İçindekiler

Maddeler

C Sharp hakkında temel bilgiler 1
İlk programımız 3
Değişkenler 5
Tür dönüşümü 13
Yorum ekleme 23
Operatörler 24
Akış kontrol mekanizmaları 34
Rastgele sayı üretme 47
Diziler 48
Metotlar 56
Sınıflar 72
Operatör aşırı yükleme 97
İndeksleyiciler 104
Yapılar 108
Enum sabitleri 109
İsim alanları 113
System isim alanı 124
O işlemleri 135
Temel string işlemleri 150
Kalıtım 162
Arayüzler 180
Partial (kısmi) tipler 188
İstisnai durum yakalama mekanizması 189
Temsilciler 202
Olaylar 209
Önişlemci komutları 218
Göstericiler 224
Assembly kavramı 234
Yansıma 238
Nitelikler 243
Örnekler 247
Şablon tipler 251
Koleksiyonlar 260
yield 273

Veri tabanı işlemleri 277
Form tabanlı uygulamalar 289

Kaynaklar

Madde Kaynakları ve Katkıda Bulunanlar 294
Resim Kaynakları, Lisanslar ve Katkıda Bulunanlar 295

Madde Lisansları

Lisans 296

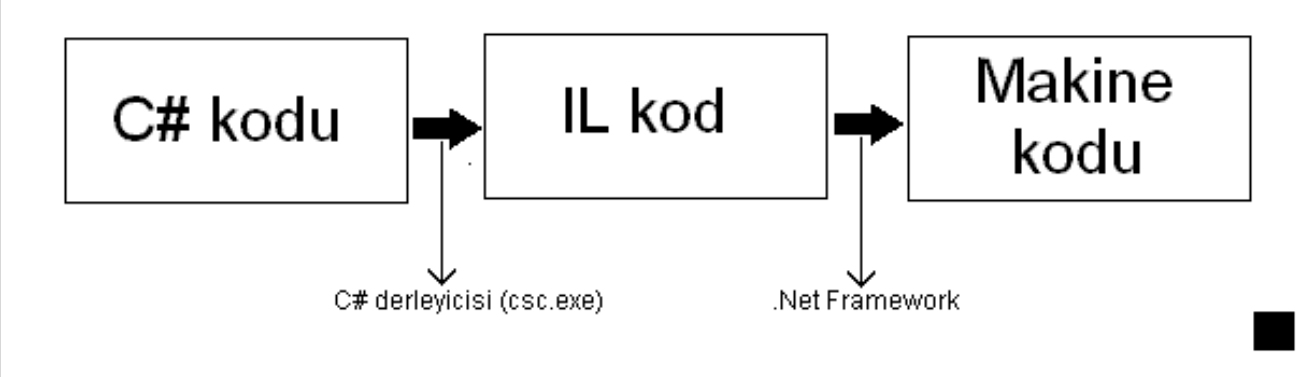
C Sharp hakkında temel bilgiler

C# (si şarp diye okunur) Microsoft tarafından geliştirilmiş olan bir programlama dilidir. C++ ve Java dillerine oldukça benzer, ancak C#'ın bu dillere benzerliği yanında farkları da vardır. Örneğin C#, C++'dan farklı olarak % 100 nesne yönelim tekniğine sahiptir. Java'dan farklı olarak ise C#'ta gösterici (pointer) kullanılabilir. Böylelikle eski yazılım bileşenleriyle uyumlu bir şekilde çalışılabilir.

.NET Framework

C# kodları, C++ veya Visual Basic'ten farklı olarak direkt makine koduna derlenmez. Önce *IL* dediğimiz bir ara koda derlenir. Bu derlenen ilk kodun dosyasına *assembly* denir ve uzantısı exe'dir. Bu dosya çalıştırılmak istendiğinde ise .Net Framework devreye girer ve IL kodu makine koduna dönüştürür, (alttaki şekle bakınız) böylelikle artık kodu bilgisayar anlayabilir. İşte bu yüzden de yazdığımız programın bir bilgisayarda çalışması için o bilgisayarda .Net Framework programının kurulu olması gerekir, çünkü .Net Framework IL kodu bilgisayarın anlayabileceği koda çevirir. .Net Framework, oluşturduğu makine kodlarını geçici bir süreliğine belleğe koyar, eğer aynı kodlar tekrar çalıştırılmak istenirse tekrar IL koddan makine koduna dönüşüm yapmak yerine bu belleğe kaydettiği makine kodlarını kullanır. Bu yüzden oluşturduğumuz programımızı ilk çalıştırdığımız zaman programımız biraz yavaş çalışabilir, ancak daha sonraki çalışmalarda oldukça hızlanacaktır.

C#'ta kodun direkt makine kodu yerine, önce IL koda çevrilmesinin bazı avantajları vardır. Bunlardan en önemlisi programımızın farklı işletim sistemlerinde çalışmasının eskiye oranla çok daha kolay olmasıdır. Çünkü makine kodu taşınabilir değildir, programları direkt makine koduna derlediğimiz zaman ilgili programın belirli bir işletim sistemine göre derlenmesi gerekir. Halbuki IL kod taşınabilirdir, ortak bir koddur, işletim sistemlerindeki çeşitli programlar vasıtasıyla makine koduna dönüştürülebilir. Örneğin Windows'ta bu işi .Net Framework yaparken, Linux'ta Mono yapabilir. Bu kitap C#'ı Windows üzerinden anlatacaktır. Ancak kitabın Linux'ta C# kullanımı kısmında ayrıca Linux'ta C# ile program geliştirme de anlatılacaktır.



C# kodlarını derleme

Son bir-iki bölüm dışındaki bütün kodları Not Defteri'nde (notepad) yazacağız. Aslında Visual Studio adında son derece gelişmiş ve işimizi son derece kolaylaştıran bir editör yazılımı var. Ancak bizim buradaki amacımız gelişmiş kurumsal yazılımlar geliştirmek yerine C#'ı tam anlamıyla öğrenmek olduğu için bu gelişmiş yazılımı kullanmayacağız. Not Deferi'nde yazdığımız kod dosyasına "cs" uzantısı verip .Net Framework programıyla birlikte gelen csc.exe derleyicisi ile derleyeceğiz. Bu derleyici komut satırında çalışıyor ve dolayısıyla da kodumuzun derlenmesi için komut satırında kod yazacağız. Yani .Net Framework programı hem kodumuzu derlemek için, hem de programımızın çalışması için gerekli. .Net Framework'u herhangi bir download sitesinden ya da Microsoft'un resmî sitesinden ücretsiz olarak indirip bilgisayarınıza kurabilirsiniz. Vista kullanıyorsanız çok büyük ihtimalle, XP kullanıyorsanız da bir ihtimalle .Net Framework siz de zaten kuruludur. Ancak işinizi garantiye almak isterseniz

veya son sürümünü edinmek isterseniz tekrar kurabilirsiniz.

