak kod satırlarının senkron hareket etmesidir. Yani kodlar sırası geldikçe işleyen parçalar bütününden oluşmaktadır. Bu elbetteki uzun sürecek bir sorgunun cevapları alınmadan izleyen kod satırlarının işlememesi anlamına gelmektedir. Oysaki kodları asenkron olarak çalıştırma şansımızda mevcuttur. Eminim ki Ado.Net 2.0' da asenkron metod yürütme tekniklerini veya asenkron web servisi uygulamalarının nasıl yazılacağını duymuşsunuzdur. Temel prensib hepsi için aynıdır. Merkezde IAsyncResult arayüzü tipinden bir nesnenin kullanıldığı temsilci (delegate) tabanlı modeller söz konusudur.

Bir temsilci her hangi bir metodun başlanıç adresini işaret eden bir tip (type) tir. Metodların başlanıç adreslerini işaret eden bir temsilci çalışma zamanında polimorfik bir yapıya sahiptir. Bu sayede yeri geldiđi zaman kendi desenine uygun her hangi bir metodun yürütülmesini sağlayabilir. İşte bu felsefeden yola çıkarak çalışma zamanında asenkron olarak yürütülecek metodlarda oluşturulabilir. Bu noktada devreye IAsyncResult arayüzü (interface) girer. Temel olarak asenkron olarak çalıştırılmak istenen metod bir temsilci vasıtasıyla yürürlüğe sokulur. Bu anda ortama IAsyncResult tipinden bir arayüz nesnesi döner. Anlık olan bu işlem nedeni ile uygulamanın geri kalan kod satırları duraksamadan işlemeye devam eder. Ancak bu sırada ilgili temsilcinin başlattığı işlemler ayrı bir thread (iş parçacığı) içerisinde yürütülmeye devam etmektedir. Peki yürütülen thread sonlandığında, üretilen sonuçlar ortama nasıl alınacaktır? İşte burada çeşitli teknikler kullanılabilir. Bizim bu makalede işleyeceğimiz olan teknik Callback modelidir.

Herşeyden önce asenkron yürütme tekniğinin kalbi olan temsilcilerin MSIL (Microsoft Intermediate Language) koduna bakmak ve anlamak gerekir. Söz gelimi aşağıdaki gibi bir temsilci tipi tanımladığımızı düşünelim. Bu son derece yalın bir delegate tanımlamasıdır.

```
public delegate void Temsilci();
```

Bu temsilciyi kullandığımız her hangi bir uygulamaya ait assembly'ı ILDASM (Intermediate Language DisAssembly) aracı yardımıyla açarsak IL kodu içerisinde temsilci nesnemize ait iki metod tanımlı olduğunu görürüz.



BeginInvoke ve EndInvoke metodları tamamıyla asenkron işlemler için geliştirilmiştir. BeginInvoke metodu, temsilcimizin çalışma zamanında işaret ettiği metodu yürürlüğe sokmak ve o anda ortama bir IAsyncResult arayüzü nesne örneği göndermek ile yükümlüdür. BeginInvoke metodu, temsilci nesnesinin işaret etmiş olduğu metodu ayrı bir thread içerisine atar. EndInvoke metodu ise dikkat edecek olursanız parametre olarak IAsyncResult arayüzü tipinden bir nesne örneğini alır. İşte bu nesne, BeginInvoke ile başlatılan process' ten sorumlu olan nesnedir. Dolayısıyla EndInvoke metodu içerisinden, asenkron metodun yer aldığı process' e ait sonuçlar yakalanıp ortama aktarılabilir.

Callback modelinde BeginInvoke metodunun AsyncCallback temsilci tipinden olan parametresi kullanılır. Bu nesne asenkron olarak çalıştırılan metoda ait işlemler sonlandığında otomatik olarak devreye girecek metodu işaret eden temsilci tipinden başka bir şey değildir. En genel kullanımda bu AsyncCallback temsilcisi EndInvoke metodunu içeren başka bir metodu temsil eder.



Asenkron Callback modelinde, işlemlerin sonuçlanmasının hemen ardından devreye girecek olan metod Callback metodu olarak adlandırılır. Bu metodu çalışma zamanında işaret edebilmek için özel AsyncCallback temsilci (delegate) tipinden faydalanılır.

Şimdi en basit haliyle Callback modelini masaya yatıralım. Basit olarak aşağıdaki Console uygulamasını geliştireceğiz.

```
using System;

namespace UsingAsyncCallback
{
    public delegate void Temsilci();

    class Yurutucu
```

```

{
    public void Calistir(Temsilci t)
    {
        t.BeginInvoke(new AsyncCallback(SonuclariAl),t);
    }

    public void SonuclariAl(IAsyncResult ia)
    {
        Temsilci t=(Temsilci)ia.AsyncState;
        t.EndInvoke(ia);
    }

    public void Islemler()
    {
        Console.WriteLine("Async yürütülecek metod");
    }
}

class AnaProgram
{
    [STAThread]
    static void Main(string[] args)
    {
        Yurutucu yrtc=new Yurutucu();

        #region Asenkron kullanıldığında

        Temsilci t=new Temsilci(yrtc.Islemler);
        yrtc.Calistir(t);
        Console.WriteLine("Diğer kod satırları...");

        #endregion

        #region Asenkron kullanılmadan

        // yrtc.Metod();
        // Console.WriteLine("Diğer kod satırları...");

        #endregion

        Console.ReadLine();
    }
}

```

İlk olarak Temsilci isimli bir delegate tipi tanımladık. Bu tip çalışma zamanında her hangi bir parametre almayan ve geri dönüş tipi bulunmayan (void) metodları işaret edebilecek cinstedir. Asenkron işlemlerin toplu olarak yer aldığı Yurutucu isimli bir sınıfımız var. Bu sınıf

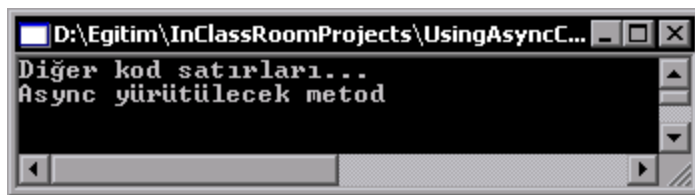
içerisinde hem asenkron metodun başlatılmasını sağlayacağız, hem de işlemlerin sonlanmasının ardından devreye girecek Callback metodumuzu işleteceğiz.

Temsilcimizin BeginInvoke ve EndInvoke metodlarını çalıştıracak iki metodumuz var. İşlemler isimli metodumuz asenkron olarak yürütmeyi düşündüğümüz metod olacak. Calistir isimli metodumuz Temsilci tipinden bir parametre almakta ve içeride bu nesne üzerinden BeginInvoke metodunu çağırılmaktadır.

BeginInvoke metodumuzun AsyncCallback temsilcisi tipinden olan parametresi İşlemler isimli metodumuzu işaret edecek şekilde tanımlanmıştır. Bu şu anlama gelmektedir. Calistir isimli metod uygulandığında BeginInvoke, temsilcinin işaret ettiğim metodu asenkron olarak yürürlüğe sokacaktır. Asenkron olarak çalışan metod sonlandığında ise AsyncCallback nesnesinin işaret ettiği SonuclariAl isimli metod otomatik olarak devreye girecektir.

SonuclariAl isimli metodumuz geriye bir değer döndürmez ve sadece IAsyncResult arayüzü tipinden bir parametre alır. Bunun sebebi AsyncCallback temsilcisinin bu tipteki metodları işaret edebilecek olmasıdır. SonuclariAl isimli metodumuz içerisinde EndInvoke metodu ile asenkron işlemlerin sonucu ortama alınmak istenmektedir. Bunu sağlayabilmek için IAsyncResult tipinden parametre Temsilci tipine dönüştürülür ki bu sayede temsilci üzerinden BeginInvoke çağırılabilir. Böylece o ana kadar çalışmış olan işlemlerin sonucunu alabilecek konuma gelmiş oluruz.

Model ilk bakışta karışık gözükebilir. Ancak temel noktaları anladığınızda sorun kalmayacaktır. Biz temsilcimizin BeginInvoke metodu ve EndInvoke metodunun olduğunu biliyoruz. Bu metodları kullanırken AsyncCallback temsilci tipi ile Callback fonksiyonu olarak EndInvoke metodunu çağırdığımız bir metodu işaret ediyoruz. Burada dikkat etmemiz gereken AsyncCallback temsilcisinin geri dönüş tipi olmayan ve IAsyncResult arayüzü tipinden nesneleri parametre olarak alan metodları işaret edebilecek olmasıdır. Bundan sonra tek yapmamız gereken çalışma zamanında temsilcimizi işaret edeceği asenkron metodu gösterecek şekilde oluşturmak ve BeginInvoke yapısını barındıran metodu yürürlüğe sokmak. Uygulamamızı yukarıdaki hali ile çalıştırdığımızda aşağıdakine benzer bir ekran görüntüsü elde ederiz.



Şimdi burada anlamamız gereken nokta şudur. Biz temsilcimiz ile işaret ettiğimiz metodu çalıştırdıktan sonra normal şartlar altında bu metod sonlanıncaya kadar kodun beklemesi ve kod tamamlandıktan sonra kalan satırların işlemesi gerekirdi. Oysaki biz metodumuzu asenkron olarak yürüttüğümüzden, temsilci nesneminin işaret ettiği metodu çalıştırdığımız kod satırından sonraki satırlar

daha önceden çalışabilmiştir. Bu ayırımı anlamak çok önemlidir. Olayı daha iyi kavrayabilmek için Main metodundaki kodları aşağıdaki gibi düzenleyelim.

```
Yurutucu yrtc=new Yurutucu();

#region Asenkron kullanıldığında

// Temsilci t=new Temsilci(yrtc.Islemler);
// yrtc.Calistir(t);
// Console.WriteLine("Diğer kod satırları...");

#endregion

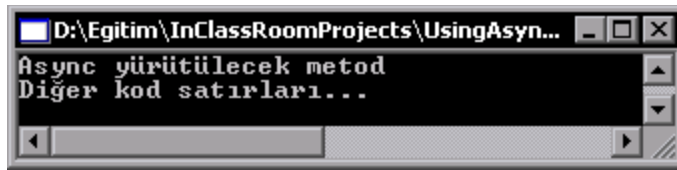
#region Asenkron kullanılmadan

yrtc.Islemler();
Console.WriteLine("Diğer kod satırları...");

#endregion

Console.ReadLine();
```

Uygulamayı tekrar çalıştırdığımızda aşağıdaki sonucu elde ederiz.



Gördüğünüz gibi önce Islemler isimli metodumuz çalıştı. Bu metodun çalışması bittikten sonra kalan kod satırlarından uygulama çalışmaya devam etti. Bunun sebebi işleyişin senkron olarak yürümesidir.

Callback modeli özellikle veri çekme işlemi uzun süren sorguların yer aldığı uygulamalarda oldukça işe yaramaktadır. Örneğin AdventureWorks2000 veritabanı üzerinde aşağıdaki gibi bir sorgunun kullanıldığı bir uygulama geliştirdiğimizi düşünelim.

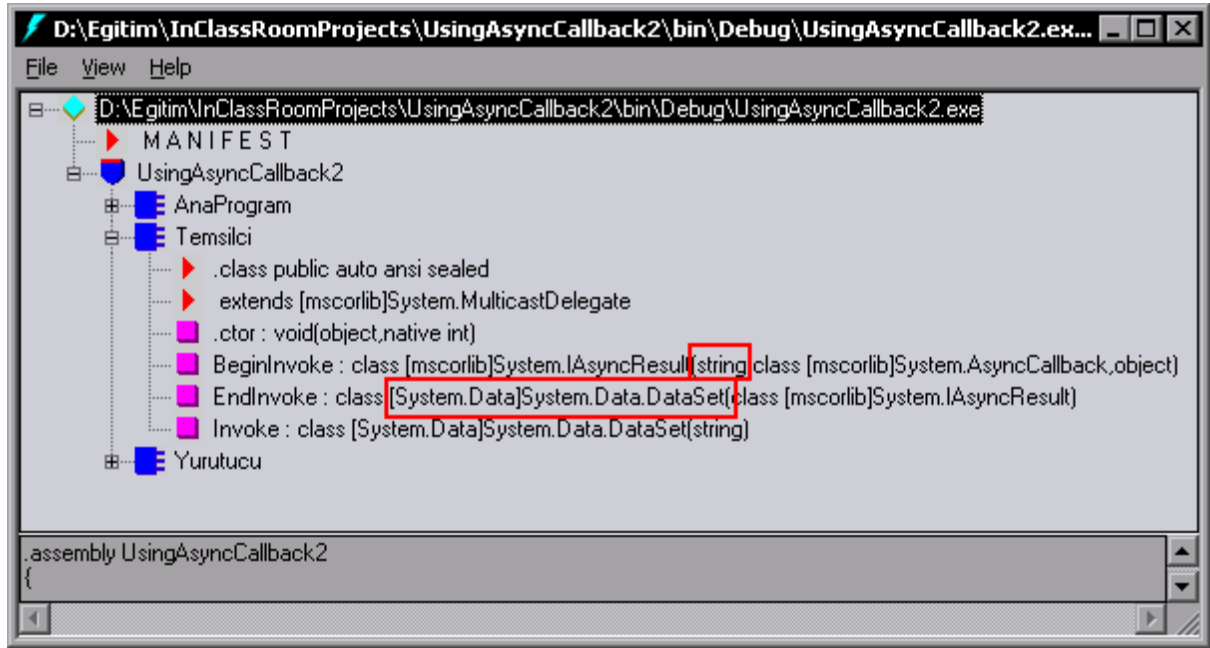
```
SELECT * FROM Customer
    INNER JOIN CustomerAddress ON Customer.CustomerID =
CustomerAddress.CustomerID
    INNER JOIN Address ON CustomerAddress.AddressID = Address.AddressID
    INNER JOIN SalesPerson ON Customer.SalesPersonID = SalesPerson.SalesPersonID
    INNER JOIN SalesPersonQuotaHistory ON SalesPerson.SalesPersonID =
SalesPersonQuotaHistory.SalesPersonID
```

Gerçek hayatta bu ve bunun gibi daha çok zaman alacak sorguları sıkça kullanıyoruz. Yukarıdaki sorgu her ne kadar bizim için pek bir şey ifade etmesede örnek uygulamamızda

Callback modelini nasıl kullanacağımıza dair uygun bir gecikme süresi sağlayacak yapıdadır. Uygulamamızın deseni yukarıdaki ile neredeyse aynı olacak. Ancak bu kez önemli bir fark var. Bu da asenkron olarak yürütülecek metodumuzun DataSet döndüren ve string parametre alan bir yapıda olması. Dolayısıyla bu metodu işaret edecek temsilci nesnemizde aşağıdaki gibi olmalıdır.

```
public delegate DataSet Temsilci(string sorguCumlesi);
```

Eğer ILDASM aracı ile bu temsilcinin MSIL koduna bakacak olursak BeginInvoke ve EndInvoke metodlarının bir önceki örneğimize nazaran biraz daha farklı oluşturulduğunu görürüz.



Dikkat ederseniz temsilcimizin string parametresi BeginInvoke metodunun ilk parametresidir. Ayrıca temsilcimizin dönüş tipide EndInvoke metodunun dönüş tipi olmuştur (DataSet) .



Asenkron yürütülecek olan metodlarımız geri dönüş tipine ve parametrelere sahip ise, BeginInvoke ve EndInvoke metodlarında bu yapıya göre şekillenecektir. EndInvoke asenkron metodun dönüş tipinden değer döndürürken, BeginInvoke asenkron metodda tanımlanan parametre sayısı kadar parametreyi ek olarak alacaktır.

Dolayısıyla hem çalıştırıcı metodumuzu hem de Callback metodumuzu bu yapıya uygun olarak tasarlamalıyız. İşte örnek kodlarımız.

```
using System;  
using System.Data;  
using System.Data.SqlClient;  
using System.Threading;
```

```

namespace UsingAsyncCallback2
{
    public delegate DataSet Temsilci(string sorguCumlesi);

    class Yurutucu
    {
        public DataSet ds;

        public void Baslat(Temsilci t,string sorgu)
        {
            t.BeginInvoke(sorgu,new AsyncCallback(Bitir),t);
        }

        public void Bitir(IAsyncResult ia)
        {
            Temsilci t=(Temsilci)ia.AsyncState;
            ds=t.EndInvoke(ia);
            Console.WriteLine(ds.Tables[0].Rows[1][5]+" "+ds.Tables[0].Rows[1][6]+"
"+ds.Tables[0].Rows[1][7]);
        }

        public DataSet SonuclariAl(string sorgu)
        {
            SqlConnection con=new SqlConnection("data
source=localhost;database=AdventureWorks2000;user id=sa");
            SqlDataAdapter da=new SqlDataAdapter(sorgu,con);
            DataSet dsSonucKumesi=new DataSet();
            da.Fill(dsSonucKumesi);
            return dsSonucKumesi;
        }
    }

    class AnaProgram
    {
        [STAThread]
        static void Main(string[] args)
        {
            Yurutucu yrtc=new Yurutucu();
            string sorgu=@"SELECT * FROM Customer INNER JOIN CustomerAddress ON
Customer.CustomerID = CustomerAddress.CustomerID INNER JOIN Address ON
CustomerAddress.AddressID = Address.AddressID INNER JOIN SalesPerson ON
Customer.SalesPersonID = SalesPerson.SalesPersonID INNER JOIN
SalesPersonQuotaHistory ON SalesPerson.SalesPersonID =
SalesPersonQuotaHistory.SalesPersonID";
            Temsilci t=new Temsilci(yrtc.SonuclariAl);
            yrtc.Baslat(t,sorgu);
            for(int i=1;i<3000;i++)
            {

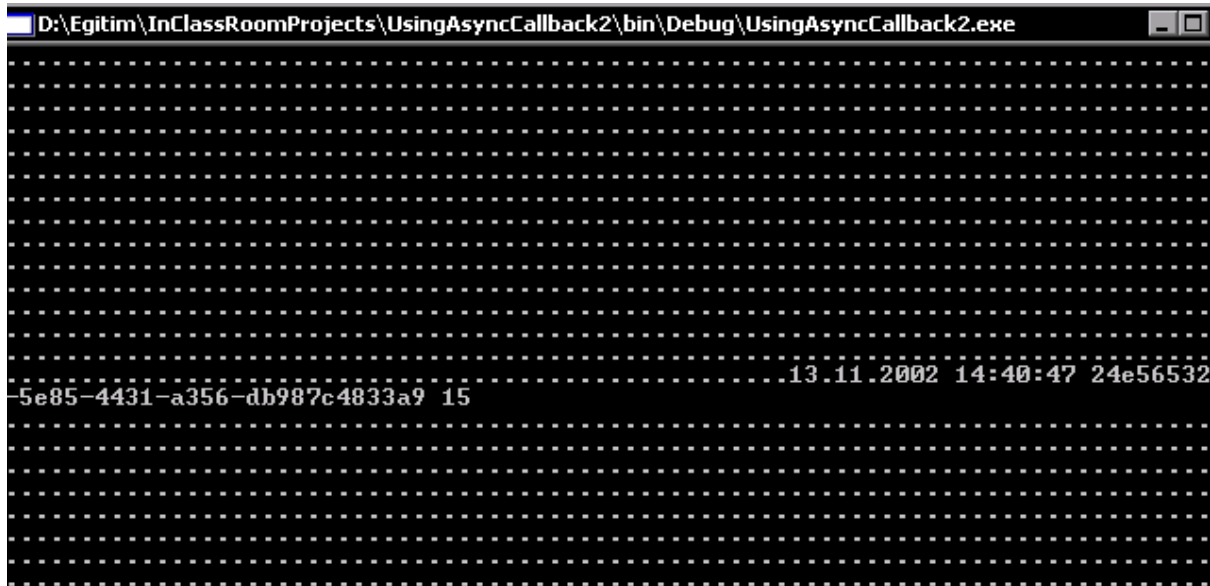
```

```

        Console.WriteLine(".");
    }
    Console.ReadLine();
}
}
}

```

Uygulamamızı çalıştırdığımızda yaklaşık olarak aşağıdakine benzer bir görüntü ile karşılaşırız.



Dikkat ederseniz for döngüsü başlamadan önce asenkron olarak yürütmek istediğimiz metodumuz olan SonuclariAl temsilci nesnemiz yardımıyla çalıştırılmıştır. Bu işlemin ardından BeginInvoke metodumuz asenkron olarak çalışan SonuclariAl metodunu ayrı bir iş parçasına bir IAsyncResult arayüzü nesnesi sorumluluğunda devretmiştir. BeginInvoke metodunu çalıştırdığımızda Callback metodumuzda bildirdiğimizden sorgu sonuçlandığı anda EndInvoke metodunun yer aldığı Bitir isimli metod devreye girecektir. Bu metod içerisinde tamamlanan asenkron process' ten sorumlu IAsyncResult nesne örneği kullanılarak Temsilci nesnemiz elde edilmiş ve bu nesne üzerinden EndInvoke çağırılarak sorgu sonucu elde edilen DataSet alınmıştır. Dikkat edin BeginInvoke metodu geriye bir DataSet nesne örneği döndürmektedir. Son olarak örnek olması açısından veri kümesinin ilk satırına ait bir kaç alan bilgisi ekrana yazdırılır.

Burada önemli olan nokta, metod çalışmaya başladıktan sonra, metodun içerdiği sorgunun sonlanması beklenmeden for döngüsünün devreye girmesidir. For döngüsü işleyişini sürdürürken, asenkron metodumuz tamamlandığında sonuçlar ekrana yazdırılmış ve for döngüsü kaldığı yerden işleyişine devam etmiştir. İşte asenkron Callback modeli.

Callback modeli yardımıyla asenkron işlemlerin gerçekleştirilmesi özellikle profesyonel uygulamalarda büyük önem arz etmektedir. Özellikle uygulamaların uzun süre duraksamasını (donmasını) engelleyebileceğimiz bir yoldur. Ancak asenkron modellerde birbirlerini etkileyebilecek işlemler varsa dikkatli davranılmalıdır. Aksi takdirde process'lerin sonsuz döngülere girerek asılı kalmasına neden olabiliriz.

Böylece geldik bir makalemizin daha sonuna bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

[Örnek Kodlar İçin Tıklayınız.](#)

Boxing ve Unboxing Performans Kritiği - 26 Eylül 2005 Pazartesi

C#, boxing, unboxing,

Değerli Okurlarım Merhabalar,

Bundan yaklaşık olarak iki sene önce boxing ve unboxing kavramları ile ilgili bir [makale](#) (30.12.2003) yazmıştım. Aradan uzun süre geçti. Ancak boxing ve unboxing kavramları ile ilgili olarak halen daha dikkat edilmesi gereken hususlar var. Bunlardan bizim için en önemlisi elbetteki performans üzerine etkileri. Uygulamalarımızda çok sık olarak farkında olmadan veya farkında olarak boxing ve unboxing işlemlerinin yer aldığı kod parçalarını kullanıyoruz.

Bildiğiniz gibi boxing, bir değer türünün, referans türünü atanması sırasında gerçekleşen işleme verilen isimdir. Unboxing ise bunun tam tersi olmakta ve referans türünün tekrar değer türüne dönüştürülmesini kapsamaktadır. Hangisi olursa olsun, değer türlerinin tutulduğu stack bellek bölgesi ile, referans türlerinin tutulduğu heap bellek bölgesi arasında yer değiştirme ve kopyalama işlemleri söz konusudur.



İster boxing ister unboxing işlemi söz konusu olsun, bellek üzerinde stack ve heap bölgeleri arasında yeniden adresleme ve kopyalama işlemi söz konusudur.

İşte bu adresleme ve kopyalama işlemlerinin uygulama içerisinde sayısız defa tekrar ediyor olması performansı olumsuz yönde etkileyen en önemli nedenlerden birisidir. Bunu daha iyi anlamadan önce, boxing ve unboxing işlemlerini biraz daha alt seviyede incelemek gerekir. Çok basit olarak aşağıdaki console uygulamasının MSIL (Microsoft Intermediate Language) koduna bir göz atalım.

```
using System;
```

```
namespace InvestigateOfBoxingUnBoxing
```

```
{
```

```
    class Class1
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            int deger=25;
```

```
            object obj=deger; // Boxing
```

```
            int sonuc=(int)obj; // Unboxing
```

```

    }
}
}

```

İlk olarak integer tipinden bir değer türünü, object tipinden bir referans türünü atıyoruz. Daha sonra ise artık kutulanmış (boxing işlemine tabi tutulmuş) referans türünün değerini bilinçli bir tip dönüşüm (explicitly cast) işlemi ile tekrar değer türünden bir değişkene atıyoruz. Bu kodun IL çıktısı aşağıdaki gibi olacaktır.

```

.method private hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size 19 (0x13)
    .maxstack 1
    .locals init ([0] int32 deger,
        [1] object obj,
        [2] int32 sonuc)
    IL_0000: ldc.i4.s 25
    IL_0002: stloc.0
    IL_0003: ldloc.0
    IL_0004: box [mscorlib]System.Int32
    IL_0009: stloc.1
    IL_000a: ldloc.1
    IL_000b: unbox [mscorlib]System.Int32
    IL_0010: ldind.i4
    IL_0011: stloc.2
    IL_0012: ret
} // end of method Class1::Main

```

Gördüğünüz gibi, IL_0004 ve IL_000b segmentlerinde **box** ve **unbox** komutları çalıştırılarak referans ve değer türleri arası bellek konumlandırma ve kopyalama işlemleri yapılmıştır. Bunun zaten böyle olacağını kodlarımızdan da biliyoruz. **Peki IL kodunun bu çıktısının bizim için önemi nedir?** Herşeyden önce bunu aşağıdaki masumane kod parçasını ele alarak anlamaya çalışmakta fayda var.

```
using System;
```

```

namespace InvestigateOfBoxingUnBoxing
{
    class BoxUnBox
    {
        public static void EkranaYaz(int yaricap,double pi)
        {
            double alan=yaricap*yaricap*pi;
            Console.WriteLine("Yaricapi {0} olan dairenin alanı = {1} dir.",yaricap,alan);
        }
    }
}
class Class1

```

```

{
    static void Main(string[] args)
    {
        BoxUnBox.EkranaYaz(10,3.14);
    }
}

```

Bu örnekte odaklanmamız gereken yer BoxUnBox sınıfımız içerisinde int tipinden yarıçap değerini ve double tipinden pi değerini alan EkranaYaz isimli metodudur. Bu metod içerisinde standart olarak alan hesabını yaptıktan sonra sonuçları ekrana yazdırmak için Console sınıfının WriteLine metodunu kullanıyoruz. Şimdi uygulamanın IL koduna tekrar bakalım. EkranaYaz metodunun içerisinde yer alan aşağıdaki satırlar bizim için oldukça önemlidir.

```

IL_000c: ldarg.0
IL_000d: box [mscorlib]System.Int32
IL_0012: ldloc.0
IL_0013: box [mscorlib]System.Double
IL_0018: call void [mscorlib]System.Console::WriteLine(string, object, object)

```

Gördüğümüz gibi değer türlerimiz boxing işlemine tabi tutulmuş ve WriteLine metoduna object tipinden geçirilmiştir. Halbuki biz kodumuzda basit olarak sonuçları ekrana yazdırmaya çalışıyoruz. Şu noktada bellek üzerinde, stack ve heap arasında bir veri değiş tokuşu olacağını düşünmeyebiliriz. Ancak IL kodlarının da söylediği gibi box ve unbox komutları çağırılmıştır. Oysaki aynı kodu aşağıdaki stilde yazsaydık eğer;

```

public static void EkranaYaz(int yarıcap,double pi)
{
    double alan=yarıcap*yarıcap*pi;
    Console.WriteLine("Yarıcapı {0} olan dairenin alanı = {1} dır.",yarıcap.ToString(),alan.ToString());
}

```

bu durumda değer türlerimiz için bir boxing işlemi uygulanmayacaktı. Aslında WriteLine metodunun beklediği object türünden bir atama söz konusudur. Biz bunu daha parametreyi geçirirken değer türünün ToString() metodu ile sağlamış oluyoruz. Dolayısıyla, IL kodlarına tekrar bakacak olursak box ve unbox komutlarının çağırılmadığını, bir başka deyişle boxing ve unboxing işlemlerinin yapılmadığını görürüz.

```

IL_000c: ldarga.s yarıcap
IL_000e: call instance string [mscorlib]System.Int32::ToString()
IL_0013: ldloc.s alan
IL_0015: call instance string [mscorlib]System.Double::ToString()
IL_001a: call void [mscorlib]System.Console::WriteLine(string, object, object)

```

Gördüğümüz gibi her hangibir box komutu çağırılmamıştır. İyi herşey hoşta, aynı sonuçları elde ettiğimiz her iki kod örneğinden hangisini tercih etmeliyiz. Bu durumu analiz edebilmek için aşağıdaki örnek uygulamayı göz önüne almakta fayda var. Amacımız boxing uygulandığı

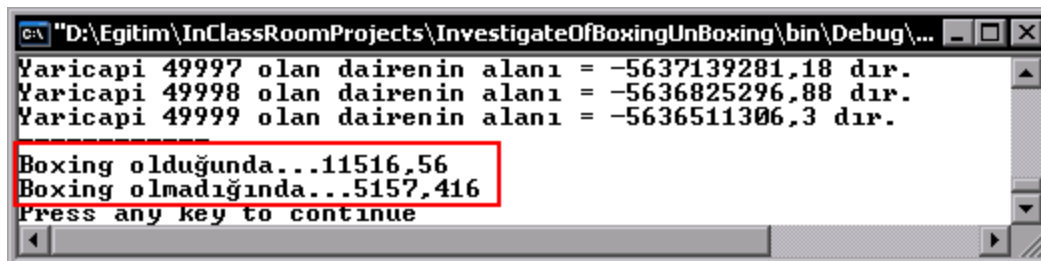
ve uygulanmadığı durumlarda süresel farkları tespit ederek performans değerlendirmesi yapabilmek.

```
static void Main(string[] args)
{
    #region Boxing içeren kod kısmı
    DateTime suAn=DateTime.Now;
    for(int i=1;i<50000;i++)
    {
        double alan=i*i*3.14;
        Console.WriteLine("Yaricapi {0} olan dairenin alanı = {1} dır.",i,alan); // Boxing var...
    }
    TimeSpan tsBox=DateTime.Now-suAn;
    #endregion

    #region boxing içermeyen kod kısmı
    suAn=DateTime.Now;
    for(int i=1;i<50000;i++)
    {
        double alan=i*i*3.14;
        Console.WriteLine("Yaricapi {0} olan dairenin alanı = {1} dır.",i.ToString(),alan.ToString()); // Boxing yok...
    }
    TimeSpan tsNoBox=DateTime.Now-suAn;
    #endregion

    Console.WriteLine("-----");
    Console.WriteLine("Boxing olduğunda..." + tsBox.TotalMilliseconds.ToString());
    Console.WriteLine("Boxing olmadığında..." + tsNoBox.TotalMilliseconds.ToString());
}
```

Uygulama kodumuz her ne kadar anlamsız görünse de sonuç gerçekten çok ilginçtir. Uygulamanın tespit ettiği süreler aslında ortalama değerlerdir. Bu genellikle kullandığınız makinenin donanımsal yeteneklerine göre değişiklik gösterebilir. Ancak tabiki önemli olan hangisinin daha hızlı olduğudur.



```
C:\> "D:\Egitim\InClassRoomProjects\InvestigateOfBoxingUnBoxing\bin\Debug\...
Yaricapi 49997 olan dairenin alanı = -5637139281,18 dır.
Yaricapi 49998 olan dairenin alanı = -5636825296,88 dır.
Yaricapi 49999 olan dairenin alanı = -5636511306,3 dır.
Boxing olduğunda...11516,56
Boxing olmadığında...5157,416
Press any key to continue
```

Her iki region altındaki kodlarda aynı işi yapar. 50000 kez i değeri üzerinden alan hesabı yaparak, sonuçları ekrana yazar. Ancak her iki teknik arasında özellikle de WriteLine metodları içerisinde az önce bahsettiğimiz ToString() kullanımı farkı vardır. İlk region içerisindeki kodlarımızda ToString metodunu kullanmadık. Bu

sebeplede, değerler ekrana yazdırılmadan önce boxing işlemi söz konusu olacaktır. Ancak ikinci region bölgesindeki kodlarımızda yer alan WriteLine metodunda ise değer türlerimiz için ToString metodunu kullanıyoruz. Sonuçta süre farkı önemsenecek derecede yüksektir. İkinci teknik daha hızlı sonuç almamızı sağlamıştır. Her ne kadar yukarıdaki gibi bir örneği pek kullanmayacak olsanızda, geniş çaplı uygulamalar düşünüldüğünde gereksiz yere yapılan boxing ve unboxing işlemleri, uygulamanın genelinde önemli oranda performans ve hız kaybına neden olabilir.