.Net Framework'u kurduktan sonra bilgisayarımıza bir ayar yapmamız gerekecek. Bu ayara "path ayarlama" diyoruz. Bu ayarı yapmamızın sebebi komut satırında hangi klasörde olursak olalım csc.exe'ye erişebilmemiz. Bunun için şunları yapın:

- Bilgisayarım'a sağ tıklayın.
- "Özellikler"i seçin.
- "Gelişmiş" sekmesine gelin.
- "Ortam Değişkenleri" butonuna tıklayın.
- "Sistem değişkenleri" kısmındaki "Path"a çift tıklayın.
- Bu pencere burada kalsın, şimdi `C:\WINDOWS\Microsoft.NET\Framework` klasörüne gidin. Oradaki klasörlerin herbirinin içine tek tek bakın. Hangisinin içinde csc.exe varsa o klasörün adres çubuğundaki yolu kopyalayın.
- Şimdi önceki açtığımız "Sistem Değişkenini Düzenle" penceresinin "Değişken Değeri" kısmının sonuna ; işaretini koyup yolu yapıştırın.
- Bütün pencerelerden "tamam" diyerek çıkın.

Artık Not Defteri'nde yazdığımız kodu csc.exe derleyicisi ile derleyebiliriz. Örneğin "cs" uzantısı verdiğimiz kaynak kodumuzun adı "deneme.cs" olsun. Komut satırını açıp, kaynak kod dosyamızın olduğu klasörü aktif hâle getirip, `csc deneme.cs` yazıp entera basarak kodumuzu derleriz. Oluşan exe dosyamız kaynak kodumuzla aynı klasörde ve deneme.exe adıyla oluşur. Eğer aktif klasörde zaten deneme.exe diye bir dosya varsa eski dosya silinip yeni dosyamız kaydedilir. Programımızın kaynak kodun adından farklı bir adla oluşmasını istiyorsak

```
csc /out:YeniAd.exe deneme.cs
```

komutunu veririz. Bu örnekte programımız `YeniAd.exe` adıyla oluşacaktır.

C#'la yapabileceklerimiz

C#'la şunları yapabilirsiniz:

- Konsol uygulaması geliştirme
- Windows uygulaması geliştirme
- ASP.NET uygulaması geliştirme
- Web servisleri yazma
- Mobil uygulama geliştirme (PDA, cep telefonları vb. için)
- DLL yazma

Biz kitap boyunca konsol uygulamaları, son bölümlerde de form tabanlı uygulamalar geliştireceğiz. Ancak kullandığınız dilin ne kadar güçlü bir dil olduğunu öğrenmeniz açısından bunları bilmenizde fayda var.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

İlk programımız

Artık kod yazmaya başlıyoruz. İlk programımız şu şekilde:

```
class ilkprogram
{
    static void Main()
    {
        System.Console.WriteLine("Merhaba dünya!");
    }
}
```

Bu program konsol ekranına `Merhaba dünya!` yazıp kapanır. Programı geçen derste anlatıldığı gibi derleyin, sonra da programımızı çalıştırmak için komut satırında kod dosyanıza verdiğiniz adı girip entera basın.

İlk programımızın satır satır incelenmesi

- `class ilkprogram` satırıyla *ilkprogram* adında yeni bir sınıf oluştururuz. C#'ta yazdığımız her programın en az bir sınıf içermesi zorunludur.
- `{` veya `}` işaretleri herhangi bir sınıfın veya metodun içeriğini belirtmek için kullanılır. İlk `{` karakteriyle önceki satırda açtığımız `ilkprogram` adlı sınıfımızın içeriğine alınacak kodların başladığını, programın son satırındaki `}` karakteriyle de sınıfımızın içeriğine alınacak kodların sona erdiğini belirtiyoruz.
- `static void Main()` satırıyla sınıfımızın içine `Main` adlı bir metot yerleştirdik. Bu metodun adı mutlaka `Main` olmalı, yoksa programımız çalışmaz. Ayrıca bu metot mutlaka oluşturduğumuz sınıfın içinde olmalı. Yine `{` ve `}` karakterleriyle metodumuzun içeriğini belirledik. Dikkat ettiyseniz bir iç içe `{` ve `}` karakterleri söz konusu. Bu durumda koyulan ilk `}` karakteri son açılan `{` karakterini kapatır.
- Programımızın pratikte iş yapan tek kısmı ise `System.Console.WriteLine("Merhaba dünya!");` satırı. Bu satırla ekrana `Merhaba dünya!` yazdırdık, peki nasıl yaptık? Bunun için .Net Framework kütüphanesindeki hazır bir metottan yararlandık. Bu metot, `System` isim alanının altındaki `Console` sınıfında bulunuyor, ismi `WriteLine` ve konsol ekranına yazı yazdırmaya yarıyor. Parantezler arasındaki çift tırnaklar arasına alınan metni ekrana yazdırıyor. Satırın sonundaki `;` karakterini ise `{` ve `}` karakterleri açıp kapatmayan bütün C# satırlarında kullanmamız gerekiyor.

Programımızın ikinci versiyonu

```
using System;
class ilkprogram
{
    static void Main()
    {
        Console.WriteLine("Merhaba dünya!");
    }
}
```

Bu programımızın işlevsel olarak ilk programımızdan herhangi bir farkı yok. Yani ikisi de aynı şeyi yapıyor. Ancak kodda bir farklılık var. `Using System;` satırı oluşmuş ve `System.Console.WriteLine("Merhaba dünya!");` satırındaki `System` kalkmış. `using` deyimi C#'ta isim alanı kullanma hakkı elde etmek için kullanılan bir anahtar sözcüktür. Yani aynı isim alanında kullanacağımız birden fazla metot varsa bu isim alanını `using` anahtar sözcüğüyle belirtmemiz son derece mantıklı.

Programımızın üçüncü versiyonu

```
using System;
 class ilkprogram
 {
    static void Main()
    {
       Console.WriteLine("Entera basın!");
       Console.ReadLine();
       Console.WriteLine("Entera bastınız!");
    }
 }
```

Bu programımız önce ekrana `Entera basın!` yazar. Kullanıcı entera bastığında da `Entera bastınız!` yazıp kendini kapatır. `Console` sınıfına ait olan `ReadLine` metodu programımızın kullanıcıdan bilgi girişi için beklemesini sağlar, yani programımızı entera basılana kadar bekletir. Kullanıcı entera bastığında da diğer satıra geçilir. Dikkat ettiyseniz `ReadLine` metodunda parantezlerin arasına hiçbir şey yazmıyoruz. C#'ta bazı metotların parantezleri arasına birşeyler yazmamız gerekirken, bazı metotlarda da hiçbir şey yazılmaması gerekir.