Boxing ve Unboxing işlemlerinin sık olarak görüldüğü diğer bir durum ise koleksiyonların kullanıldığı uygulamalarda göze çarpmaktadır. Özellikle koleksiyonlara eleman aktarılırken veya koleksiyon içerisindeki bir eleman okunurken boxing ve unboxing işlemleri ile karşılaşılır. Burada eleman sayısının yükselmesi, gerçekleşen boxing ve unboxing işlemlerinin sayısını arttıracaktır. Dolayısıyla stack ve heap arasındaki kopyalama ve yer değiştirme işlemleri de oldukça fazlaşacaktır ki bu da uygulamanın yavaşlamasına neden olan bir faktördür. Söz gelimi aşağıdaki örnek uygulamayı göz önüne alalım. Burada Urun isimli struct (yapı) tipinden bir nesnemizi ilk önce bir ArrayList koleksiyonunda, ardından object tipinden bir dizide ve son olarakta kendi tipinden bir dizide kullanıyoruz.

```
using System;  
using System.Collections;
```

```
namespace InvestigateOfBoxingUnBoxing  
{  
    public struct Urun  
    {  
        private int m_Fiyat;  
        public int Fiyat  
        {  
            get  
            {  
                return m_Fiyat;  
            }  
            set  
            {  
                m_Fiyat=value;  
            }  
        }  
    }  
  
    public Urun(int fiyat)  
    {  
        m_Fiyat=fiyat;  
    }  
}  
class Class1
```

```

{
    static void Main(string[] args)
    {
        #region ArrayList koleksiyonu kullanıldığında
        ArrayList alUrun=new ArrayList();
        DateTime dtSuan=DateTime.Now;
        for(int i=1;i<500000;i++)
        {
            alUrun.Add(new Urun(i*1000)); // boxing olacaktır
        }
        TimeSpan tsFark=DateTime.Now-dtSuan;
        Console.WriteLine("ArrayList Kullanımı....."+tsFark.TotalMilliseconds.ToString());
        #endregion

        #region object dizisi kullanıldığında
        object[] objUrunler=new object[500000];
        dtSuan=DateTime.Now;
        for(int i=1;i<500000;i++)
        {
            objUrunler[i]=new Urun(i*1000); // boxing olacaktır
        }
        tsFark=DateTime.Now-dtSuan;
        Console.WriteLine("Object Dizisi
Kullanımı....."+tsFark.TotalMilliseconds.ToString());
        #endregion

        #region Struct tipinden bir dizi kullanıldığında
        Urun[] urunList=new Urun[500000];
        dtSuan=DateTime.Now;
        for(int i=1;i<500000;i++)
        {
            urunList[i]=new Urun(i*1000); // değer türüne aktarma var. Yani boxing yok...
        }
        tsFark=DateTime.Now-dtSuan;
        Console.WriteLine("Struct Dizisi
Kullanımı....."+tsFark.TotalMilliseconds.ToString());
        #endregion

    }
}

```

Eğer Main metodumuzun IL koduna bakacak olursak, ArrayList koleksiyonunu ve object dizisini kullandığımız döngüler için box komutunun çağırıldığını kolayca görebilirsiniz. Bunun sebebi struct tipimizin değer türü olmasıdır. ArrayList ve object dizilerimizin elemanları ise object tipinden bir başka deyişle referans türündendir. Dolayısıyla ArrayList' e ve object tipinden olan dizimize, Urun isimli struct' ımıza ait nesne örneklerini eklemeye çalıştığımızda,

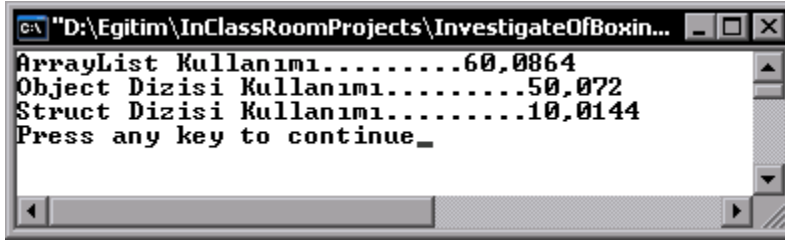
değer türünden referans türüne geçiş (boxing) işlemi söz konusu olacaktır. Tahmin edeceğimiz üzere bu performans ve hız kaybına neden olan bir durumdur.

```
Class1::Main : void(string[])
newobj instance void InvestigateOfBoxingUnBoxing.Urun::.ctor(int32)
box InvestigateOfBoxingUnBoxing.Urun
callvirt instance int32 [mscorlib]System.Collections.ArrayList::Add(object)
pop
ldloc.2
ldc.i4.1
add
stloc.2
ldloc.2
ldc.i4 0x7a120
blt.s IL_0010
call valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_
ldloc.1
call valuetype [mscorlib]System.TimeSpan [mscorlib]System.DateTime::op_

stloc.3
ldstr bytearray (41 00 72 00 72 00 61 00 79 00 4C 00 69 00 73 00 // A.
74 00 20 00 4B 00 75 00 6C 00 6C 00 61 00 6E 00 // t.
31 01 6D 00 31 01 2E 00 2E 00 2E 00 2E 00 2E 00 // 1.
2E 00 2E 00 2E 00 2E 00 ) // ..

ldloc.s tsFark
call instance float64 [mscorlib]System.TimeSpan::get_TotalMilliseconds(
stloc.s CS$000000002$00000000
ldloc.s CS$000000002$00000000
call instance string [mscorlib]System.Double::ToString()
call string [mscorlib]System.String::Concat(string,
string)
call void [mscorlib]System.Console::WriteLine(string)
ldc.i4 0x7a120
newarr [mscorlib]System.Object
stloc.s objUrunler
call valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_
stloc.1
ldc.i4.1
stloc.s U_5
br.s IL_0093
ldloc.s objUrunler
ldloc.s U_5
ldloc.s U_5
ldc.i4 0x3e8
mul
newobj instance void InvestigateOfBoxingUnBoxing.Urun::.ctor(int32)
box InvestigateOfBoxingUnBoxing.Urun
stelem.ref
```

Öyleki uygulamayı çalıştırdığımızda Urun isimli struct tipinden dizinin kullanıldığı döngünün, diğerlerine göre belirgin olarak daha hızlı çalıştığını görebiliriz.



```
C:\ "D:\Egitim\InClassRoomProjects\InvestigateOfBoxin...  
ArrayList Kullanımı.....60,0864  
Object Dizisi Kullanımı.....50,072  
Struct Dizisi Kullanımı.....10,0144  
Press any key to continue_
```

Elbette bu tip koleksiyonları kullandığınız durumlarda sadece Urun tipinden nesneler taşınacak ise, yine Urun tipinden bir nesne dizisini kullanmak en mantıklı seçimdir. Ama çoğu zaman kod yazarken bu gibi durumları gözden kaçırmaz. Burada belkide en büyük problem elimizdekiler ile tam olarak ne istediğimizi bilemememizdir. Urun tipinden bir diziye ihtiyacım var ise bir koleksiyona gerek var mıdır? Yoksa bir koleksiyonun sağladığı avantajları kullanamayacağım bir diziye tercih etmek için performansı ne kadar düşünmeliyim? vb...Bu sorulara doğru yanıtları vererek en uygun kullanımı seçebiliriz. Bu gibi kullanımlar uygulamanın pek çok yerinde var olabilir. İşte bu sebepten özellikle değer türlerini ve referans türlerini bir arada kullanırken boxing ve unboxing işlemlerini minimize edecek tekniklere gidilirse performans olarak büyük kazanımlar sağlanılabilir. İlk zamanlarda bunu aşmak için IL kodu ile biraz daha fazla haşırneşir olmamız gerekebilir. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

Constructor Initializers (Yapıcı Metod Başlatıcıları) Deyip Geçmeyin - 03 Ekim 2005 Pazartesi

C#, constructor,

Değerli Okurlarım Merhabalar,

Yapıcı metodlar nesne yönelimli programlamada çok büyük öneme sahiptir. Uygulamada oluşturduğumuz her bir nesnenin en az bir yapıcı metodu (ki bu varsayılan yapıcı metoddur) vardır. Kuşkusuz ki yapıcı metodlar (constructors), bir nesne örneğinin kapsüllediği verilere başlangıç değerlerinin atanabileceği en elverişli elemanlardır.



Yapıcıları, nesneleri başlangıç konumlarına getirmek, bir başka deyişle nesne ilk oluşturulduğunda sahip olması gereken değerleri belirlemek amacıyla kullanırız.

Uygulamalarımızda çoğunlukla yapıcı metodların aşırı yüklenmiş versiyonlarına ihtiyaç duyarız. Bu gereksinim genellikle, bir nesnenin verilerinin parametrik olarak birden fazla şekilde başlatılabileceği durumlarda oluşmaktadır. Örneğin veri işlemlerini üstlenen bir sınıfın yapıcı metodunda bu işlemler için temel teşkil edecek bir bağlantı (connection) nesnesini oluşturmaya çalıştığımızı düşünelim. En az iki versiyon kullanabiliriz. Bağlantı cümlecığının (Connection string) parametrik olarak yapıcı metoda geçirildiği bir versiyon ve varsayılan bağlantı cümlecığının kullanılacağı başka bir versiyon. Elbetteki aşırı yüklenmiş yapıcı metod versiyonlarını daha da çoğaltabiliriz. Lakin burada dikkate değer bir durum vardır. O da aşırı yüklenmiş yapıcı metodların içerideki değerlere atamaları nasıl yapacağıdır. Genellikle burada iki tip versiyon kullanılır. Acem programcıların ilk zamanlarda en çok kullandığı teknik başlangıç değer atamalarının her bir yapıcı metod içerisinde ayrı ayrı yapıldığı durumu kapsar. Diğer teknik ise this anahtar sözcüğü kullanılarak uygulanır ve değer atamaları merkezi bir yapıcı metod içerisine yönlendirilir.

Bu versiyonları daha iyi kavrayabilmek amacıyla basit bir örnek üzerinde tartışacağız. Farzedelim ki Dortgenleri temsil edecek tipte bir sınıf tasarlıyoruz. Dörtgen tipinden nesneleri temsil edecek bu sınıfın en azından en ve boy gibi iki değeri kapsülleyeceği aşıkardır. Peki dörtgen sınıfına ait nesne örnekleri kaç şekilde başlatılabilir? Başka bir deyişle bir Dortgen nesnesi oluşturulduğunda, en ve boy değişkenlerinin değerleri kaç şekilde belirlenebilir? Bu soruya cevap ararken Dortgen sınıfımızı aşağıdaki gibi tasarlayalım.

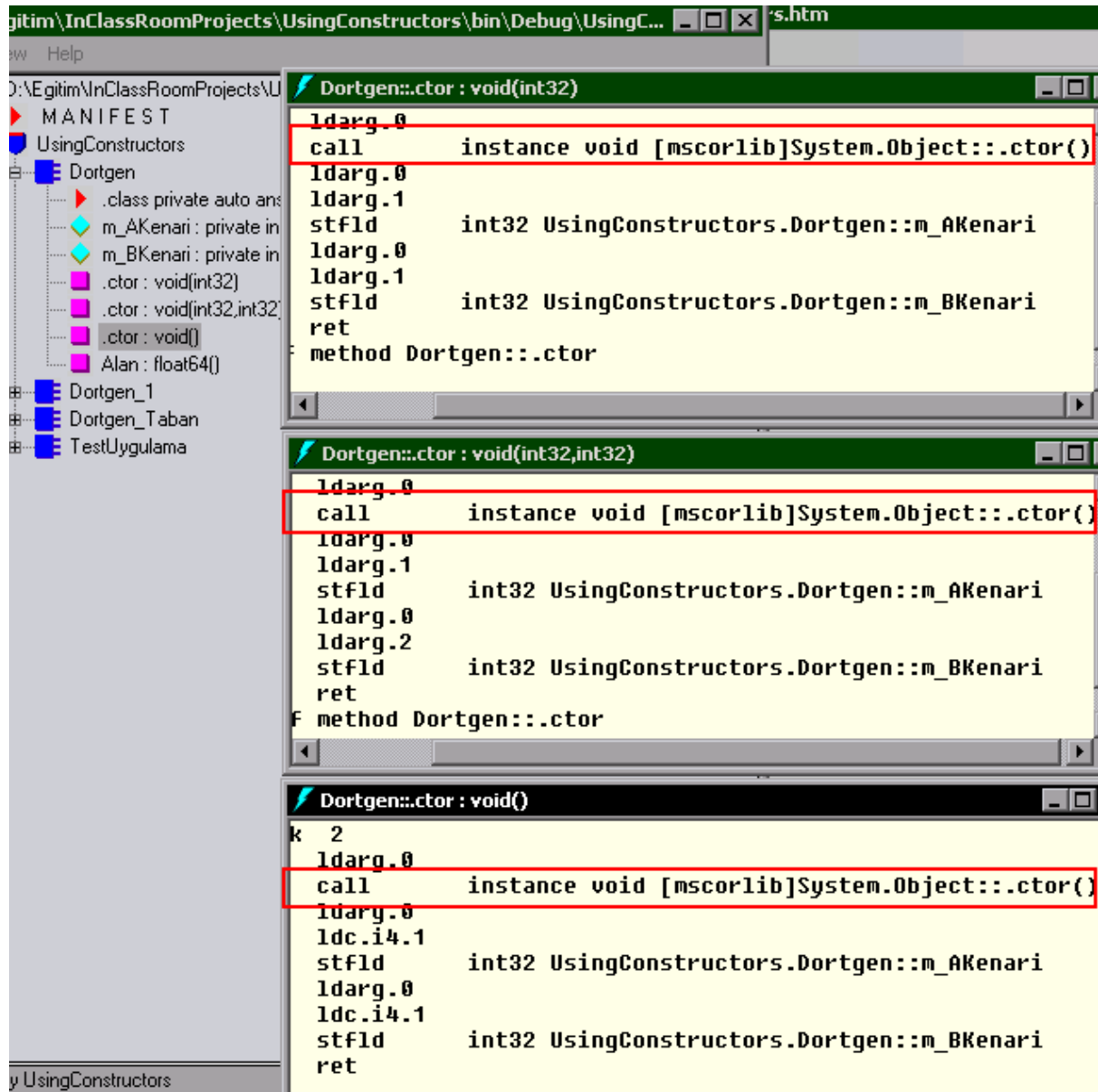
Dortgen
- m_aKenari - m_bKenari
+ Dortgen() + Dortgen(akenari: int) + Dortgen(akenari: int,bkenari: int)
+ Alan(): double

Sınıfımızın kodlarına gelince;

```
class Dortgen
{
    private int m_AKenari;
    private int m_BKenari;

    public Dortgen() // Varsayılan yapıcı (default constructor)
    {
        m_AKenari=1;
        m_BKenari=1;
    }
    public Dortgen(int akenari) // Kare olma durumu
    {
        m_AKenari=akenari;
        m_BKenari=akenari;
    }
    public Dortgen(int akenari,int bkenari) // dikdörtgen olma durumu
    {
        m_AKenari=akenari;
        m_BKenari=bkenari;
    }
    public double Alan()
    {
        return m_AKenari*m_BKenari;
    }
}
```

Gördüğümüz gibi son derece basit bir tasarımı var. Her bir yapıcı metod içerisinde, Dortgen sınıfının kapsüllediği a ve b kenarlarına ait değerlere atamalar yapıyoruz. Bu daha önceden de vurguladığımız gibi uygulamalarımızda sıkça kullandığımız bir tekniktir. Aslında durumu biraz daha derinden incelemekte fayda var. Hemen ILDASM aracı yardımıyla, bu sınıfı kullandığımız her hangibir uygulamanın intermediate language (ara dil) kodlarına bakalım.



Üç yapıcı metodumuzda kendi içlerinde `Object` tipinden bir nesne örneğini oluşturmaktadır. Aslında bu en tepedeki sınıf olan `Object` sınıfının yapıcı metoduna bir çağrıdır. Bu başka bir açıdan bakıldığında, Framework içerisinde yer alan nesne hiyerarşisinin ve kalıtımın (inheritance) bir sonucudur. Öyleki, .net bünyesinde yer alan her nesne mutlaka en tepede yer alan `Object` sınıfından türeyerek gelmektedir.



Bir yapıcı metodun IL kodu içerisinde, türediği temel sınıfa ait varsayılan yapıcı metodu çağırıyor olması, kalıtım (inheritance) için önemli bir faktördür. Bir türeyen sınıf nesne örneği oluşturulduğunda, içerideki kapsüllenmiş tiplere ilgili değerler atanmadan daha önce, temel sınıfın yapıcısı (default constructor) çağırılır. Bu mekanizma, türeyen sınıfa ait nesne örneği oluşturulurken, base anahtar sözcüğü yardımıyla ortak değişkenlere

	<i>ait değerlerin temel sınıfa(lara) kolayca aktarılabilirmelerini sağlar.</i>
--	--

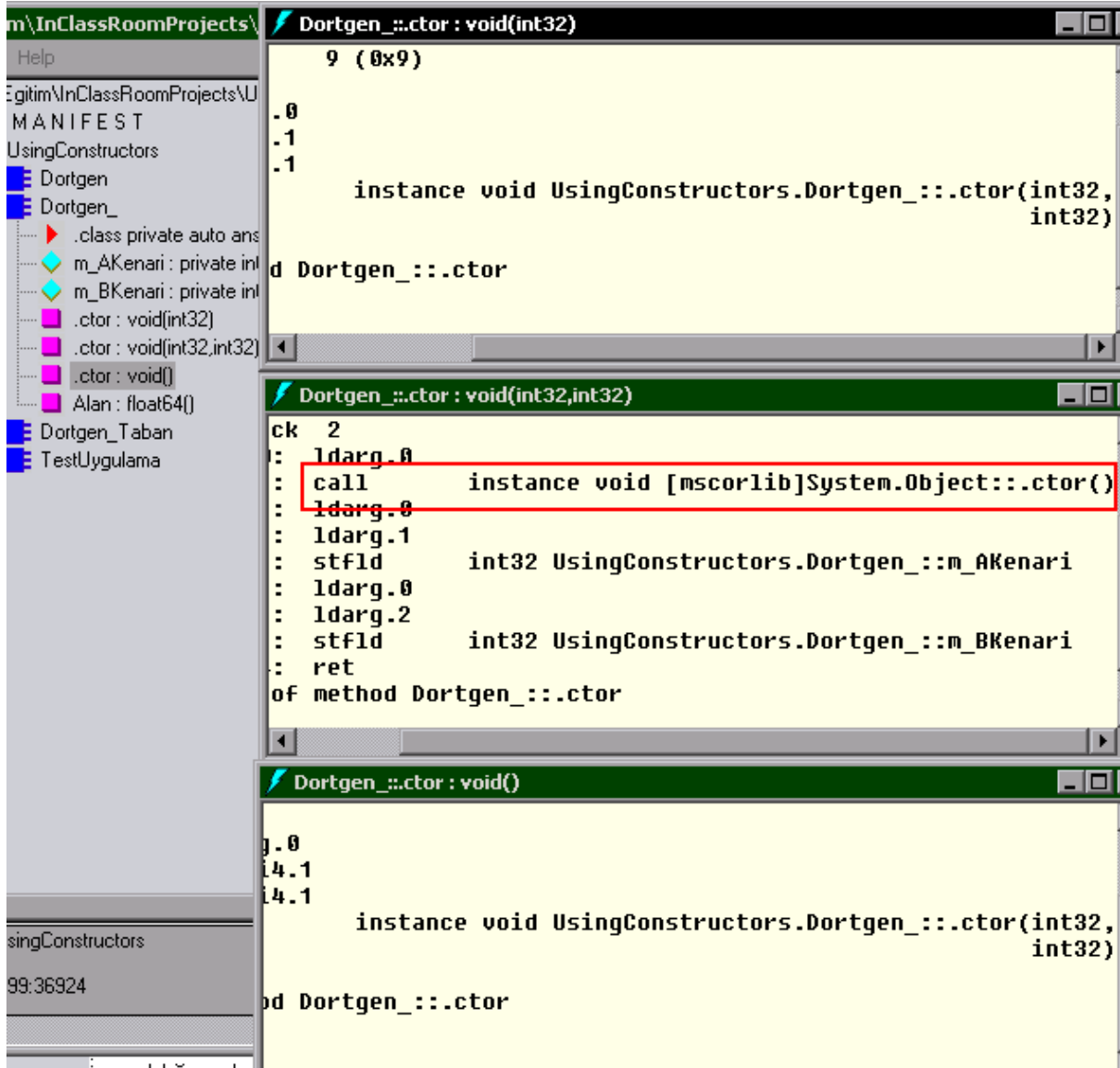
Buradaki kodlama aslında yapıcı metod başlatıcıları (constructor initializers) kullanılarak daha sade ve merkezileştirilmiş bir hale getirilebilir.

Merkezileştirilmeden kasıt, atamaların tek bir yerde toplanmasıdır. Bunun en büyük faydası kodun bakımı ve onarımı sırasında ortaya çıkar. Sonuç itibariyle yapıcı metodlarımızın ortak noktası, gelen parametreleri aynı şekilde atamaya çalışmalarıdır. Dolayısıyla Dortgen sınıfını this anahtar sözcüğünü de kullanarak aşağıdaki gibi de geliştirebiliriz .

```
class Dortgen_// :Dortgen_Taban
{
    private int m_AKenari;
    private int m_BKenari;

    public Dortgen_():this(1,1)
    {
    }
    public Dortgen_(int akenari):this(akenari,akenari)
    {
    }
    public Dortgen_(int akenari,int bkenari)
    {
        m_AKenari=akenari;
        m_BKenari=bkenari;
    }
    public double Alan()
    {
        return m_AKenari*m_BKenari;
    }
}
```

Bu sefer Dortgen sınıfının her bir yapıcı metodu aslında en genel yapıcı metodu çağırıp, this anahtar sözcüğü yardımıyla ortamdaki gelen parametreleri tek bir noktaya aktarmaktadır. Bu teknik kod okunurluğunu daha da kolaylaştırır. Ayrıca, değer atamalarının bakımını olumlu yönde etkiler. Çünkü her hangibir değişiklik için tüm yapıcı metodları gezmektense, sadece atamaların yapıldığı asıl yapıcı metodu değiştirmek çok daha akılcı bir yoldur. Ancak her iki uygulama tekniği arasındaki farklar bunlar ile sınırlı değildir. Asıl farkı görebilmek için yine IL kodlarına bakmamız gerekir.



Gördüğünüz gibi bu sefer Object sınıfına ait varsayılan yapıcı metod sadece this anahtar sözcükleri ile parametreleri yönlendirdiğimiz merkezi yapıcı metod içerisinde çağırılmaktadır. Bu kodun içerisinde çalışma zamanında object nesneleri için tahsis edilecek bellek miktarını biraz da olsa azaltan bir faktördür. Tabi durumu bir de hız açısından incelemek gerekir. Her iki teknik uygulanabilirlik, merkezileştirme, idareli bellek kullanımı açısından oldukça farklıdır aslında. Ama aşağıdaki örnek kodu uyguladığımızda çok daha farklı bir sonucu elde edeceğimizi görürüz.

```
class TestUygulama
{
    static void Main(string[] args)
    {
        DateTime dtSimdi;
        TimeSpan fark;

        #region birinci tip constructor kullanımı (initializers ile)
```

```

dtSimdi=DateTime.Now;
for(int i=1;i<500000;i++)
{
    Dortgen_ d1=new Dortgen_();
    d1.Alan();
}
fark=DateTime.Now-dtSimdi;
Console.WriteLine("Birinci tip constructor kullanımı {0} (Initializers
ile)",fark.TotalMilliseconds.ToString());

#endregion

#region ikinci tip constructor kullanımı

dtSimdi=DateTime.Now;
for(int i=1;i<500000;i++)
{
    Dortgen d2=new Dortgen();
    d2.Alan();
}
fark=DateTime.Now-dtSimdi;
Console.WriteLine("İkinci tip constructor kullanımı {0}
",fark.TotalMilliseconds.ToString());

#endregion
}
}

```

Buradaki ilk döngümüz, yapıcı başlatıcıları (constructor initializers) kullanan Dortgen_ nesnesine ait 500000 nesne örneğini oluşturur ve alan hesabı yapar. İkinci döngümüz ise, her bir değer atamasının kendi yapıcı metodu içerisinde yapıldığı ilk tekniğimizi kullanır. Uygulamayı test ettiğimizde her iki döngünün tamamlanma süreleri arasında belirgin bir fark vardır.



Sanılanın aksine this kullanılan teknik, diğerine göre daha yavaş çalışmaktadır. Bu elbetteki göz ardı edilebilecek bir süre farkıdır. Alternatif bir yöntem olarak değer atamalarının yapıldığı ortak bir metod, her bir constructor içinden ayrı ayrı çağırılabilir. Yani aşağıdaki Dortgen_2 sınıfının kodlarında olduğu gibi.

```

public class Dortgen_2
{

```

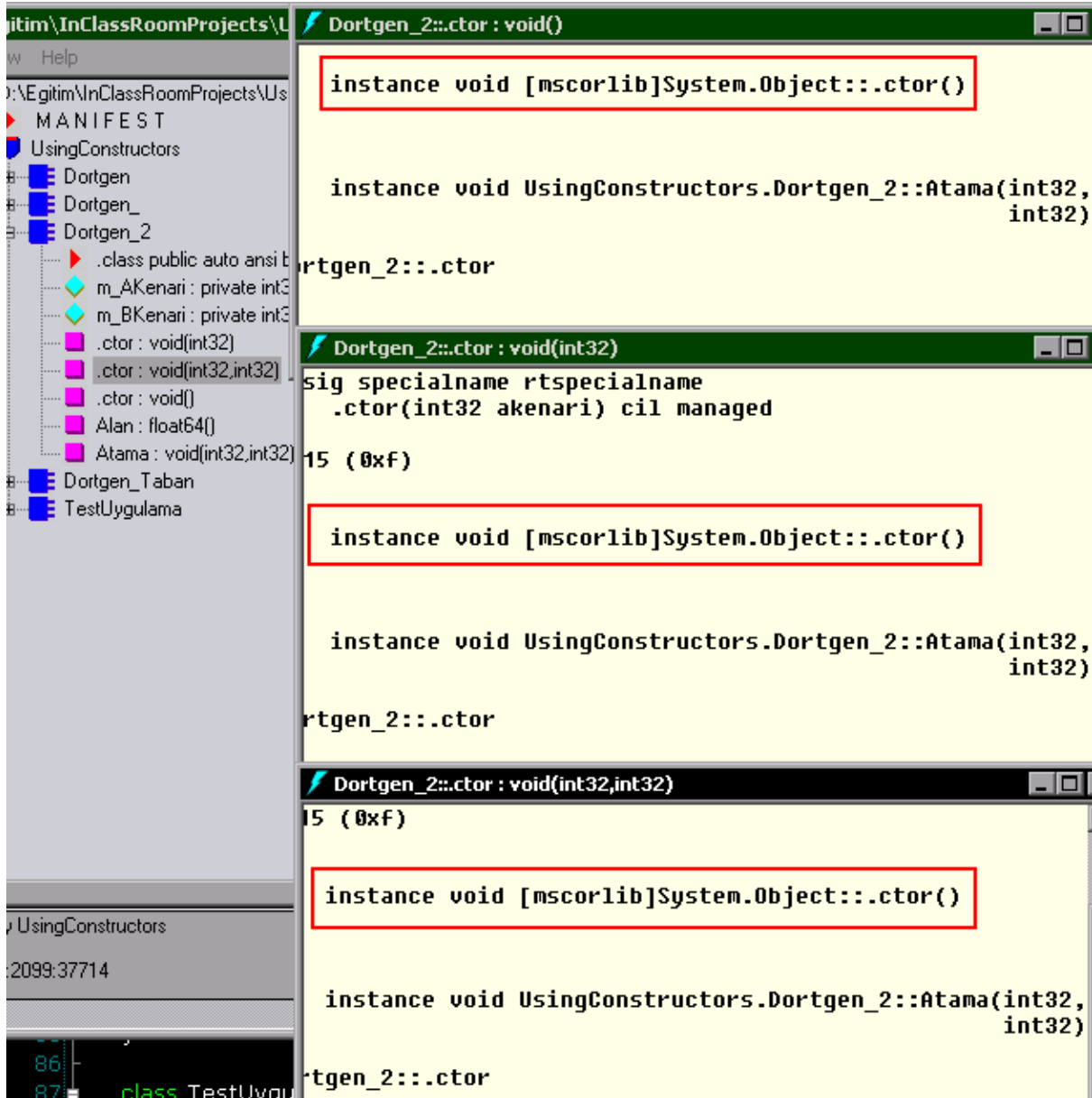
```

private int m_AKenari;
private int m_BKenari;

public Dortgen_2()
{
    Atama(1,1);
}
public Dortgen_2(int akenari)
{
    Atama(akenari,akenari);
}
public Dortgen_2(int akenari,int bkenari)
{
    Atama(akenari,bkenari);
}
public double Alan()
{
    return m_AKenari*m_BKenari;
}
private void Atama(int akenari,int bkenari)
{
    m_AKenari=akenari;
    m_BKenari=bkenari;
}
}

```

Burada atama işlemleri sadece bu sınıf içerisinde erişilebilen (private) bir metod ile sağlanmaktadır. Bu merkezileştirmeyi ve bakımın kolaylaştırılabilmesini sağlar. Ancak this kullanımı terk edildiği için IL kodunda yine her bir yapıcı metod içerisinde, Object sınıfına ait varsayılan yapıcı metodun çağırılması durumu devam etmektedir.



Elbette bu şartlar göz önüne alındığında seçim yapmak zorlaşmaktadır. Hangisi olursa olsun çalışacaktır. Ancak ben performans açısından kayba neden olsa da, merkezileştirme, bakım ve onarım kolaylığı sağladığı düşünüldüğünde yapıcı başlatıcılarını (constructor initializers) kullanmayı tercih ediyorum. Böylece geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

Ado.Net 2.0(Beta 2) - Connection String Security (Bağlantı Katarı için Güvenlik) - 17 Ekim 2005 Pazartesi

ado.net 2.0 beta, connection string security,

Değerli Okurlarım Merhabalar,

Güvenlik günümüz uygulamalarında çok önemli bir yere sahiptir. Özellikle veritabanı kullanımını içeren uygulamalarda güvenliğin ayrı bir önemi vardır. Veritabanına gönderilen sorguların korunması, özellikle web servislerinden dönen veri kümelerinin etkin olarak şifrelenmesi gibi durumlar söz konusudur. Güvenlik prosedürü içerisinde yer alan unsurlardan biriside bağlantı bilgilerinin saklanmasıdır. Biz bu makalemizde, .Net 2.0 ile birlikte gelen yeni tekniklerden birisine değinerek, bağlantı bilgisinin (çoğunlukla connection string olarakda ifade edebiliriz) kolay bir şekilde nasıl korunabileceğini incelemeye çalışacağız.

Bildiğiniz gibi, .Net tabanlı uygulamalar özellikle XML tabanlı konfigürasyon dosyalarını yoğun olarak kullanır. Varsayılan olarak windows uygulamalarında app.config veya web uygulamalarında yer alan web.config dosyaları, genel bilgilerin yer aldığı kaynaklardır. Çoğunlukla proje genelinde kullanılan bağlantı bilgilerini bu dosyalarda key-value (anahtar-değer) çiftleri şeklinde tutmayı tercih ederiz. Aslında bağlantı katarı(connection string) bilgilerini kod içerisinde de global seviyede tanımlayıp kullanabiliriz. Ancak bunun bir takım dezavantajları vardır.



Bağlantı katarı gibi sonradan (özellikle ürün canlı yayına başladıktan sonra) sıkça değişebilecek bilgileri kod içerisinde (hard-coded) tutmak güvenlik ve yönetilebilirlik açısından dezavantajlara sahiptir.

Birincisi, yazılan kodlar sonuç olarak IL dilini kullandığından ters çevrilerek açığa çıkartılabilir. (Disassembly araçları) Bu nedenle bağlantı katarı bilgiside öğrenilebilir. Diğer yandan, uygulamalar canlı hayata geçtikten sonra, kullandıkları veri tabanı sunucularının ip bilgilerinin, kullanıcı tanımlamalarının sıkça değişmesi nedeni ile bağlantı katarı bilgilerinin sıkça güncellenmesi gerekebilir. İşte bu iki nedenden ötürü bu tip bilgiler xml tabanlı konfigürasyon dosyalarında, özel şifreleme teknikleri ile tutulur.

Aşağıdaki kod parçalarında bir web uygulaması ve bir console uygulaması için kullandığımız standart konfigürasyon dosyalarında yer alan bağlantı katarı tanımlamalarını ve bu bağlantı katarları içerisindeki bilgilerin kullanımını gösteren iki örnek yer almaktadır.

Sıradan bir console uygulaması için app.config içeriği;

```
<?xml version="1.0" encoding="utf-8" ?>
  <configuration>
    <appSettings>
      <add key="conStr" value="data
source=localhost;database=AdventureWorks2000;user id=sa;password="></add>
    </appSettings>
  </configuration>
```

conStr isimli anahtarın uygulama içerisinde kullanımı;

```
using System;
using System.Data.SqlClient;

namespace UsingConnectionString
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            string
conStr=System.Configuration.ConfigurationSettings.AppSettings["conStr"].ToString();
            using(SqlConnection con=new SqlConnection(conStr))
            {
                con.Open();
            }
        }
    }
}
```

Bir web uygulamasında web.config içerisinde bağlantı katarı (connection string) için anahatar-değer (key-value) kullanımı.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="conStr" value="data source=localhost;database=AdventureWorks2000;user
id=sa;password="></add>
  </appSettings>
  <system.web>
    <!-- Diğer ayarlar-->
  </system.web>
</configuration>
```

Web.Config içerisindeki bağlantı katarı bilgisinin default.aspx içerisinde kullanımı;

```
private void Page_Load(object sender, System.EventArgs e)
```

```

{
    string
    conStr=System.Configuration.ConfigurationSettings.AppSettings["conStr"].ToString();
    using(SqlConnection con=new SqlConnection(conStr))
    {
        con.Open();
    }
}

```

Gelelim .Net 2.0' a. Yeni versiyonda özellikle bağlantı katarlarına yönelik olarak geliştirilmiş özel bir güvenlik tekniği mevcuttur. Bu tekniğin kilit noktaları olarak, konfigürasyon ayarlarına eklenen iki yeni nitelik söz konusudur. Bu nitelikler konfigürasyon **protectedData** ve **protectedDataSections**' dır. ProtectedDataSections niteliği altında sonradan şifrelenmesi düşünülen anahtar bilgileri tutulmaktadır. Bağlantı katarı için kullanacağımız anahtar bilgisini burada tutabiliriz. ProtectedData kısmında ise, ProtectedDataSections içerisinde yer alan anahtar bilgisini şifreleyecek (encryption) ve deşifre edecek (decryption) provider nesnesine ait bilgileri yer alır. Şu an için **RSAProtectedConfigurationProvider** ve **DPAPIProtectedConfigurationProvider** adında iki adet şifreleme provider' ı mevcuttur. Bu providerlar, ilgili verinin şifrelenmesinden ve özellikle çalışma zamanında kullanılırken deşifre edilmesinden sorumludur. İlk olarak konfigürasyon dosyası içerisinde bu yapıları kullanarak gerekli düzenlemeleri yapmamız gerekmektedir.

.Net 2.0 için bir console uygulamasına ait app.config konfigürasyon dosyasının örnek içeriği.

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <connectionStrings>
        <EncryptedData/>
    </connectionStrings>

    <protectedData>
        <providers>
            <add name="prvCon"
type="System.Configuration.RsaProtectedConfigurationProvider"
keyContainerName="AnahtarDeger" useMachineContainer="true"/>
        </providers>

        <protectedDataSections>
            <add name="connectionStrings" provider="prvCon" inheritedByChildren="false"/>
        </protectedDataSections>

    </protectedData>
</configuration>

```

Dikkat ederseniz protectedDataSections sekmesi, protectedData altında yer alan bir bölümdür. Providers kısmında, şifreleme işlemini yapacak olan tipe ait bilgiler yer almaktadır. Bu tipin takma adı name özelliği ile, tipin kendisi type özelliği ile, kullanılacak şifreleme

anahtarının adı ise keyContainerName özellikleri ile tutulur. ProtectedDataSections kısmında, şifrelenecek olan anahtar bilgisi yer alır. Ancak dikkat ederseniz burada bağlantı katarına ilişkin herhangi bir bilgi yoktur. Sadece şifrelemeyi yapacak provider' ın takma adını alan name özelliği vardır. Bu özellik ile, provider sekmesinde yer alan tipi işaret ederiz. Peki bağlantı katarımız nerede yer almaktadır?

Dikkat ederseniz dosyanın başında **connectionStrings** isimli takılar arasında yer alan **EncryptedData** isimli bir sekme yer almaktadır. ConnectionStrings konfigürasyon dosyalarına eklenen yeni niteliklerden birisidir ve sadece bağlantı katarı bilgisi ile ilgilidir. Biz biraz sonra yazacağımız kodları çalıştırdığımızda, bağlantı katarımıza ait bilgiler şifrelenerek bu kısım içerisine eklenecektir. Bir başka deyişle EncryptedData kısmı şifrelenen verinin taşıyıcısıdır. Burada dikkat edilmesi gereken noktalardan birisi isim uyumluluğudur. Öyleki connectionStrings takısının ismi, protectedDataSections kısmındaki anahtarın ismi ile aynı olmak zorundadır.