Bu kitabın diğer sayfaları

- C# hakkında temel bilgiler
- İlk programımız
- Değişkenler
- Tür dönüşümü
- Yorum ekleme
- Operatörler
- Akış kontrol mekanizmaları
- Rastgele sayı üretme
- Diziler
- Metotlar
- Sınıflar
- Operatör aşırı yükleme
- İndeksleyiciler
- Yapılar
- Enum sabitleri
- İsim alanları
- System isim alanı
- Temel I/O işlemleri
- Temel string işlemleri
- Kalıtım
- Arayüzler
- Partial (kısmi) tipler
- İstisnai durum yakalama mekanizması
- Temsilciler
- Olaylar
- Önişlemci komutları
- Göstericiler
- Assembly kavramı
- Yansıma
- Nitelikler
- Örnekler
- Şablon tipler
- Koleksiyonlar
- yield
- Veri tabanı işlemleri
- XML işlemleri
- Form tabanlı uygulamalar
- Visual Studio.NET
- Çok kanallı uygulamalar
- Linux'ta C# kullanımı
- Kaynakça
- Giriş sayfası
- **Ayrıca Bakınız:**

  ASP.NET

Değişkenler

Program yazarken her zaman sabit verilerle çalışmayız, çoğu zaman programımızda bir verinin kullanıcının davranışına göre değişmesi gerekir. Kullanıcıdan bir metin alıp bunu ekrana yazdıran bir program buna örnek verilebilir. Değişken kısaca bellek gözeneklerinin programlamadaki karşılıklarıdır.

C#'ta değişken tanımlama

Çoğu programlama dilinde değişkenler kullanılmaya başlanmadan önce tanımlanırlar. Aşağıdaki şekli inceleyiniz.

Yukarıdaki şekilde C#'ta değişken tanımlamanın nasıl yapıldığı anlatılmıştır. Değişken türü bellekte ayrılan gözeneğin büyüklüğünü belirtmemizi sağlar. Değişken adı da bu gözeneğe verdiğimiz adı belirtir. Doğal olarak bu gözenekteki veriye erişmek istediğimizde veya bu gözenekteki veriyi değiştirmek istediğimizde bu adı kullanacağız. Yukarıdaki şekilde -2,147,483,648 ile 2,147,483,647 arasında (sınırlar dâhil) bir değer tutabilen ve adı "ad" olan bir bellek gözeneği oluşturduk.

Değişkenlere değer atama

Çoğu programlama dilinde değişkenler tanımlandıktan sonra direkt olarak programda kullanılabilirler. Ancak C#'ta değişkeni tanımladıktan sonra ayrıca bir de ilk değer atamak zorundayız. Aksi bir durumda değişkeni programımız içinde kullanamayız. Değişkenlere değer atama şöyle yapılır:

```
ad=5;
```

Burada ad değişkenine 5 değerini atadık. Bu en basit değer atama yöntemidir. Ayrıca şunlar da mümkündür:

```
int a=5;
int b, c, d, e;
int f=10, g, m=70;
```

Birinci satırda tanımlama ve değer vermeyi aynı satırda yaptık. İkincisinde aynı türden birden fazla değişken tanımladık. Üçüncü satırda ise tanımladığımız değişkenlerin bazılarına değer verirken bazılarına vermedik.

Değişken türleri

Yukarıda değişken tanımlarken değişken türü için `int` kullanmıştık. C#'ta bunun gibi farklı kapasitelere sahip bir hayli daha değişken türü vardır. Ayrıca bazı değişken türleri sayısal, bazıları da metinseldir. Sayısal türler aşağıdaki tabloda listelenmiştir:

Tür	Boyut	Kapasite	Örnek
byte	1 bayt	0, ..., 255 (tam sayı)	`byte a=5;`
sbyte	1 bayt	-128, ..., 127 (tam sayı)	`sbyte a=5;`
short	2 bayt	-32768, ..., 32767 (tam sayı)	`short a=5;`
ushort	2 bayt	0, ..., 65535 (tam sayı)	`ushort a=5;`
int	4 bayt	-2147483648, ..., 2147483647 (tam sayı)	`int a=5;`
uint	4 bayt	0, ..., 4294967295 (tam sayı)	`uint a=5;`
long	8 bayt	-9223372036854775808, ..., 9223372036854775807 (tam sayı)	`long a=5;`
ulong	8 bayt	0, ..., 18446744073709551615 (tam sayı)	`ulong a=5;`
float	4 bayt	$\pm1.5*10^{-45}$, ..., $\pm3.4*10^{38}$ (reel sayı)	`float a=5F;` veya `float a=5f;`
double	8 bayt	$\pm5.0*10^{-324}$, ..., $\pm1.7*10^{308}$ (reel sayı)	`double a=5;` veya `double a=5d;` veya `double a=5D;`
decimal	16 bayt	$\pm1.5*10^{-28}$, ..., $\pm7.9*10^{28}$ (reel sayı)	`decimal a=5M;` veya `decimal a=5m;`

Dikkat ettiyseniz bazı değişken türlerinde değer atarken değerin sonuna bir karakter eklenmiş, bu değişken türlerindeki değişkenlere değer atarken siz de bunlara dikkat etmelisiniz. Sıra geldi metinsel türlere:

Tür	Boyut	Açıklama	Örnek
char	2 bayt	Tek bir karakteri tutar.	`char a='h';`
string	Sınırsız	Metin tutar.	`string a="Ben bir zaman kaybıyım, beni boşver hocam";`

String türüne ayrıca char ve/veya string sabit ya da değişkenler + işaretiyle eklenip atanabilir. Örnekler:

```
char a='g';
string b="deneme";
string c=a+b+"Viki"+'m';
```

C#'ta hem metinsel hem de sayısal olmayan türler de vardır:

bool

Koşullu yapılarda kullanılır. Bool türünden değerlere `true`, `false` veya `2<1` gibi ifadeler örnek verilebilir. Örnekler:

```
bool b1=true;
bool b2=false;
bool b3=5>4;
```

object

Bu değişken türüne her türden veri atanabilir. Örnekler:

```
object a=5;
object b='k';
object c="metin";
object d=12.7f;
```

Aslında C#'taki bütün değişken türleri `object` türünden türemiştir. Bu yüzden `object` türü diğerlerinin taşıdığı bütün özellikleri taşır. Ancak şimdilik bunu düşünmenize gerek yok. Bu, nesneye dayalı programlamanın özellikleriyle ilgili.