Bağlantı katarının connectionStrings isimli anahtara eklenmesi için ekstra kod yazmamız gerekmektedir. .Net 2.0' da System.Configuration isim alanına bir takım yeni sınıflar eklenmiştir. Biz bu sınıflar yardımıyla, konfigürasyon dosyamıza müdahale ederek, connectionStrings isimli anahtara hem değer ataması yapabilir hemde şifreleme işlemini başlatabiliriz. Bu amaçla aşağıdaki gibi bir kod yazmamız gerekmektedir. Örnek olarak bir console uygulaması göz önüne alınmıştır.

```
using System;
using System.Configuration;
using System.Collections.Generic;
using System.Text;

namespace UsingAppSettings
{
    class Program
    {
        static void Main(string[] args)
        {
            Configuration cnfg =
            ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None, "");
            cnfg.ConnectionStrings.ConnectionStrings.Add(new
            ConnectionStringSettings("conStr","data
            source=localhost;database=AdventureWorks;integrated security=SSPI"));
            cnfg.Save();
        }
    }
}
```

Çok fazla detaya girmeden yukarıdaki kodlar ile ne yaptığımızdan kısaca bahsetmek istiyorum. İlk olarak bir configuration nesnesine atama yapıyoruz. Bu atamada

ConfigurationManager sınıfının statik **OpenExeConfiguration** metodunu kullanmaktayız. Bu metod aslında belirtilen konfigürasyon dosyasını bir Configuration nesnesi olarak atamak için kullanılmaktadır. İkinci parametre konfigürasyon dosyasının yolunu belirtir. Biz aynı uygulama içerisinde olduğumuzdan burayı boş bıraktık. Daha sonra bu nesne üzerinde hareket ederek ConnectionStrings sınıfının Add metodunu çağırıyor ve bağlantı katarımızı conStr ismi ile ilgili dosyaya ekliyoruz. Tabiki bu ekleme işlemi sırasında provider' ımız devreye giriyor ve bilgiyi şifreleyerek uygulamaya ait konfigürasyon dosyasına yazıyor. Son olarak Save metodu ile konfigürasyon dosyamızı kaydediyoruz.

Normalde bu tip işlemleri 1.1 versiyonunda yapmak için, yani xml tabanlı konfigürasyon dosyasına kod bazında müdahale etmek için XmlDocument sınıfını kullanarak ayrıştırma işlemlerini (parsing) yapmamız gerekirdi. Yeni versiyonda gördüğünüz gibi bu tarz işlemleri nesne bazında kolayca yapabilmekteyiz.



Uygulamayı denediğim .Net 2.0 Beta versiyonunda, Configuration sınıfına erişebilmek için, System.Configuration isim alanını projenin referanslarına Add Reference tekniği ile açıkça eklemem gerekti. Aksi takdirde, System.Configuration üzerinden Configuration ve ConfigurationManager gibi sınıflara direkt olarak erişilemiyor.

Yazdığımız Console uygulamasını ilk çalıştırdığımızda konfigürasyon dosyalarında her hangibir değişiklik olmadığını görürüz. Özellikle uygulamanın debug klasöründe yer alan ve exe dosyamız ile aynı isme sahip konfigürasyon dosyalarında bir değişiklik olmayacaktır. Bunun sebebi, providerın şifreleme için kullanacağı bir anahtar değerin bulunmayışındır. Hatırlarsanız provider tanımında **keyContainerName** isimli bir özellik belirlemiştik. KeyContainerName özelliğine atadığımız AnahtarDeger , RS şifrelemesi için gerekli anahtar kodu taşımak için kullanılacaktır. Peki böyle bir anahtar değerini nasıl elde edebiliriz? Bunun için aspnet_regiis aracını kullanabiliriz. Aşağıdaki komut satırı yardımıyla bu işlemi gerçekleştirmekteyiz.

```
C:\ Visual Studio 2005 Command Prompt
D:\Documents and Settings\burak senyurt\My Documents\Visual Studio 2005\Projects\UsingAppSettings\UsingAppSettings>aspnet_regiis -pc "AnahtarDeger" -exp
Creating RSA Key container...
Succeeded!
```

Bu işlemi yaptığımız takdirde uygulamamızın exe kodunun bulunduğu klasörde yer alan aynı isimli konfigürasyon dosyası(UsingAppSettings.EXE.xml) içeriğinin aşağıdaki gibi yazıldığını görürüz. Özellikle EncryptedData ve EncryptedKey kısımlarına dikkat ediniz.

```
<?xml version="1.0" encoding="utf-8" ?>
  <configuration>
    <connectionStrings>
      <EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"
        xmlns="http://www.w3.org/2001/04/xmlenc#">
```

```

    <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"
  />
    <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
      <EncryptedKey Recipient="" xmlns="http://www.w3.org/2001/04/xmlenc#">
        <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
        <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
          <KeyName>Rsa Key</KeyName>
        </KeyInfo>
      <CipherData>
        <CipherValue>ElSkDt2vsIN6xk5qC1S6slkWX97Ua28KGGJTkHx500UsfMbed1TnZpQLYJaJ
        W5XJXQPe
        Y3IWm7iKdEZ+idx7THKlAx8eLf/HHDSi/HWp7sSJe6/4vcS6uQ+OyigdLpO6X3LEqoGYhwC
        NPozq7e/e
        B/P+w9pzLwivdczR1sOYMIQ=</CipherValue>
      </CipherData>
    </EncryptedKey>
  </KeyInfo>
  <CipherData>
    <CipherValue>fKJrwr1tdHNyVH9OYKIFeK7B7C1tzxp2ikrgu+KbCNBHM9fffFGt
    EIRc9n0edC8poSskU7+APE18O6SRFqXD3OG0NX+9b65OIIHAXhE HdMYAbejU
    VvqGoNw4xkisDfk/CgANrO9kST5f15g/0bH+5Cv83ptAV8mzlvzbvAzd+kpWdk
    Q0T99jmiymcwqICBExmvUhT1wSUemYMC2Rzz3
    JbJISvyqOZH9AYIxCesaeWwcyOvQfzLyHEw==</CipherValue>
  </CipherData>
</EncryptedData>
</connectionStrings>
<protectedData>
  <providers>
    <add name="prvCon"
type="System.Configuration.RsaProtectedConfigurationProvider"
keyContainerName="AnahtarDeger" useMachineContainer="true"/>
  </providers>
  <protectedDataSections>
    <add name="connectionStrings" provider="prvCon" inheritedByChildren="false"/>
  </protectedDataSections>
</protectedData>
</configuration>

```

Gördüğünüz üzere, içeride şifrelenmiş bir takım veriler bulunmaktadır. Bu veriler aslında, programatik olarak atadığımız bağlantı katarına ilişkin bilgilerdir. Peki bu bilgiyi uygulamamızda nasıl kullanabiliriz? ConfigurationManager sınıfı yardımıyla bu bağlantı bilgisini kod içerisinde okuyabilir ve ilgili bağlantı nesneleri için atayabiliriz. Aşağıdaki örnek kod parçası bu işlemin örnek olarak nasıl yapılabileceğini göstermektedir.

```

string
conStr=ConfigurationManager.ConnectionStrings["conStr"].ConnectionString.ToString();

```

```
using (SqlConnection con = new SqlConnection(conStr))
{
    // Bir takım kodlar
}
```

Yapmış olduklarımızı özetlersek; .Net 2.0, Configuration isim alanı altında yeni özelliklere sahip pek çok tip sunmaktadır. Buradaki sınıfları ve konfigürasyon dosyaları için geliştirilen yeni nitelikleri kullanarak, config dosyalarında tutulan bağlantı katarı bilgilerini yabancı gözlerden saklayabiliriz. Elbetteki, kod içerisinde ConfigurationManager üzerinde ilgili bağlantı katarı bilgisini okuyabilirsiniz. Ancak herkesin görebildiği config dosyalarından bağlantı katarı bilgisini okuyamazsınız. Tabi ne yaparsanız yapın yinede güvenliği tam olarak sağlamak söz konusu olamaz. Her zaman için en azından %1 ihtimal ile tüm sistemler güvensizdir. Böylece geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

Kendi Referans Tiplerimizi Klonlamak -

14 Kasım 2005 Pazartesi

C# 2.0, ICloneable,

Değerli Okurlarım Merhabalar,

Bu makalemizde kendi yazmış olduğumuz referans tipleri arasında yapılan atama işlemleri sırasında üyeden üyeye (Member by member) kopyalamanın nasıl yapılabileceğini incelemeye çalışacağız. Bildiğiniz gibi referans tipleri belleğin heap bellek bölgesinde tutulurlar. Bu tutuluş yapısının özellikle referans tipleri arasında yapılan atama işlemlerinde önemli bir etkisi vardır. İki referans tipi arasında bir atama işlemi söz konusu olduğunda, aslında bu referans tiplerinin heap bellek bölgesinde yer alan adresleri eşitlenmektedir. Bu eşlemenin doğal bir sonucu olarak da referans tiplerinin her hangibirisinde yapılan değişiklik diğerinde otomatikman etkileyecektir.

Ancak bazı durumlarda, özellikle kendi yazdığımız referans tiplerini kullanırken bu durumun tam tersini isteyebiliriz. Yani kendi yazmış olduğumuz bir sınıfın iki nesne örneği arasında yaptığımız atama işlemi sonrası, bu referansların birbirini etkilemelerini istemeyebiliriz. Bu durumda yazmış olduğumuz sınıfa **ICloneable** arayüzünü uygulayarak referans tipinin klonlanmasını sağlayabiliriz. Bu durumu analiz etmeden önce, referans tipleri arasında yapılam atamanın doğal sonucunu aşağıdaki örnek ile incelemeye çalışalım. Örneğimizde bir dörtgene ait kenar uzunluklarını tutacak olan Dortgen isimli bir sınıf kullanacağız. Dortgen sınıfımızın UML şeması ve kodları aşağıdaki gibidir.



Dortgen.cs

```
public class Dortgen
{
```



```

private double kenarA;
private double kenarB;

public double A
{
    get
    {
        return kenarA;
    }
    set
    {
        kenarA=value;
    }
}

public double B
{
    get
    {
        return kenarB;
    }
    set
    {
        kenarB=value;
    }
}

public Dortgen()
{
    kenarA=0;
    kenarB=0;
}

public Dortgen(double aKenari,double bKenari)
{
    kenarA=aKenari;
    kenarB=bKenari;
}

public override string ToString()
{
    string dortgenBilgi=kenarA.ToString()+" "+kenarB.ToString();
    return dortgenBilgi;
}
}

```

Şimdi Dortgen sınıfından iki nesne örneğini kullanacağımız bir console uygulamasına ait Main metodunda, aşağıdaki kod satırlarını yazalım.

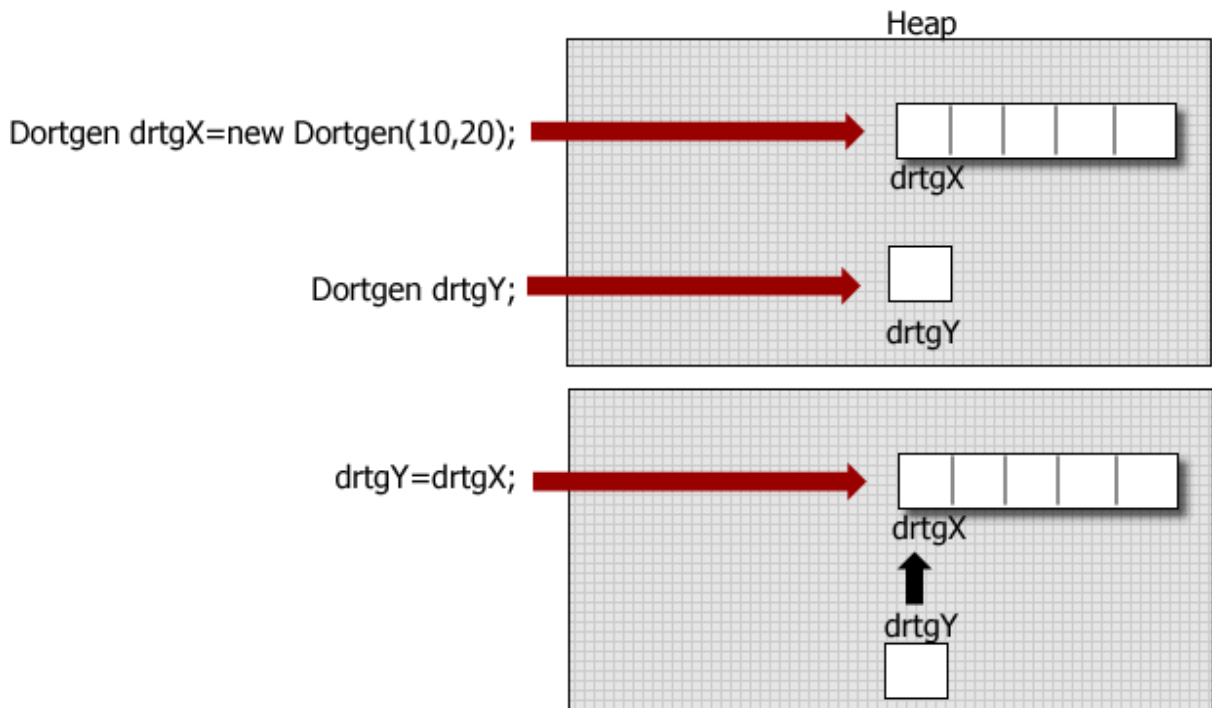
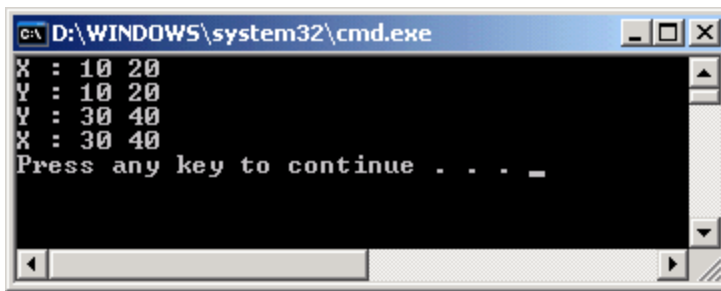
```
Dortgen drtgX=new Dortgen(10,20);  
Console.WriteLine("X : "+drtgX.ToString());
```

```
Dortgen drtgY;  
drtgY=drtgX; // referanslar eşitlenir.  
Console.WriteLine("Y : "+drtgY.ToString());
```

```
drtgY.A=30;  
drtgY.B=40;
```

```
Console.WriteLine("Y : "+drtgY.ToString());  
Console.WriteLine("X : "+drtgX.ToString());
```

Burada ilk olarak Dortgen sınıfına ait bir nesne örneğini (drtgX) oluşturuyor. Daha sonra ise drtgY isimli bir nesne örneği oluşturup bu örneğe, drtgX nesnesini atıyoruz. İşte bu noktada her iki nesne örneğinin referans adreslerini eşitlemiş oluyoruz. İzleyen satırlarda bu kez drtgY nesnesi üzerinden A ve B özelliklerinin değerlerini değiştiriyoruz. Az önce yapılan eşleştirme işlemi nedeniyle drtgY nesnesi için gerçekleştirilen değişiklikler drtgX nesnesi içinde söz konusu olacaktır. Dolayısıyla drtgX nesnesini ilk örneklerken belirlenen A ve B değişkenlerinin değerlerini kaybetmiş oluyoruz.

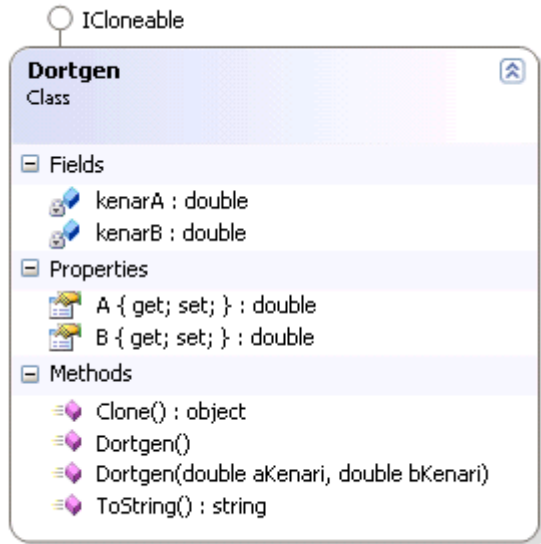




Referans tiplerinin bir birlerine atanması işlemi sonrası, bu tiplerin heap bellek bölgesindeki başlangıç adresleri eşitleneceğinden, birisi içerisindeki varlıklarda yapılacak değişiklikler diğerinde yansiyacaktır.

Yukarıdaki örnek referans tipleri arasında yapılan atamaların bir eksiklik olduğunu göstermez. Bu, bellek sisteminin doğal bir sonucudur. Dahası, referans tiplerinin bu şekilde tutulmasının ve atama işlemleri sonrası adreslerin eşitlenmesinin avantajlı olduğu durumlar da vardır. Örneğin n elemanlı bir diziyi, bir metoda parametre olarak geçirmek istediğinizde, dizinin metod bloğu içinde değer türlerinde olduğu gibi tekrardan örneklenmemesi, referans adreslerinin taşınması sayesinde gerçekleşir. Böylece bellekte gereksiz yere ikinci bir dizi örneği yaratılmamış olunur.

Elbetteki bazı hallerde referans tiplerinin üyeden-üyeye (member by member) kopyalanmasını yani ikinci bir adresleme ile yeni bir örneğin oluşturulmasını isteyebiliriz. İşte bunu gerçekleştirmek için var olan nesneyi atama işlemini yaparken klonlarız. Klonlama işleminin gerçekleştirilebilmesi için, yazmış olduğumuz sınıfa ICloneable arayüzünü uygulamamız gerekir. Bu arayüz sadece Clone isimli tek bir metod içermektedir. Yukarıdaki örneğimizde kullandığımız Dortgen sınıfına ICloneable arayüzünü aşağıdaki gibi uygulayabiliriz.



```
public class Dortgen:ICloneable
{
    // Diğer kodlar

    #region ICloneable Members
```

```

public object Clone()
{
    return new Dortgen(this.kenarA,this.kenarB);
}

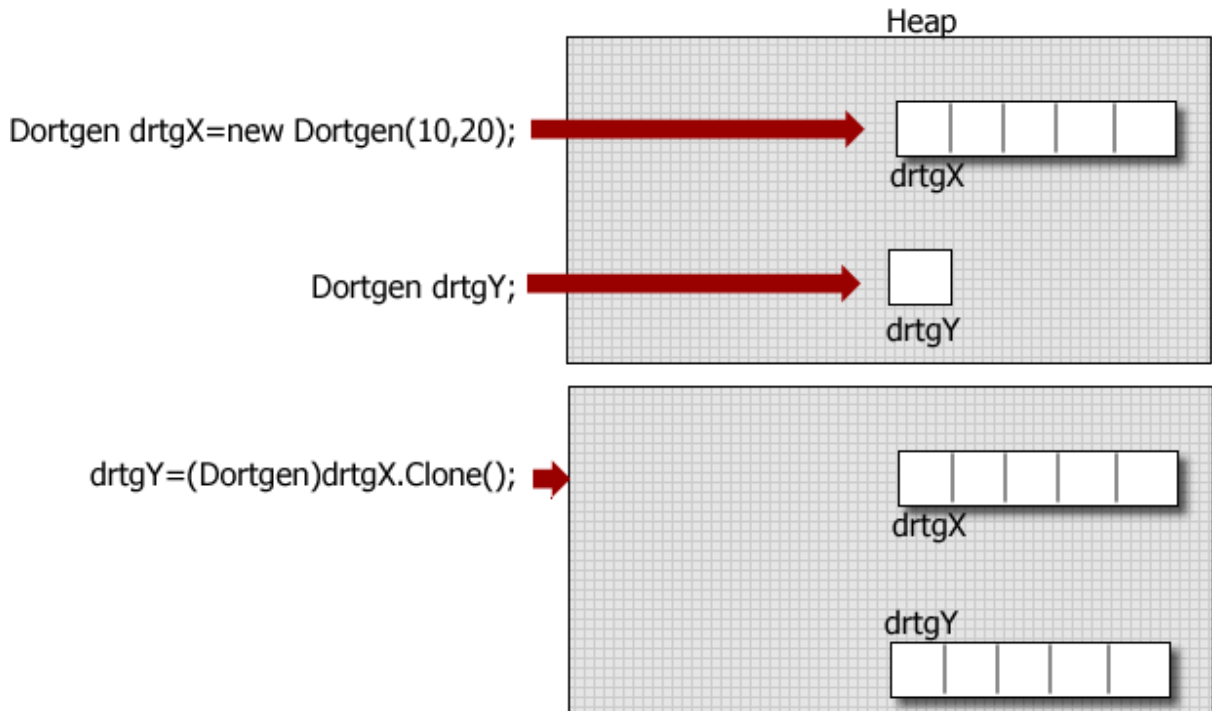
#endregion
}

```

Elbette Clone metodunu atama işlemi sırasında aşağıdaki kod parçasında olduğu gibi kullanmamız gerekir.

```
drtgY=(Dortgen)drtgX.Clone();
```

Clone metodu, Dortgen sınıfına ait yeni bir nesne örneğini o an sahip olduğu A ve B değerleri ile geriye döndürür. Clone metodu varsayılan olarak Object tipinden değerler döndürdüğü için, drtgY nesnesine yapılan atama işlemi sırasında tür dönüşüm işlemi(casting) yaparak uygun tipe atama yapmamız gerekmektedir. Bu işlem ile birlikte bellekte Dortgen sınıfına ait iki nesne örneği için ayrı ayrı adreslemeler söz konusu olacaktır. drtgX nesnesi var olan değerlerini korurken oluşturulduğu sıradaki adreste konuşlanmaya devam edecektir. drtgY nesnemiz ise drtgX nesnesinin o anki içeriğine sahip olmak üzere, belleğin farklı bir adres bölgesinde yeniden örneklendirilecektir. Durumu aşağıdaki şekil ile daha kolay anlayabiliriz.



Uygulamayı bu haliyle çalıştırdığımızda atama işlemi sonrası drtgY nesnesi üzerinden yapılan değişikliklerin drtgX nesnesinin değerlerini etkilemediğini görürüz.

```
C:\D:\WINDOWS\system32\cmd.exe
X : 10 20
Y : 10 20
Y : 30 40
X : 10 20
Press any key to continue . . .
```

Aynı etkiyi şu şekilde de yapabiliriz,

```
public object Clone()
{
    return this.MemberwiseClone();
}
```

MemberwiseClone metodu Object sınıfına ait bir metoddur. Protected erişim belirleyicisine sahiptir bu yüzden türetme işlemi söz konusu olduğunda kullanılabilir. Ayrıca override edilebilir bir metod da değildir. MemberwiseClone metodu, kullanıldığı sınıfın nesne örneğinden bir kopya daha oluşturmaktır. Yukarıdaki örneğimizi bu haliyle çalıştırdığımızda, atama sonrası klonlama işleminin başarılı bir şekilde yapıldığını ve drtgY nesnesi üzerinde yapılan değişikliklerin drtgX nesnesini etkilemediğini görürüz.

```
C:\D:\WINDOWS\system32\cmd.exe
X : 10 20
Y : 10 20
Y : 30 40
X : 10 20
Press any key to continue . . .
```

Ancak MemberwiseClone metodunu kullanırken dikkat etmemiz gereken bir durum vardır.



MemberwiseClone metodu, klonlama işlemi sırasında nesnenin static olmayan tüm değer türlerini (value types) bit-bit kopyalar. Ancak içeride kullanılan referans tipi nesne örnekleri varsa bunların adreslerini aynen yeni örneğe geçirir. Dolayısıyla referans tipleri arası yapılan atama işlemi sonrası oluşan aynı adresi gösterme çatışması, dahili referans tipi nesne örnekleri içinde geçerli olur.

Bu durumda yeni nesne örneğimiz içerisinde var olan bir referans tipi üzerinde yapılacak bir değişiklik, yine ilk nesne içindeki referans nesne örneği içinde geçerli olacaktır. Bu dikkat edilmesi gereken önemli bir durumdur. Örneğin Dortgen sınıfımız içerisinde kullanılacak yeni bir sınıfımız olduğunu varsayalım.

```
public class DortgenHesaplama
```

```
{  
}
```

```
public class Dortgen:ICloneable  
{  
    private double kenarA;  
    private double kenarB;  
    private DortgenHesaplama drgH=new DortgenHesaplama();  
  
    public double A  
    {  
        get { return kenarA; }  
        set { kenarA=value; }  
    }  
    public double B  
    {  
        get { return kenarB; }  
        set { kenarB=value; }  
    }  
    public DortgenHesaplama Hesaplama  
    {  
        get { return drgH; }  
    }  
    public Dortgen()  
    {  
        kenarA=0;  
        kenarB=0;  
    }  
  
    public Dortgen(double aKenari,double bKenari)  
    {  
        kenarA=aKenari;  
        kenarB=bKenari;  
    }  
  
    public override string ToString()  
    {  
        string dortgenBilgi=kenarA.ToString()+" "+kenarB.ToString();  
        return dortgenBilgi;  
    }  
  
    #region ICloneable Members  
  
    public object Clone()  
    {  
        return this.MemberwiseClone();  
        // return new Dortgen(this.kenarA,this.kenarB);  
    }  
}
```

```
#endregion  
}
```

Burada dikkat ederseniz, DortgenHesaplama isimli sınıfa ait bir nesne örneğini Dortgen sınıfı içerisinde kullanmaktayız. Bu, Dortgen referans tipi içerisinde yer alan başka bir referans tipi nesne örneğidir. Atama işlemi sonrası MemberwiseClone metodu DortgenHesaplama sınıfına ait nesne örneğinin sadece adresini kopyalayacaktır. Uygulamamızın kodlarını yukarıdaki Dortgen sınıfına göre aşağıdaki gibi değiştirelim.

```
Dortgen drtgX=new Dortgen(10,20);  
Dortgen drtgY;  
drtgY=(Dortgen)drtgX.Clone();  
drtgY.A=30;  
drtgY.B=40;
```

```
bool refAdrEsitmi=object.ReferenceEquals(drtgX,drtgY);  
Console.WriteLine("drtgX & drtgY referans adresleri eşit mi ?"+refAdrEsitmi.ToString());
```

```
refAdrEsitmi=object.ReferenceEquals(drtgX.Hesaplama,drtgY.Hesaplama);  
Console.WriteLine("drtgX.Hesaplama & drtgY.Hesaplama referans adresleri eşit mi  
?" +refAdrEsitmi.ToString());
```



ReferenceEquals parametre olarak aldığı nesne örneklerinin bellek referanslarının aynı olup olmadığını kontrol ederek sonucu bool tipinden geriye döndüren Object sınıfına ait static bir metoddur.

Görüldüğü gibi drtgX ve drtgY nesnelerini Object sınıfının ReferenceEquals metodu ile karşılaştırdığımızda, false değerini alıyoruz. Çünkü biz Dortgen sınıfımıza IClonable arayüzünü ve Clone metodu içerisinde MemberwiseClone fonksiyonunu uygulayarak, drtgX nesne örneğini klonluyoruz. Bu sebeple heap bellek bölgesinde ayrı bir Dortgen nesne örneği oluşturuluyor. Dolayısıyla adresler artık farklı olacağından ReferenceEquals metodu geriye false değer döndürecektir.

Ancak drtgX ve drtgY nesne örnekleri içerisinde yer alan DortgenHesaplama sınıfına ait nesne örneklerinin referanslarını karşılaştırdığımızda true değerinin döndüğünü görmekteyiz. Yani DortgenHesaplama sınıfına ait nesne örnekleri için klonlama işlemi gerçekleşmemiş bunun yerine nesnelerin heap bellek bölgesindeki adresleri eşitlenmiştir. Bu dikkat edilmesi gereken bir durumdur ve

sınıflarımızı programlarken gerekli tedbirlerin alınmasını gerektirebilecek kadar önemlidir. Böylece geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

C# 2.0 ile Generic Delegates - 21 Kasım 2005 Pazartesi

C# 2.0, generics, delegate, generic delegate,

Değerli Okurlarım Merhabalar,

Bu makalemizde generic temsilcilerin (generic delegates) ne olduğunu ve nasıl kullanılabildiğini incelemeye çalışacağız. .Net 2.0 ile gelen en önemli yenilik generic mimarisidir. Generic mimarisi, tür bağımsız algoritmalar kurmamıza imkan sağlayan gelişmiş bir yapıdır. .Net 2.0' da sınıfları (class), yapıları (struct), arayüzleri (interface), metodları (method), koleksiyonları (collection) ve temsilcileri (delegate) generic olarak oluşturabilir ve kullanabiliriz. Bildiğiniz gibi generic mimarisinin sağlamış olduğu iki önemli özellik vardır. Bunlar tip güvenliği (type-safety) ve performans artışıdır. Özellikle performans ile kastedilen konu gereksiz boxing ve unboxing işlemlerinin ortadan kaldırılabilmesidir. Generic mimarinin getirdiği bu avantajları delegate (temsilci) tipi içinde kullanabilmekteyiz.

İlk olarak generic temsilcilere neden ihtiyacımız olabileceğini 1.1 versiyonundaki kullanımını göz önüne alarak irdelemeye çalışalım. Aşağıdaki örnek uygulamada overload edilmiş Toplam isimli iki metod yer almaktadır. Bu iki metod farklı tipte parametreler almaktadır. Bunun yanında dönüş tipleride farklıdır. Bu metodları temsilci nesneleri ile işaret etmek istediğimizde iki ayrı temsilci tipi tanımlamamız gerekecektir. Nitekim temsilci tipi, tanımlamış olduğu desen ile(dönüş tipi ve metod imzası) birebir uyumlu metodları işaret edebilmektedir.

```
using System;
```

```
namespace UsingGenericDelegates
```

```
{
```

```
    #region Temsilci tipleri tanımlanır
```

```
    public delegate float TemsilciFloat(float a,float b);
```

```
    public delegate int TemsilciInt(int a,int b);
```

```
    #endregion
```

```
    public class TemelAritmetik
```

```
    {
```

```
        public TemelAritmetik() {}
```

```
        public float Toplam(float x,float y)
```

```
        {
```

```
            return x+y;
```

```

    }

    public int Toplam(int x,int y)
    {
        return x+y;
    }
}

```

TemelAritmetik sınıfımıza ait Toplam metodlarını çalışma zamanında belirlediğimiz temsilci nesneleri ile aşağıdaki kod parçasında olduğu gibi ilişkilendirebiliriz.

using System;

```

namespace UsingGenericDelegates
{
    class TestClass
    {
        [STAThread]
        static void Main(string[] args)
        {
            TemelAritmetik ta=new TemelAritmetik();

            #region Temsilciler Oluşturulur

            TemsilciInt tint=new TemsilciInt(ta.Toplam);
            TemsilciFloat tFloat=new TemsilciFloat(ta.Toplam);

            #endregion

            int toplamInt=tint(1,2);
            float toplamFloat=tFloat(1.2f,1.2f);

            Console.WriteLine(toplamInt.ToString());
            Console.WriteLine(toplamFloat.ToString());

        }
    }
}

```

Bu uygulama sorunsuz şekilde çalışmaktadır. Ancak benzer işlemlere sahip olan yani int ve float tipinden verileri toplayıp bu türde geri döndüren metodları çalışma zamanında ilgili temsilci nesne örnekleri ile işaret edebilmek için iki ayrı delegate tipi tanımlamamız gerekmiştir. İşte .Net 2.0 ile birlikte gelen generic mimarisi sayesinde tek bir temsilci tipi tanımını kullanarak birden fazla metodu istenilen veri tipi için kullanabiliriz. Başka bir deyişle, yukarıdaki örnekte olduğu gibi iki farklı delegate tipi tanımlamak yerine generic yapıda tek bir temsilci tipi (delegate type) tanımlayıp, bu tip üzerinden çalışma zamanında uygun metodları çağırabiliriz.

.Net 2.0 için generic temsilci (generic delegate) tanımlamasında da anahtar nokta, açılabilir ayraçlar (<>) arasına yazılan tür belirtme operatörüdür. Örneğin, void dönüş tipine sahip ama parametre tipleri belli olmayan (başka bir deyişle çalışma zamanında belli olacak olan) bir temsilciyi aşağıdaki gibi tanımlayabiliriz.

```
public delegate void TemsilciX<T>(T deger);
```

Bu ifadede, TemsilciX isimli delegate tipinin çalışma zamanında geri dönüş tipi olmayan (void) ve her hangibir tipte tek bir parametre alan metodları işaret edebileceğini söylemiş oluyoruz. Örneğimizde bu metod desenine uyan aşağıdaki Yarıcap fonksiyonumuzun olduğunu düşünelim.

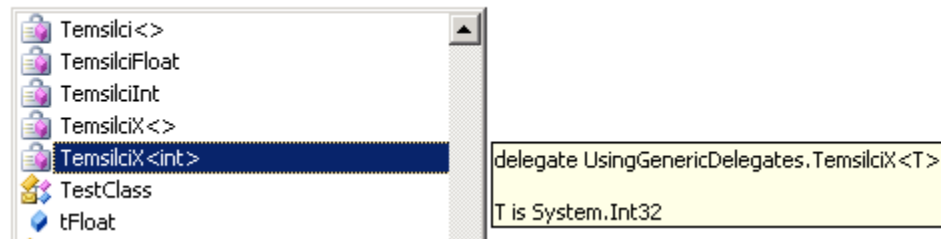
```
public void Yarıcap(double r)
{
    Console.WriteLine("yarıcap ={0}", r);
}
```

Buna göre TemsilciX delegate tipini uygulamamızda aşağıdaki gibi kullanabiliriz.

```
TemsilciX<double> tX = new TemsilciX<double>(ta.Yarıcap); // Temsilci nesne örneği oluşturulur.
tX(1.3); // Temsilcimizin işaret ettiği metod çağırılır.
```

Bu kod parçasında tX örneğini oluştururken, çalışma zamanında double tipinden parametre alacak metodları işaret edebileceğini belirtmiş oluyoruz. Diğer yandan örneğin, int tipinden bir parametre alıp void dönüş tipine sahip başka bir metodu işaret etmek istediğimizde, TemsilciX delegate tipini buna göre tekrardan örneklendirebiliriz.

```
TemsilciX<int> tY=new
```



Böylece tek bir temsilci nesnesini kullanarak çalışma zamanında kendi belirlediğimiz tipleri kullanan metodları işaret edebiliriz. Generic temsilcileri kullanarak, işaret edecekleri metodların dönüş tiplerinin çalışma zamanında ne olacağını da belirleyebiliriz. Makalemizin başındaki örneği dikkate aldığımızda Toplam metodunun her iki versiyonunda farklı tipten geri dönüş değerlerine sahip olduğunu görmekteyiz. Buna göre, generic temsilci tipimizi aşağıdaki gibi tanımlayabiliriz. Bu sefer, her metod için ayrı birer temsilci tipi tanımlamaktansa, tek bir generic temsilci tipi tanımlamayı görecektir.

```
public delegate R Temsilci<T,R>(T deger1,T deger2);
```

Burada R harfi ile temsilcimizin çalışma zamanında işaret edeceği metodun dönüş tipini belirtmiş oluruz. T harfi ilede, metodun alacağı parametrelerin tipini belirliyoruz. Bu değişiklikleri göz önüne aldığımızda, makalemizin başındaki örneğimizi aşağıdaki haliyle güncelleyebiliriz.

```
using System;
```

```
namespace UsingGenericDelegates
```

```
{
```

```
    // Temsilci tipimiz tanımlanır.
```

```
    public delegate R Temsilci<T, R>(T deger1, T deger2);
```

```
    class TestClass
```

```
    {
```

```
        [STAThread]
```

```
        static void Main(string[] args)
```

```
        {
```

```
            TemelAritmetik ta=new TemelAritmetik();
```

```
            #region temsilcimize ait nesne örnekleri oluşturuluyor
```

```
            Temsilci<float, float> t1 = new Temsilci<float, float>(ta.Toplam);
```

```
            Temsilci<int,int> t2=new Temsilci<int,int>(ta.Toplam);
```

```
            #endregion
```

```
            Console.WriteLine(t1(1.2f, 1.3f));
```

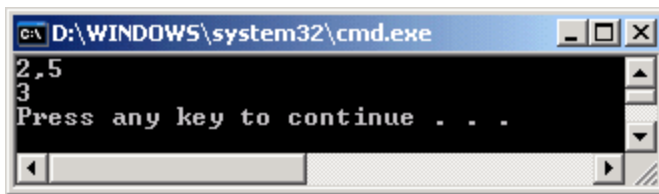
```
            Console.WriteLine(t2(1,2));
```

```
        }
```

```
    }
```

```
}
```

Her ne kadar t1 ve t2 olmak üzere TemelAritmetik sınıfı içerisindeki Toplam metodlarının her biri için ayrı ayrı iki delegate nesnesi örneklemiş olsakta; bu nesne örneklerinin ait olduğu delegate tipi tektir. Uygulamayı çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.