Değişkeni programımız içinde kullanma

Şimdiye kadar değişkenleri tanımlayıp ilk değer verdik. Şimdi değişkenleri programımızda kullanmanın zamanı geldi. Bir örnek:

```
using System;
class degiskenler
{
   static void Main()
   {
      byte a=5;
      Console.WriteLine(a);
   }
}
```

Burada `a` değişkeninin değerini ekrana yazdırdık. Başka bir örnek:

```
using System;
class degiskenler
{
   static void Main()
   {
      byte a=5;
      byte b=8;
      Console.WriteLine(a+b);
   }
}
```

Bu programda iki değişkenimizin değerlerinin toplamını ekrana yazdırdık.

```
using System;
class degiskenler
{
   static void Main()
   {
      string a="Viki", b="kitap";
      Console.WriteLine(a+b);
   }
}
```

Bu programda aynı satırda iki tane `string` değişkeni tanımladık ve değer verdik. Bu değişkenlerin değerlerini `WriteLine` metoduyla ekrana yan yana yazdırdık. `WriteLine` metodu `+` işaretini gördüğünde sayısal değişken ve değerleri toplar, string türünden değişken ve değerleri yan yana yazar, `char` türünden değişken ve değerlerin Unicode karşılıklarını toplar. Ancak tabii ki `+` karakterinin ayırdığı değişken veya değerler `char` ile `string`se char karakterle string metni yan yana yazar.

```
using System;
class degiskenler
{
   static void Main()
   {
      string a;
      a=Console.ReadLine();
      Console.WriteLine(a+" metnini yazdınız.");
   }
}
```

Sanırım şimdiye kadar yazdığımız en gelişmiş program buydu. Bu program kullanıcıdan bir metin alıp bunu ekrana "... metnini yazdınız." şeklinde yazıyor. Geçen derste `ReadLine` metodunu kullanıcı entera basana kadar programı bekletmek için kullanmıştık. Aslında `ReadLine` metodunun en yaygın kullanımı kullanıcının bilgi girişi yapmasını sağlamaktır. Dikkat ettiyseniz programımızda kullanıcının girdiği bilgi `a` değişkenine atanıyor. Sonra da `WriteLine` metoduyla ekrana bu değişken ve `" metnini yazdınız."` metni yan yana yazdırılıyor. Burada asıl önemsememiz gereken şey `Console.ReadLine()` ifadesinin string türünden bir değer gibi kullanılabilmesidir. C#'ta bunun gibi birçok metot bir değer gibi kullanılabilir. Tahmin edebileceğiniz üzere `WriteLine` gibi birçok metot da bir değer gibi kullanılamaz. Başka bir örnek:

```
using System;
class degiskenler
{
   static void Main()
   {
      int a, b;
      a=20;
      b=a;
      Console.WriteLine(b);
   }
}
```

Bu programda da görebileceğiniz gibi değişkenlere değer olarak başka bir değişkeni atayabiliriz. Ancak değerini atadığımız değişkene daha önceden bir değer atanmış olması gerekiyor. Burada `b` değişkenine `a` değişkeninin değeri atanıyor. Ancak aşağıdaki gibi bir kullanım kesinlikle hatalıdır.

```
using System;
class degiskenler
{
   static void Main()
   {
      Console.ReadLine()="metin";
      string a=Console.ReadLine();
      Console.WriteLine(a);
   }
}
```

Burada `Console.ReadLine()` ifadesi bir değişkenmiş gibi kullanılmaya çalışılıyor, ancak hatalı. Çünkü `Console.ReadLine()` ifadesi yalnızca bir değermiş gibi kullanılabilir.

Değişken adlandırma kuralları

Şimdiye kadar değişkenlere `ad`, `a` veya `b` gibi basit adlar verdik. Ancak aşağıdaki kuralları ihlal etmemek şartıyla değişkenlere istediğiniz adı verebilirsiniz.

- Değişken adları boşluk, simge içeremez.
- Değişkenler bir numerik karakterle başlayamaz.
- C#'ın diğer bütün komut, metot ve benzerlerinde olduğu gibi değişken adlarında büyük-küçük harf duyarlılığı vardır. Yani `degisken` isimli bir değişkenle `Degisken` isimli bir değişken birbirinden farklıdır.
- Değişken adları Türkçe karakterlerden(ğ,ü,ş,ö,ç,ı) oluşamaz.

Sık yapılan hatalar

C#'ta değişkenlerle ilgili sık yapılan hatalar şunlardır:

- Aynı satırda farklı türden değişkenler tanımlamaya çalışma. Örneğin aşağıdaki örnek hatalıdır:

```
int a, string b;
```

- Değişkene uygunsuz değer vermeye çalışma. Örnek:

```
int a;
a="metin";
```

- Değişkeni tanımlamadan ve/veya değişkene ilk değer vermeden değişkeni kullanmaya çalışma. Aşağıdaki örnekte iki değişkenin de kullanımı hatalıdır.

```
using System;
class degiskenler
{
   static void Main()
   {
      int b;
      Console.WriteLine(a);
      Console.WriteLine(b);
   }
}
```

- Değişken tanımlaması ve/veya değer vermeyi yanlış yerde yapma. Örnek:

```
using System;
class degiskenler
{
   int a=5;
   static void Main()
   {
      Console.WriteLine(a);
   }
}
```

Diğer `using` dışındaki bütün komutlarda da olduğu gibi değişken tanım ve değer vermelerini de `Main` bloğunun içinde yapmalıyız.

- Bazı değişken türlerindeki değişkenlere değer verirken eklenmesi gereken karakteri eklememek. Örnek:

```
using System;
class degiskenler
{
   static void Main()
   {
      float a=12.5;
      Console.WriteLine(a);
   }
}
```

- Ondalık sayıların ondalık kısmını ayırırken nokta (.) yerine virgül (,) kullanmak. Örnek:

```
using System;
class degiskenler
{
   static void Main()
   {
      float a=12,5f;
      Console.WriteLine(a);
   }
}
```

- Metinsel değişkenlerle matematiksel işlem yapmaya çalışmak. Örnek:

```
using System;
class degiskenler
{
   static void Main()
   {
      string a="1", b="2";
      int c=a+b;
      Console.WriteLine(a);
   }
}
```

- Bir değişkeni birden fazla kez tanımlamak. Örnek:

```
using System;
class degiskenler
{
   static void Main()
   {
      string a;
      string a="deneme";
      Console.WriteLine(a);
   }
}
```

- Değişkenlere değer verirken yanlış şekilde değer vermek. Örnek:

```
using System;
class degiskenler
{
   static void Main()
   {
      string a=deneme;
      Console.WriteLine(a);
   }
}
```