Tek bir generic temsilci tipi ile, çalışma zamanında değişik dönüş tipi ve parametrelere sahip metodları işaret edebiliriz.

Temsilcilerin sağlamış olduğu tip güvenliği (type-safety) ve performans artışı gibi faydalar dışında özellikle kendi tiplerimizi kullandığımız takdirde devreye giren kısıtlamalar (constraints), delegate tipleri içinde geçerlidir. Bildiğiniz gibi generic yapılarda **Where** anahtar sözcüğü yardımıyla, parametre tiplerine ilişkin kesin bazı kurallar koyabiliyoruz. Örneğin TemelAritmetik isimli sınıfımızda aşağıdaki gibi iki string tipte parametre alan bir metodumuz olduğunu düşünelim.

```
public string Toplam(string x, string y)
{
    return x + y;
}
```

Böyle bir durumda, Temsilci delegate tipimiz ile çalışma zamanında bu metodu aşağıdaki kod parçasında olduğu gibi işaret edebiliriz.

```
Temsilci<string, string> t3 = new Temsilci<string, string>(ta.Toplam);
Console.WriteLine(t3("Burak", "Selim"));
```

Burada önemli olan nokta Toplam metodunun bu versiyonunun string tipinde, (bir başka deyişle referans tipinde) parametreler alıyor oluşudur. Oysaki biz temsilcimizin sadece değer türünde (value types) parametreler alan metodlar işaret etmesini de isteyebiliriz. İşte böyle bir sorunu, where anahtar sözcüğü ile parametre tiplerine yönelik (veya dönüş tipine yönelik) kısıtlamalar girerek aşabiliriz. Dolayısıyla Temsilci isimli delegate tipimizi aşağıdaki gibi tanımlamamız yeterli olacaktır.

```
public delegate R Temsilci<T, R>(T deger1, T deger2) where T:struct where R:struct;
```

Burada where anahtar sözcüklerini kullanarak T ve R türlerinin mutlaka struct tipinde olmaları gerektiğini, bir başka deyişle değer türü olmaları gerektiğini belirtmiş oluyoruz. Dolayısıyla uygulamamızı bu haliyle derlemek istediğimizde aşağıdaki şekilde görüldüğü gibi, derleme zamanı hata mesajlarını alırız.

```
// Temsilci tipimiz tanımlanır.
public delegate R Temsilci<T, R>(T deger1, T deger2) where T:struct where R:struct;

class TestClass
{
    [STAThread]
    static void Main(string[] args)
    {
        TemelAritmetik ta=new TemelAritmetik();

        #region temsilcimize ait nesne örnekleri oluşturuluyor
        Temsilci<float, float> t1 = new Temsilci<float, float>(ta.Toplam);
        Temsilci<int,int> t2=new Temsilci<int,int>(ta.Toplam);
        Temsilci<string, string> t3 = new Temsilci<string, string>(ta.Toplam);
        #endregion
    }
}
```

Kısıtlamalar

Error List

4 Errors 0 Warnings 0 Messages

	Description
1	The type 'string' must be a non-nullable value type in order to use it as parameter 'T' in the generic type or method 'UsingGenericDelegates.Temsilci<T,R>'
2	The type 'string' must be a non-nullable value type in order to use it as parameter 'R' in the generic type or method 'UsingGenericDelegates.Temsilci<T,R>'
3	The type 'string' must be a non-nullable value type in order to use it as parameter 'T' in the generic type or method 'UsingGenericDelegates.Temsilci<T,R>'
4	The type 'string' must be a non-nullable value type in order to use it as parameter 'R' in the generic type or method 'UsingGenericDelegates.Temsilci<T,R>'

Yaptıklarımızı kısaca gözden geçirecek olursak generic temsilcilerin sağlamış olduğu avantajları şu şekilde sıralayabiliriz.

*Temsilci tiplerini **tür bağımsız** olacak şekilde tanımlayabiliriz.*

*Where anahtar sözcüğünün imkanlarından faydalanarak temsilci tiplerine ait parametre ve dönüş değerleri üzerinde **tip güvenliği (type-safety)** daha üst düzeyde sağlayabiliriz.*

*Çalışma zamanında farklı parametre tiplerini kullanan uygun metodlar için **ayrı ayrı temsilci tipleri** tanımlamak zorunda kalmayız.*

Böylece geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.

Generic Mimaride

Kısıtlamaların(Costraints) Kullanımı - 31

Aralık 2005 Cumartesi

C# 2.0, generics, constraints,

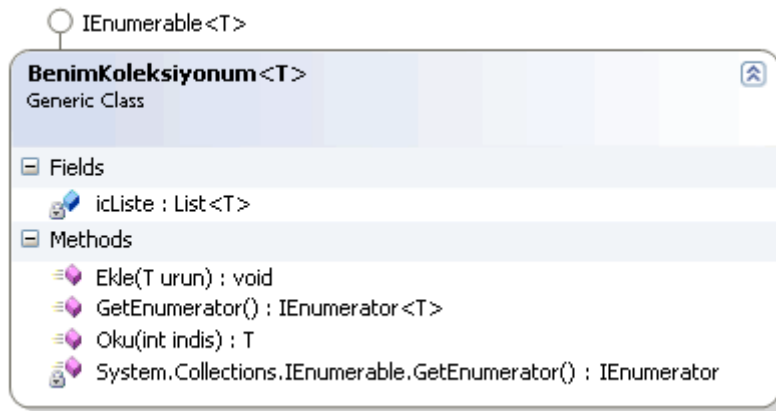
Değerli Okurlarım Merhabalar,

Bu makalemizde .Net 2.0 ile birlikte gelen Generic mimarisinin uygulanışında, kısıtlamaların rolünü basit örnekler ile incelemeye çalışacağız. Generic mimari her ne kadar tür bağımsız algoritmaların geliştirilmesine izin versede, bazı durumlarda çeşitli zorlamaların uygulanmasında gerektirir. Örneğin generic olması planlanan tiplerin sadece referans tipleri ile çalışmasını isteyebiliriz. Generic bir tipe her hangibir zorunluluk kuralını uygulayabilmek için where anahtar sözcüğünü içeren bir ek ifade kullanılır. Bu ifadeler 5 adettir ve aşağıdaki tabloda gösterilmektedir.

Koşul	Syntax
Değer tipi olma zorunluluğu	where Tip : struct
Referans tipi olma zorunluluğu	where Tip : class
Constructor zorunluluğu	where Tip : new()
Türeme zorunluluğu	where Tip : <Temel Sınıf>
Interface zorunluluğu	where Tip : <Interface>

İlk olarak struct ve class zorunluluklarını incelemeye çalışacağız. Bu kısıtlamaları, Generic bir tip içerisinde yer alan tiplerin değer türü veya referans türlerinden mutlaka ve sadece birisi olmasını istediğimiz durumlarda kullanırız. Generic' lik, uygulandığı tip için tür bağımsızlığını ortaya koyan bir yapıdır. Generic mimari sayesinde bir tipin çalışma zamanında kullanacağı üyelerin türünü belirleyebiliriz. Ancak, hangi tür olursa olsun, ya değer türü ya da referans türü söz konusu olacaktır. İşte kısıtlamaları kullanarak, generic mimari üzerinde referans türümü yoksa değer türümü olacağını belirleyebiliriz.

Konuyu daha iyi anlayabilmek için şu örneği göz önüne alalım. Bildiğiniz gibi C# 2.0 beraberinde, Framework 1.1 ile gelen koleksiyonların generic karşılıklarını da getirmiştir. Generic koleksiyonlar çalışma zamanında sadece belirtilen türden nesneleri kullanır. Doğal olarak ya değer türlerini ya da referans türlerini kullanılır. Peki ya generic bir koleksiyonun sadece değer türlerini taşımasını istersek ne yapabiliriz. Bir şekilde T tipinin sadece değer türü olması zorunluluğunu bildirmemiz gerekecektir. Aşağıdaki şemada BenimKoleksiyonum isimli özel bir koleksiyon tanımı yer almaktadır. BenimKoleksiyonum isimli generic sınıfımız içeride List türünden bir generic koleksiyonu kullanmaktadır.



```

public class BenimKoleksiyonum<T> :IEnumerable<T>
{
    private List<T> icListe = new List<T>();
    public void Ekle(T urun)
    {
        icListe.Add(urun);
    }
    public T Oku(int indis)
    {
        return icListe[indis];
    }
    #region IEnumerable<T> Members

    public IEnumerator<T> GetEnumerator()
    {
        return icListe.GetEnumerator() ;
    }
    #endregion

    #region IEnumerable Members

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return icListe.GetEnumerator();
    }
  
```



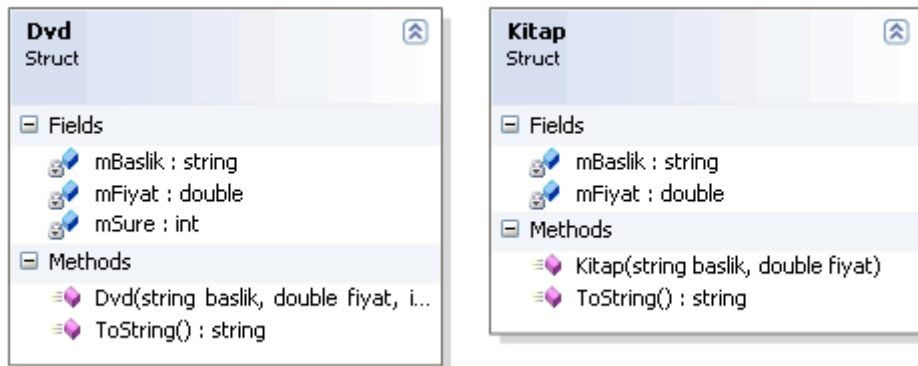
```

    }

    #endregion
}

```

Burada sınıfımız içerisinde yer alan icListe isimli List tipinden generic koleksiyonumuz, T türünden elemanlar ile iş yapacak şekilde tasarlanmıştır. Aynı şekilde, Ekle, Oku ve GetEnumerator metodunun bir versiyonuda sadece T tipinden elemanlar üzerinde iş yapmaktadır. Şimdi BenimKoleksiyonum sınıfına ek olarak aşağıda bilgileri verilen iki yapımız(struct) olduğunu düşünelim.



Dvd.cs ve Kitap.cs;

```

struct Kitap
{
    private string mBaslik;
    private double mFiyat;

    public Kitap(string baslik, double fiyat)
    {
        mBaslik = baslik;
        mFiyat = fiyat;
    }
    public override string ToString()
    {
        return mBaslik + " " + mFiyat;
    }
}

struct Dvd
{
    private string mBaslik;
    private double mFiyat;
    private int mSure;

    public Dvd(string baslik, double fiyat,int sure)
    {

```

```

        mBaslik = baslik;
        mFiyat = fiyat;
        mSure = sure;
    }
    public override string ToString()
    {
        return mBaslik + " " + mFiyat+" "+mSure;
    }
}

```

Diyelimki BenimKoleksiyonum isimli sınıfın uygulama içerisinde sadece yukarıda bilgileri verilen Kitap, Dvd ve ileride bunlar gibi geliştirilebilecek başka struct' lar ile ilgili işlemler yapmasını istiyoruz. Yani asla ve asla referans tiplerinin kullanmasını istemediğimizi düşünelim. Bu durumda tek yapmamız gereken şey BenimKoleksiyonum isimli sınıfa bir generic Constraint uygulamaktır.

```
public class BenimKoleksiyonum<T> :IEnumerable<T> where T:struct
```

Nasıl ki generic tipi değer türünden olmaya yukarıdaki söz diziminde olduğu gibi zorlayabiliyorsak, aynı işi referans türlerine zorlamak içinde yapabiliriz. Tek yapmamız gereken struct yerine class anahtar sözcüğünü kullanmak olacaktır.

```
public class BenimKoleksiyonum<T> :IEnumerable<T> where T:class
```

Struct zorlamasını kullandığımız takdirde eğer ki, BenimKoleksiyonum sınıfını kod içerisinde herhangi bir referans türü ile kullanmaya çalışırsak (örneğin string referans türü ile) derleme zamanında aşağıdaki hata mesajlarını alırız.

```
BenimKoleksiyonum<String> myCol = new BenimKoleksiyonum<String>();
```

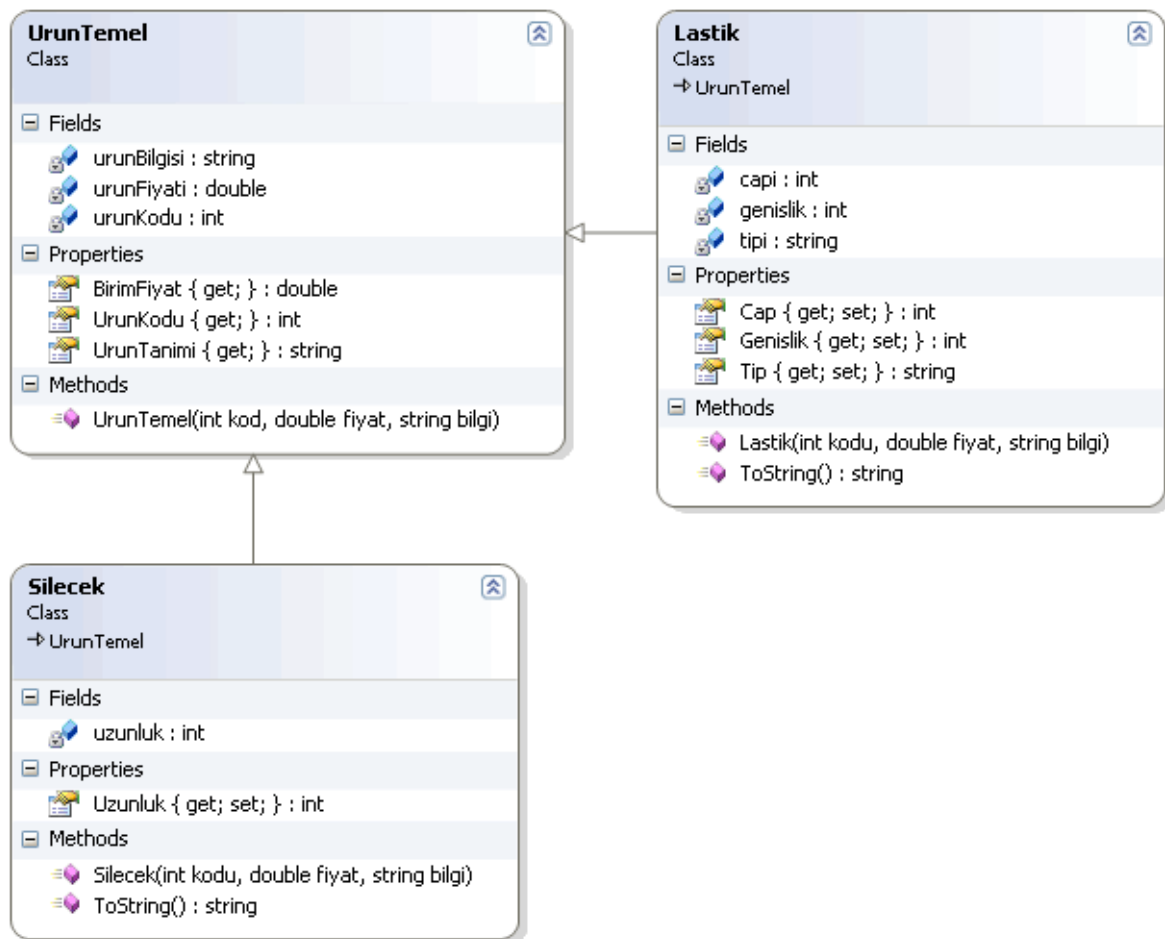
List		
2 Errors	0 Warnings	0 Messages
Description	File	
The type 'string' must be a non-nullable value type in order to use it as parameter 'T' in the generic type or method 'UsingGenericConstraint.BenimKoleksiyonum<T>'	Program.cs	
The type 'string' must be a non-nullable value type in order to use it as parameter 'T' in the generic type or method 'UsingGenericConstraint.BenimKoleksiyonum<T>'	Program.cs	

Örneğimizde kullandığımız değer türü kısıtlaması her ne kadar koleksiyonun sadece struct' ları kullanmasını sağlıyorsa da, eğer sadece Kitap ve Dvd gibi kendi tanımladığımız struct' ların kullanılmasına bir zorunluluk getirmemektedir. Nitekim int, double gibi struct' larıda BenimKoleksiyonum ile birlikte kullanabilirsiniz. Bu noktada daha güçlü bir kısıtlama kullanmakta fayda vardır. Tam olarak soy bağımlılığı kısıtlaması bu talebimizi karşılar. Bu kısıtlamaya göre generic olarak kullanılan türün belli bir tipten veya bu tipten türeyen başka tip(tipler)den olması zorunluluğu vardır. Dolayısıyla, ister değer türü ister referans türü olsun, belli bir tür veya bu türden kalıtsal olarak türeyen tiplerin

kullanılmasını zorunlu hale getirebiliriz. Bu kısıtlamayı uygulamak için aşağıdaki söz dizimi kullanılır.

where Tip : <Temel Sınıf>

Bu kısıtlamayı anlamak için şu örneği göz önüne alalım. Otomobillere ait çeşitli ürünleri nesneye dayalı mimari altında tasarlamaya çalıştığımızı düşünelim. Her ürünü ayrı bir sınıf olarak tasarlayabiliriz. Lakin pek çok ürünün ortak olan bir takım özellikleri ve işlevleride vardır. Bu tip üyeleri temel bir sınıfta toplayabiliriz. Çok basit olarak aşağıdaki sınıf diagramında görülen bir örnek geliştirelim. Burada Lastik ve Silecek isimli sınıflarımız, UrunTemel sınıfından türemiştir. UrunTemel isimli sınıfımız tüm ürünler için ortak olan ürün kodu , fiyat ve kısa açıklama bilgileri için gerekli özellikleri barındırmaktadır.