Sabit değişkenler

Programımızda bazen değeri hiç değişmeyecek değişkenler tanımlamak isteyebiliriz. Örneğin `pi` isimli float türünden bir değişken tanımlayıp buna 3.14 değerini verip programımızda pi sayısına ihtiyaç duyulduğunda bu değişkeni kullanabiliriz. Sabit değişkenlerin normal değişkenlerden farkı değişkeni değiştirmek istediğimizde ortaya çıkar, sabit olarak belirtilen değişkeni değiştirirsek derleyici hata verip programımızı derlemez. Bu daha çok uzun program kodlarında işe yarayabilir. Ayrıca sabit değişkenlere tanımlandığı satırda değer vermeliyiz. Herhangi bir değişkeni sabit olarak belirtmemiz için değişken türünden önce `const` anahtar sözcüğü kullanılır. Örnekler:

```
const int a = 5;
const string b ="deneme";
const char c ='s';
```

Aşağıdaki gibi bir durum, değişkene tanımlandığı satırda değer verilmediği için hatalıdır.

```
const int a;
a=5;
```

Sabit değişkenlere değer olarak sabit, sabit değişken ya da sabit ve/veya sabit değişkenlerden oluşan matematiksel ifadeler verilebilir. Örnek:

```
const int a=5;
const int b=a+7;
const int c=a*b;
```

Aşağıdaki gibi bir durum hatalıdır.

```
int a=5;
const int b=a+8;
```

Escape sequence (\) kullanımı

Bir string sabitin içinde özel karakterler olması için escape sequence kullanılır. Örnekler:

```
string ad="Deneme\"Deneme";
Console.WriteLine(ad);
```

Bu satırda " karakterinin ad değişkenine atanan string sabitin içine koyulmasını sağladık. Yukarıdaki kod ekrana `Deneme"Deneme` yazar. Başka bir örnek:

```
string yol="Windows\\Program Files";
```

Burada bir illegal karakter olan \ karakterinin başına tekrar \ koyarak stringin bir tane \ almasını sağladık.

```
string yol=@"Windows\Program Files";
```

Burada yol değişkenine tırnak içerisindeki metin olduğu gibi aktarılır, ancak doğal olarak " karakterinde işe yaramaz.

```
Console.WriteLine("Satır\nYeni satır\nYeni satır");
```

Örnekte de gördüğünüz üzere C#'ta `\n` yeni satır yapmak için kullanılır.

Ek bilgiler

Burada değişkenlerle ilgili olan ancak herhangi bir başlıkta incelenemeyecek olan önemli bilgiler bulunmaktadır. Bunların bazıları kendiniz çıkarabileceğiniz mantıksal bilgilerdir.

- `ReadLine()` metodunun tuttuğu değer string türündedir.
- `WriteLine()` metodunda parantezler arasına girilen değer `object` türünden olmalıdır. Yani herhangi bir türden değer yazılabilir.
- Bütün değişkenler bir değermiş gibi kullanılabilir ancak değerler değişkenmiş gibi kullanılamaz.
- Eğer bir değişkeni tanımlamış veya tanımlayıp değer vermiş, ancak programımızın hiçbir yerinde kullanmamışsak derleyici hata vermez, sadece bir uyarı verir.
- `"deneme"+"yalnızlık"+'d'+'m'` gibi bir ifade aslında string türünden bir sabittir.
- (3+8/9)*6 gibi bir ifade, `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` veya `double` türünden değişkenlere atanabilir.
- Aslında C#'ta sabitlerin de türleri vardır. Mesela:
  - "5" string türünden
  - '5' char türünden
  - 5f float türünden
  - 12.7 veya 5d veya 5D double türünden (C# harfi olmayan ondalıklı sayıları double sayar.)
  - 5m decimal türünden
  - 5 ise int türündendir. Çünkü C# herhangi bir ayırt edici özelliği bulunmayan tam sayı sabitleri `int` türünden sayar. Ancak 5'in türünün int olması 5'in byte türünden bir değişkene atanamayacağı anlamına gelmez. int türünden sabitler byte, sbyte, short, ushort ve uint türünden değişkenlere de atanabilirler.
- `5+"deneme"+6.7f` gibi bir ifade aslında `object` türünden bir sabittir.
- C#'ta herhangi bir sabit ya da değişkenin türünü anlamak için `x.GetType()` metodu kullanılır. Örnek:

```
Console.WriteLine(5.GetType());
Console.WriteLine(14.4.GetType());
Console.WriteLine("deneme".GetType());
byte a=2;
Console.WriteLine(a.GetType());
```

Ancak GetType() metodunun tuttuğu değer Type türündedir, dolayısıyla herhangi bir string türündeki değişkene vs. atanamaz, ayrıca da `GetType()` metodu değişkenin CTS'deki karşılığını verir. Değişkenlerin CTS karşılıkları bir sonraki dersimizin konusudur.

- Sabitin diğer adı değerdir.
- `a=b` ifadesi atama işlemidir. b'nin değeri a'ya atanır. b sabit ya da değişken olabilir. Ancak a kesinlikle değişken olmalıdır.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Tür dönüşümü

Modern programlamada birçok kez değişkenlerde tür dönüşümüne ihtiyaç duyulur. Örneğin string türündeki sayılarla ("5" veya "2" gibi) matematiksel işlem yapmamız gerektiğinde tür dönüşümü yapmamız gerekir. Aslında bahsettiğimiz tam olarak tür dönüşümü değildir, sadece bir değişkenin değişik türdeki hâlinin başka bir değişkene atanmasıdır. Tür dönüşümleri bilinçli tür dönüşümü ve bilinçsiz tür dönüşümü olmak üzere ikiye ayrılır.

Bilinçsiz tür dönüşümü

Aslında bunu önceki derste görmüştük. Bu derste sadece adını koyuyoruz ve tanımlıyoruz: C#'ta düşük kapasiteli bir değişken, sabit ya da değişken ve sabitlerden oluşan matematiksel ifade daha yüksek kapasiteli bir değişkene atanabilir. Buna bilinçsiz tür dönüşümü denir, bunun için herhangi bir özel kod gerekmez.

```
using System;
class TurDonusumu
{
	static void Main()
	{
		byte a=5;
		short b=10;
		sbyte c=30;
		int d=a+b+c;
		string e="deneme";
		char f='k';
		object g=e+f+d;
		long h=d;
		float i=h;
		double j=i;
		double k=12.5f;
		Console.WriteLine(j+k);
	}
}
```

Bilinçsiz tür dönüşümüyle ilgili ilginç bir durum söz konusudur. `char` türünü kendisinden daha kapasiteli bir sayısal türe bilinçsiz olarak dönüştürebiliriz. Bu durumda ilgili karakterin Unicode karşılığı ilgili sayısal değişkene atanacaktır.

```
using System;
class TurDonusumu
{
	static void Main()
	{
		char a='h';
		int b=a;
		Console.WriteLine(b);
	}
}
```

Bilinçsiz tür dönüşümüyle ilgili diğer ilginç bir nokta ise `byte`, `sbyte`, `short` ve `ushort` türündeki değişkenlerle yapılan matematiksel işlemlerdir. Bu tür durumda oluşan matematiksel ifade intleşir. Yani aşağıdaki durumda programımız hata verir:

```
using System;
class TurDonusumu
{
	static void Main()
	{
		byte a=5, b=10;
		short d=2, e=9;
		byte f=a+b;
		short g=d+e;
		Console.WriteLine(f+g);
	}
}
```

Çünkü artık 8. satırdaki a+b ifadesi intleşmiş, ayrıca da 9. satırdaki d+e ifadesi de intleşmiştir, ve bunları en az int türdeki bir değişkene atayabiliriz. Size yardımcı olması açısında bilinçsiz tür dönüşümü yapılacabilecekler tablosu aşağıda verilmiştir:

Kaynak	Hedef
sbyte	short, int, float, long, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long, ulong	float, double, decimal
char	ushort, int, uint, long, ulong, float, double, decimal
float	double

Bilinçli tür dönüşümü

Bilinçli tür dönüşümü genellikle derleyicinin izin vermediği durumlarda kullanılır. Bilinçli tür dönüşümüyle küçük türün büyük türe dönüştürülmesi sağlanabilse de aslında bu gereksizdir, çünkü aynı şeyi bilinçsiz tür dönüşümüyle de yapabilirdik. Aşağıdaki programda bilinçli tür dönüşümü gerçekleştirilmektedir.

```
using System;
class TurDonusumu
{
	static void Main()
	{
		int a=5;
		byte b=(byte)a;
		Console.WriteLine(b);
	}
}
```

Programımızda da görebileceğiniz gibi `(byte)a` ifadesi, a değişkeninin byte hâlini tuttu. Aşağıdaki şekil bilinçli tür dönüşümünü anlatmaktadır.

- Eğer değişken adı kısmında tek bir değişken yoksa, bir ifade varsa parantez içine alınması gerekir.
- Bu şekilde tür dönüşümü değişkenlere uygulanabildiği gibi sabitlere de uygulanabilir:

```
using System;
class TurDonusumu
{
	static void Main()
	{
		byte b=(byte)12.5f;
		Console.WriteLine(b);
	}
}
```

- Reel türler tam sayı türlere dönüşürken ondalık kısım atılır.
- Bilinçsiz tür dönüşümüyle yalnızca küçük türler büyük türlere dönüşebiliyordu, yani veri kaybı olması imkansızdı. Halbuki bilinçli tür dönüşümünde veri kaybı gerçekleşebilir, eğer dönüşümünü yaptığımız değişkenin tuttuğu değer dönüştürülecek türün kapasitesinden büyükse veri kaybı gerçekleşir. Bu gibi durumlar için C#'ın `checked` ve `unchecked` adlı iki anahtar sözcüğü vardır.

checked

Kullanımı aşağıdaki gibidir:

```
using System;
class TurDonusumu
{
	static void Main()
	{
		int i=256;
		byte b;
		checked
		{
			b=(byte)i;
		}
		Console.WriteLine(b);
	}
}
```

i, byte'a çevrilirken veri kaybı gerçekleştiği için programımız çalışma zamanında hata verecektir. Ayrıca yeri gelmişken bir konuya değinmek istiyorum. Bu programımızda b değişkeni `checked` bloklarının içinde tanımlansaydı `checked` bloklarının dışında bu b değişkenine erişemezdik, çünkü değişkenler yalnızca tanımlandıkları en iç bloğun içinde aktiftirler. Buradaki "en iç" sözüne dikkat etmenizi öneririm.

```
{
	int b;
	{
		int a;
	}
}
```

Burada a değişkeni yalnızca en iç blokta aktif olmasına karşın, b değişkeni hem dıştaki hem içteki blokta aktiftir.

```
{
	{
		int a=0;
	}
	{
		int b=0;
	}
	Console.WriteLine(a+b);
}
```

Burada her iki değişken de `WriteLine` satırının olduğu yerde geçersiz olduğu için bu kod da geçersizdir. Çünkü her iki değişken de tanımlandıkları en iç bloğun dışında kullanılmaya çalışılıyor.

unchecked

Eğer programımızda veri kaybı olduğunda programın hata vermesini istemediğimiz kısımlar varsa ve bu kısımlar `checked` bloklarıyla çevrilmişse `unchecked` bloklarını kullanırız. `unchecked`, `checked`'ın etkisini bloklarının içeriği çerçevesinde kırar. Programımızın varsayılan durumu `unchecked` olduğu için, yalnızca eğer programımızda `checked` edilmiş bir kısım var ve bu kısım `unchecked` etmek istediğimiz alanı kapsıyorsa gereklidir, diğer durumlarda gereksizdir. Örnek:

```
using System;
class TurDonusumu
{
	static void Main()
	{
		int i=256;
		int a=300;
		byte b, c;
		checked
		{
			b=(byte)i;
			unchecked
			{
				c=(byte)a;
			}
		}
		Console.WriteLine(b);
	}
}
```

Programımız çalışma zamanında hata verir, ancak bunun sebebi `c=(byte)a;` satırı değil, `b=(byte)i;` satırıdır.

object türüyle ilgili kurallar

C#'taki en ilginç türün `object` olduğunu geçen derste söylemiştik, işte bu ilginç türün daha başka kuralları:

- object türündeki bir değişkene başka herhangi bir türdeki değişken ya da sabit (string dışında) + işaretiyle eklenemez. Örneğin aşağıdaki gibi bir durum hatalıdır:

```
using System;
class TurDonusumu
{
	static void Main()
	{
		object a='5';
		int b=4;
		Console.WriteLine(a+b);
	}
}
```

- object türündeki bir değişkene herhangi bir türdeki değişken ya da sabit atanabilir (bilinçsiz tür dönüşümü). Ancak object türündeki herhangi bir değişkeni başka bir türe çevirmek için tür dönüştürme işlemi kullanılır.