Yukarıdaki şemada görülen UrunTemel, Lastik ve Silecek isimli sınıflarımıza ait kod satırlarımız ise aşağıdaki gibidir.

UrunTemel.cs

```
using System;
using System.Collections.Generic;
using System.Text;
```

```

namespace UsingGenericConstrains
{
    public class UrunTemel
    {
        private int urunKodu;
        private double urunFiyati;
        private string urunBilgisi;

        public UrunTemel(int kod,double fiyat,string bilgi)
        {
            urunKodu = kod;
            urunFiyati = fiyat;
            urunBilgisi = bilgi;
        }

        public int UrunKodu
        {
            get{return urunKodu;}
        }
        public double BirimFiyat
        {
            get{return urunFiyati;}
        }

        public string UrunTanimi
        {
            get{return urunBilgisi;}
        }
    }
}

```

Lastik.cs

```

using System;
using System.Collections.Generic;
using System.Text;

namespace UsingGenericConstrains
{
    public class Lastik : UrunTemel
    {
        private int capi;
        private int genislik;
        private string tipi;

        public Lastik(int kodu, double fiyat, string bilgi)
            : base(kodu, fiyat, bilgi)
        {

```

```

    }

    public int Cap
    {
        get{return capi;}
        set{capi = value;}
    }

    public int Genislik
    {
        get{return genislik;}
        set{genislik = value;}
    }

    public string Tip
    {
        get{return tipi;}
        set{tipi = value;}
    }

    public override string ToString()
    {
        return UrunKodu.ToString() + " " + BirimFiyat.ToString() + " " + UrunTanimi + " " +
        Cap.ToString() + " " + Genislik.ToString() + Tip;
    }
}

```

Silecek.cs

```

using System;
using System.Collections.Generic;
using System.Text;

namespace UsingGenericConstrains
{
    public class Silecek : UrunTemel
    {
        private int uzunluk;

        public Silecek(int kodu, double fiyat, string bilgi)
            : base(kodu, fiyat, bilgi)
        {
        }

        public int Uzunluk
        {
            get{return uzunluk;}
            set{uzunluk = value;}
        }
    }
}

```

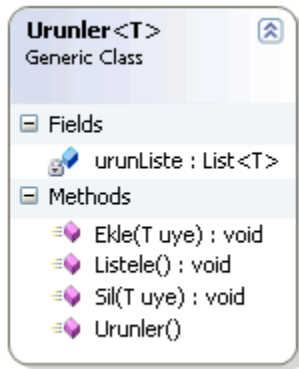
```

    }

    public override string ToString()
    {
        return UrunKodu.ToString() + " " + BirimFiyat.ToString() + " " + UrunTanimi + " " +
        Uzunluk.ToString();
    }
}
}

```

Şimdi buradaki soy ilişkisini kullanacak tipte bir yönetici sınıf geliştirdiğimizi düşünelim. Urunler isimli bu sınıfımızı değişik tipleri barındırabilecek bir generic koleksiyon ile birlikte kullanacağız. Doğal olarak, Urunler isimli sınıfımızda generic bir mimaride geliştireceğiz. Urunler isimli sınıfımıza ait şema bilgisini ve kod satırlarını aşağıdaki grafikte görebilirsiniz.



Urunler.cs

```

using System;
using System.Collections.Generic;
using System.Text;

namespace UsingGenericConstraint
{
    public class Urunler<T> // where T : UrunTemel
    {
        private List<T> urunListe;

        public Urunler()
        {
            urunListe = new List<T>();
        }

        public void Ekle(T uye)
        {
            urunListe.Add(uye);
        }

        public void Sil(T uye)

```

```

    {
        urunListe.Remove(uye);
    }
    public void Listele()
    {
        foreach (T uye in urunListe)
        {
            Console.WriteLine(uye.ToString());
        }
    }
}

```

Şimdi bu sınıflarımızı Main metodumuzda aşağıdaki kod satırları ile kullanmaya çalışalım.

```

Urunler<Int16> urunler = new Urunler<short>();
urunler.Ekle(13);
urunler.Ekle(15);
urunler.Ekle(24);
urunler.Listele();

```

Uygulamamızı bu haliyle çalıştırdığımızda her hangibir sorun ile karşılaşmayız. Urunler isimli sınıfımız Generic bir yapıda olduğundan Short veri türünden değişkenleri işleyecek şekilde tasarlayabiliriz. Fakat bu istediğimiz bir kullanım şekli değildir. Nitekim biz Urunler isimli sınıfımızın generic olmasını ama sadece Urun grubu ile ilgili türleri işlemesini istemekteyiz. Bu sebepten yorumsal olarak yazdığımız kısıtlama satırını kaldırmamız ve bu sayede Urunler sınıfını sadece UrunTemel soyundan gelecek tiplerin kullanımına açmamız gerekiyor. Uygulamamızı yukarıdaki kodları ile bırakıp, kısıtlamamızı devreye soktuğumuzda, build işleminden sonra aşağıdaki hata mesajları alırız.

```

namespace UsingGenericConstraint
{
    public class Urunler<T> where T : UrunTemel
    {
        private List<T> urunListe;

        public Urunler()
        {
            urunListe = new List<T>();
        }
    }
}

```

Error List	
<div> <div>2 Errors</div> <div>0 Warnings</div> <div>0 Messages</div> </div>	
	Description
1	The type 'short' must be convertible to 'UsingGenericConstraint.UrunTemel' in order to use it as parameter 'T' in the generic type or method 'UsingGenericConstraint.Urunler<T>'
2	The type 'short' must be convertible to 'UsingGenericConstraint.UrunTemel' in order to use it as parameter 'T' in the generic type or method 'UsingGenericConstraint.Urunler<T>'

Urunler isimli işlevsel sınıfımız, generic tip olarak sadece TemelUrun ve soyundan gelen sınıf nesne örnekleri ile çalışabilecek şekilde kısıtlandırıldığı için bu hata mesajları alınmıştır. Ancak uygulama kodlarımızı aşağıdaki gibi değiştirdiğimizde herhangi bir problem ile karşılaşmayız.

```
Urunler<UrunTemel> urunler = new Urunler<UrunTemel>();
```

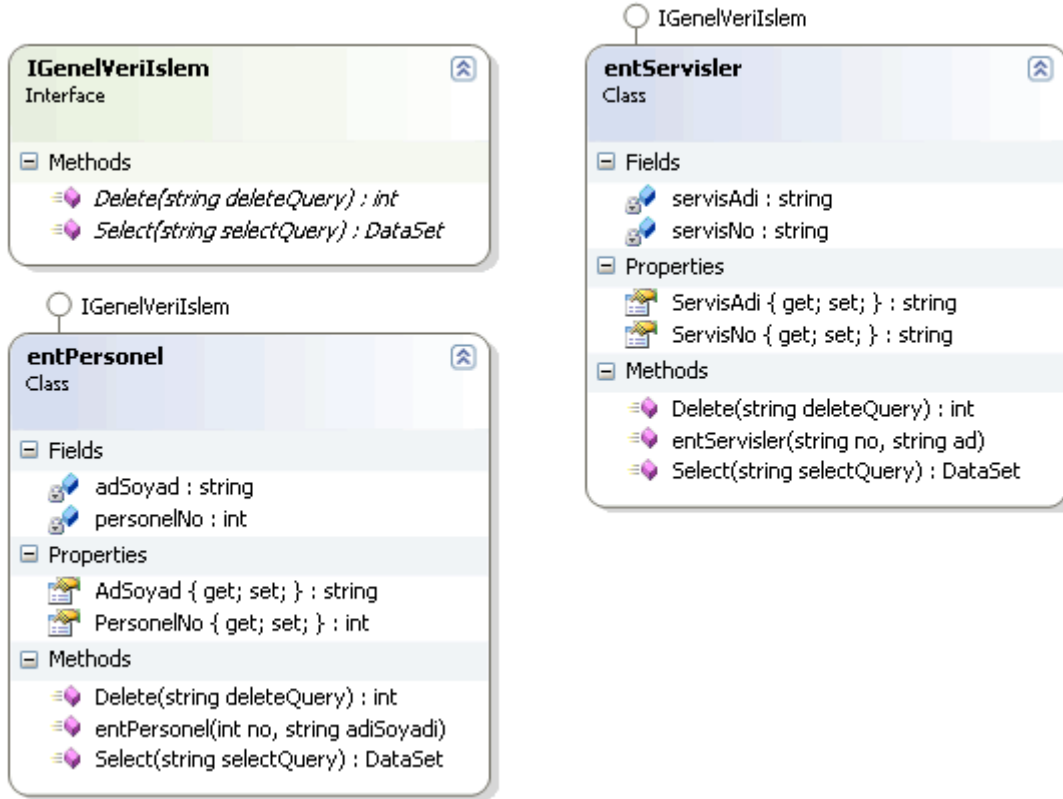
```
Lastik lst = new Lastik(1000, 10, "Otomobil Lastiği");  
lst.Tip = "Kış Lastiği";  
lst.Cap = 185;  
lst.Genislik = 75;
```

```
Silecek slc = new Silecek(1001, 5, "On silecek takimi");  
slc.Uzunluk = 60;
```

```
urunler.Ekle(lst);  
urunler.Ekle(slc);
```

```
urunler.Listele();
```

Generic kısıtlamalar ile ilgili olarak göreceğimiz bir diğer modelde interface uygulama zorunluluğudur. Bu kurala göre, generic mimari içinde kullanılacak olan tür, koşul olarak belirtilen interface' i veya ondan türeyenlerini mutlaka implemente etmiş bir tip olmak zorundadır. Örneğin aşağıdaki mimariyi göz önüne alalım. Bu örnekte bir veritabanı sisteminde yer alan varlıklar çeşitli sınıflar ile temsil edilmeye çalışılmıştır. Bu varlıkların ortak özelliği mutlaka ve mutlaka IGenelVeriIslem isimli arayüzü uyguluyor olmalarıdır.



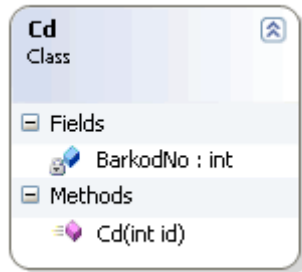
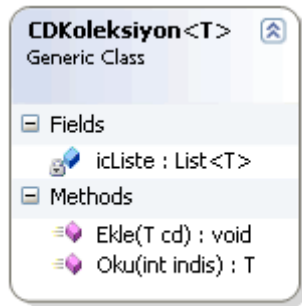
Bizim bu varlıkları yönetecek bir sınıfımız var ise ve bu sınıfı generic bir mimari içerisinde kullanmak istiyorsak sadece IGenelVeriIslem arayüzünü uygulayan tiplerin kullanılmasını da garanti edebiliriz. Tek yapmamız gereken ilgili yönetici sınıfımıza arayüz kısıtlamasını aşağıdaki kod satırlarında görüldüğü gibi eklemek olacaktır.

```

class entYoneticici<T> where T:IGeneVeriIslem
{
    // örnek kod satırları
}
  
```

Burada T tipinin mutlaka IGenelVeriIslem arayüzünü implemente etmiş bir tür olması zorunluluğu getirilmektedir. Böylece yönetici sınıfımız çeşitli tipleri kullanabiliyor olmakla birlikte şu an için sadece ilgili arayüzü uygulayan varlık sınıflarına destek vermektedir.

Generic kısıtlamalar ile ilgili bir diğer özellikte, varsayılan yapıcı metod olması zorunluluğudur. Buna göre generic mimari içerisinde kullanılan bağımsız türün mutlaka varsayılan yapıcı (default constructor-parameterless constructor) metoda sahip olması amaçlanmaktadır. Bunu daha iyi anlayabilmek için, varsayılan yapıcısı olmayan bir tipi, generic olarak kullanmaya çalışmalıyız. Aşağıdaki örneği göz önüne alalım.



Bu örnekte, CDKoleksiyon basit olarak tasarlanmış generic tipte bir koleksiyondur. Cd isimli referans tipimizi pekala bu generic koleksiyon içerisinde kullanabiliriz. Lakin Cd isimli sınıfımızın default constructor metodu mevcut değildir. Bunun yerine parametre alan overload edilmiş bir versiyonu kullanılmıştır. CDKoleksiyon sınıfının, taşıyacağı generic tiplerin mutlaka ve mutlaka varsayılan yapıcı metodları içermesini isteyeceğimiz durumlar söz konusu olabilir. Bu zorlamayı gerçekleştirmek için tek yapmamız gereken new kısıtlamasını kullanmak olacaktır.

```
public class CDKoleksiyon<T> where T:new()
{
    private List<T> icListe = new List<T>();

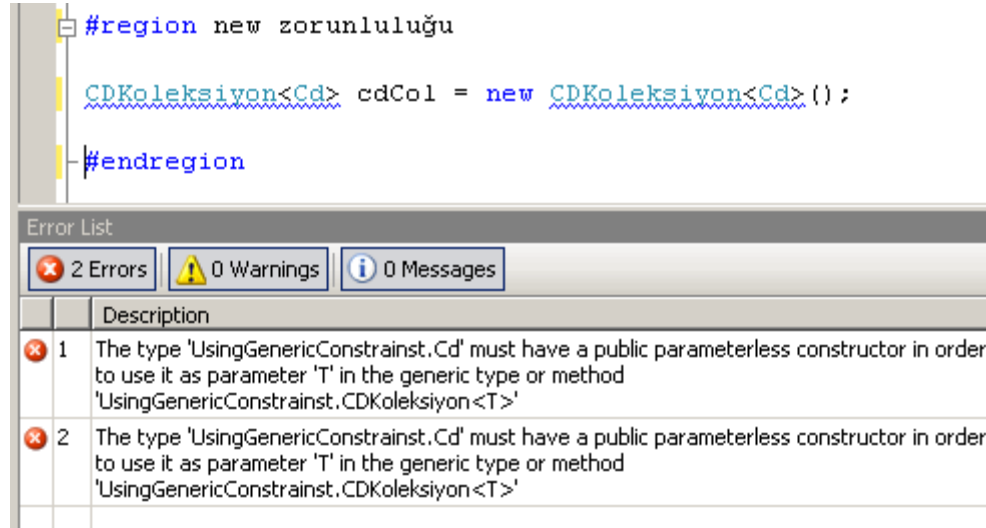
    public void Ekle(T cd)
    {
        icListe.Add(cd);
    }

    public T Oku(int indis)
    {
        return icListe[indis];
    }
}

public class Cd
{
    private int BarkodNo;
    public Cd(int id)
    {
```

```
BarkodNo = id;
}
}
```

Uygulamada CDKoleksiyon sınıfına ait bir nesne örneğini Cd tipini kullanacak şekilde oluşturmaya çalıştığımızda aşağıdaki ekran görüntüsünde verilen hata mesajlarını alırız. Buna göre, generic tipin mutlaka parametresiz bir constructor kullanması gerektiği derleme zamanında hata mesajı olarak bildirilmektedir.



Dilersek generic kısıtlamaların bir kaçını bir arada kullanabiliriz. Örneğin bir generic türün hem belli bir tipten gelmesini hemde struct olmasını sağlayabiliriz. Bu kombinasyonları arttırmamız mümkündür. Buradaki tek şart, eğer varsayılan yapıcı kısıtlamasıda var ise new anahtarının her zaman için en sonda belirtilmesi gerekliliğidir.

Bu sayede, zorlamaları kullanarak tip güvenliğini daha belirleyici şekilde sağlamış oluyoruz. Bu makalemizde generic mimarinin önemli özelliklerinden birisi olan kısıtlamaları incelemeye çalıştık. Kısıtlamalar yardımıyla tür bağımsızlığını kullanırken belirli şartların sağlanmasını zorunlu hale getirebileceğimizi gördük. Böylece geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşünceye dek hepinize mutlu günler dilerim.