Ayrıca da dönüştürülen türlerin uyumlu olması gerekir. Yani object türüne nasıl değer atandığı önemlidir. Örneğin aşağıdaki gibi bir durum hatalıdır:

```
using System;
class TurDonusumu
{
	static void Main()
	{
		object a="5";
		int b=(int)a;
		Console.WriteLine(b);
	}
}
```

Ancak aşağıdaki gibi bir kullanım doğrudur.

```
using System;
class TurDonusumu
{
	static void Main()
	{
		object a='k';
		char b=(char)a;
		Console.WriteLine(b);
	}
}
```

Aynı satırda çifte dönüşüm yapılamaz. Aşağıdaki gibi bir kullanım hatalıdır.

```
using System;
class TurDonusumu
{
	static void Main()
	{
		object a='c';
		int b=(int)a;
		Console.WriteLine(b);
	}
}
```

Burada önce objectin chara, sonra da charın inte dönüşümü ayrı ayrı yapılmalıydı. Aşağıdaki gibi bir kullanım hatalıdır:

```
using System;
class TurDonusumu
{
	static void Main()
	{
		object a=5;
		byte b=(byte)a;
		Console.WriteLine(b);
```

```
        }
}
```

Çünkü derleyici harfsiz tam sayı sayıları int sayar, dolayısıyla da burada uygunsuz bir tür dönüşümünden bahsedebiliriz. Yani 7. satırda a inte dönüştürülmeliydi. Ancak aşağıdaki gibi bir kullanım doğrudur:

```
using System;
class TurDonusumu
{
        static void Main()
        {
                byte m=5;
               object a=m;
                byte b=(byte)a;
                Console.WriteLine(b);
        }
}
```

Bu program hata vermez. Çünkü derleyici a'nın gizli türünün byte olduğunu biliyor.

string türüyle ilgili dönüşümler

Dikkat ettiyseniz şimdilik sadece sayısal türleri, char ve object (uygun şekilde değer verilmiş olması şartıyla) türünü kendi aralarında dönüştürebiliyoruz. Şimdi göreceğimiz metotlarla string türünü bu türlere ve bu türleri de string türüne dönüştürebileceğiz.

x.ToString()

Bu metot x değişken ya da sabitini stringe çevirip tutar. Örnekler:

```
using System;
class TurDonusumu
{
        static void Main()
        {
                string b=3.ToString();
                Console.WriteLine(b);
        }
}
```

veya

```
using System;
class TurDonusumu
{
        static void Main()
        {
                int a=6;
                string b=a.ToString();
                Console.WriteLine(b);
        }
```

```
}
```

veya

```
using System;
class TurDonusumu
{
	static void Main()
	{
		string b=12.5f.ToString();
		Console.WriteLine(b);
	}
}
```

veya

```
using System;
class TurDonusumu
{
	static void Main()
	{
		string b='k'.ToString();
		Console.WriteLine(b);
	}
}
```

`ToString()` metodu `System` isim alanında olduğu için programımızın en başında `using System;` satırının bulunması bu metodu kullanabilmemiz için yeterlidir. string türüyle ilgili bilmeniz gereken birşey daha var:

```
using System;
class TurDonusumu
{
	static void Main()
	{
		string b=5+"deneme"+'k'+12.5f;
		Console.WriteLine(b);
	}
}
```

Böyle bir kod mümkündür. Ayrıca şöyle bir kod da mümkündür:

```
using System;
class TurDonusumu
{
	static void Main()
	{
		int a=5;
		char m='k';
		string r="deneme";
		float f=12.5f;
		string b=a+m+r+f;
```

```
            Console.WriteLine(b);
      }
}
```

Burada dikkat etmemiz gereken şey yan yana gelen sayısal ve char değer ve değişkenlerle matematiksel toplama yapılmasına rağmen bunların yan yana gelmemesi durumunda da normal yan yana yazma olayının gerçekleşmesidir. Ancak tabii ki şöyle bir durum hatalıdır:

```
using System;
class TurDonusumu
{
      static void Main()
      {
            int a=5;
            char m='k';
            float f=12.5f;
            string b=a+m+f;
            Console.WriteLine(b);
      }
}
```

Yani bir sabit ve/veya değişken grubunun bir stringe atanabilmesi için ifadenin içinde en az bir string olmalıdır.

System.Convert sınıfı ile tür dönüşümü

`System` isim alanının altındaki `Convert` sınıfı içinde tür dönüşümü yapabileceğimiz birçok metot bulunur. Bu metotlarla hemen hemen her türü her türe dönüştürebiliriz. Ancak bunun için değişken türlerinin CTS karşılıklarını bilmeliyiz. Değişken türleri ve CTS karşılıkları aşağıdaki tabloda listelenmiştir.

Tür	CTS karşılığı
bool	Boolean
byte	Byte
sbyte	Sbyte
short	Int16
ushort	UInt16
int	Int32
uint	UInt32
long	Int64
ulong	UInt64
float	Single
double	Double
decimal	Decimal
char	Char

Şimdi sıra geldi bu metotlara:

- Convert.ToBoolean(x)
- Convert.ToByte(x)

- Convert.ToSbyte(x)
- Convert.ToInt16(x)
- Convert.ToUInt16(x)
- Convert.ToInt32(x)
- Convert.ToUInt32(x)
- Convert.ToInt64(x)
- Convert.ToUInt64(x)
- Convert.ToSingle(x)
- Convert.ToDouble(x)
- Convert.ToDecimal(x)
- Convert.ToChar(x)

Bu metotlar x'i söz konusu türe dönüştürüp o türde tutarlar. x bir değişken, sabit ya da ifade olabilir. x, ifade olduğu zaman önce bu ifade mevcut türe göre işlenir, sonra tür dönüşümü gerçekleşir. Kullanımına bir örnek:

```
using System;
class TurDonusumu
{
       static void Main()
       {
              string s1, s2;
              int sayi1, sayi2;
              int Toplam;
              Console.Write("Birinci sayı girin: ");
              s1=Console.ReadLine();
              Console.Write("İkinci sayıyı girin: ");
              s2=Console.ReadLine();
              sayi1=Convert.ToInt32(s1);
              sayi2=Convert.ToInt32(s2);
              Toplam=sayi1+sayi2;
              Console.Write("Toplam= "+Toplam);
      }
}
```

Bu programla kullanıcıdan iki sayı alıp bunların toplamını ekrana yazdırdık. Ayrıca burada şimdiye kadar kullanmadığımız `Write` metodunu kullandık. Bu metodun `WriteLine`'dan tek farkı ekrana yazıyı yazdıktan sonra imlecin alt satıra inmemesi.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Yorum ekleme

Yorumlar, programa etki etmeyen ancak kendimizin veya kodu inceleyen bir başkasının nerede ne yapıldığını anlamasını sağlayacak yazılardır.Bir diğer ismi sözde koddur. Yorumlar tek satırlık ve çok satırlık olmak üzere ikiye ayrılır.

Tek satırlık yorumlar

C#, iki tane slash'ten (//) sonra gelen aynı satırdaki yazıları yorum sayar. Örnek:

```
using System;
class yorum
{
   static void Main()
   {
      int a=0; //Burada a'yı tanımladık ve sıfır değerini verdik.
      int b=5; //Burada b'yi tanımladık ve beş değerini verdik.
      Console.Write(a+b); //Burada toplamlarını ekrana yazdırdık.
   }
}
```

Çok satırlık yorumlar

C#, /* ve */ arasına yazılan her şeyi yorum sayar. Dolayısıyla da çok satırlık yorumlar yazılabilir. Örnek:

```
using System;
class yorum
{
   static void Main()
   {
      Console.Write("deneme"); /*Burası birinci satır,
                                 burası ikinci satır,
                                 işte burada bitti.*/
   }
}
```

Ayrıca C#'ın bu yorum özelliği programımızda henüz çalışmasını istemediğimiz kodlar varsa yararlı olabilmektedir.

Bu kitabın diğer sayfaları

- C# hakkında temel bilgiler
- İlk programımız
- Değişkenler
- Tür dönüşümü
- Yorum ekleme
- Operatörler
- Akış kontrol mekanizmaları
- Rastgele sayı üretme
- Diziler
- Metotlar
- Sınıflar
- Operatör aşırı yükleme
- İndeksleyiciler
- Yapılar
- Enum sabitleri
- İsim alanları
- System isim alanı
- Temel I/O işlemleri
- Temel string işlemleri
- Kalıtım
- Arayüzler
- Partial (kısmi) tipler
- İstisnai durum yakalama mekanizması
- Temsilciler
- Olaylar
- Önişlemci komutları
- Göstericiler
- Assembly kavramı
- Yansıma
- Nitelikler
- Örnekler
- Şablon tipler
- Koleksiyonlar
- yield
- Veri tabanı işlemleri
- XML işlemleri
- Form tabanlı uygulamalar
- Visual Studio.NET
- Çok kanallı uygulamalar
- Linux'ta C# kullanımı
- Kaynakça
- Giriş sayfası
- **Ayrıca Bakınız:**

  ASP.NET

Operatörler

Programlama dillerinde tek başlarına herhangi bir anlamı olmayan ancak programın işleyişine katkıda bulunan karakter ya da karakter topluluklarına operatör denir. Örneğin `a+b` ifadesinde + işareti bir operatördür. Operatörlerin etki ettikleri sabit ya da değişkenlere ise operand denir. Örneğin `a+b` ifadesinde a ve b birer operanddır. C#'ta operatörlerın bazıları tek, bazıları çift, bazıları da üç operand alır. Peki eğer bir ifadede birden fazla operatör varsa ne olacak? Örneğin 2+3*7 gibi bir ifadede önce hangi işlem yapılacak? Burada önce toplama işlemi yapılırsa sonuç 35 olur, ancak önce çarpma işlemi yapılırsa sonuç 23 olur. İşte C#'ta bu gibi durumları önlemek için operatör önceliği diye kavram bulunmaktadır. Operatör öncelikleri ile ilgili şu kuralları bilmenizde fayda var:

- Önce parantez içleri yapılır. Örneğin (3+5)*9 gibi bir ifadede önce toplama işlemi yapılır.
- İç içe parantezler söz konusuysa önce en içteki parantez yapılır, sıra ile dıştaki parantezlere gidilir. Örneğin (2+(9+2)*5)*2 gibi bir ifadede önce 9 ile 2 toplanıp 11 değeri bulur, sonra 11 ile 5 çarpılıp 55 değeri bulunur, sonra 55 ile 2 toplanıp 57 değeri bulunur, en son da 57 ile 2 çarpılıp 114 değeri bulunur.
- Dikkat ettiyseniz yukarıdaki örnekte bir parantez söz konusu olmamasına rağmen önce toplama yerine çarpmanın yapıldığı bir durumla karşılaştık. Çünkü C#'ta farklı operatörlerin farklı önceliği vardır. Yani ifadede bir parantez yoksa önce öncelik sırası daha önde olan operatörün işlemi yapılır. Aşağıdaki tabloda operatörler öncelik sırasına göre listelenmiştir. Tablo, öncelik sırası en önde olandan en geride olana göre sıralanmıştır. Yani aynı ifadede parantez söz konusu değilse bu tablodaki daha üstte olan operatör daha önce işletilecektir.

Birinci öncelikliler	x++, x--
Tek operand alan operatörler	+, -, !, ~, ++x, --x, (Tür)x
Çarpma ve bölme	*, /, %
Toplama ve çıkarma	+, -
Kaydırma operatörleri	<<, >>
İlişkisel ve tür testi operatörleri	<, >, <=, >=, is, as
Eşitlik operatörü	==, !=
Bitsel Ve (AND)	&
Bitsel Özel Veya (XOR)	^
Bitsel Veya (OR)	|
Mantıksal Ve	&&
Mantıksal Veya	||
Koşul operatörü	?:
Atama ve işlemli atama operatörleri	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

Dikkat ettiyseniz yukarıdaki tablodaki bazı operatörler aynı sırada, yani bunların öncelik sırası eşit. C#'ta eğer operatörler eşit öncelikliyse, soldan sağa işlem yapılır. Yani 2*6/5 işleminde önce 2 ile 6 çarpılır, sonra da sonuç 5'e bölünür. Çoğu durumda bunun sonuca bir değişiklik yapmadığı düşünülse de bazı durumlarda işleme soldan başlanmasının sağdan başlamaya göre değişik sonuçlar doğurduğu durumlara rastlanabilir. Örneğin:

```
using System;
class Operatorler
{
	static void Main()
	{
```

```
            int i=3*5/7;
            Console.Write(i);
      }
}
```

Bu programda ekrana 2 yazılır. Ancak;

```
using System;
class Operatorler
{
      static void Main()
      {
            int i=3*(5/7);
            Console.Write(i);
      }
}
```

Bu programda ekrana 0 (sıfır) yazılır. Çünkü C# sonunda herhangi bir harf olmayan tam sayı sabitleri int sayar ve intin kurallarına göre işlem yapar. int sabitlerle yapılan her işlemden sonra -oluşursa- ondalık kısım atılır. Ancak programı aşağıdaki gibi değiştirirsek sonuç değişmeyecektir.

```
using System;
class Operatorler
{
      static void Main()
      {
            int i=(int)(3f*(5f/7f));
            Console.Write(i);
      }
}
```

Burada sabitleri floatlaştırdık, dolayısıyla da işlemin sonunda float bir sabit oluştu. Sonra tür dönüştürme operatörüyle sonucu inte çevirip i değişkenine atadık ve bu değişkeni de ekrana yazdırdık. Şimdi isterseniz bütün operatörleri tek tek inceleyelim.

Matematiksel operatörler

+, -, / ve * operatörleri hakkında açıklama yapma gereksinmi görmüyorum. Burada henüz görmediğiniz bir operatörü tanıtmak istiyorum. % operatörü mod alma işlemi yapar, yani;

```
using System;
class Operatorler
{
      static void Main()
      {
            int i=5%2;
            Console.Write(i);
      }
}
```

Bu program ekrana 1 yazar. Yani 5 sayısının 2'ye bölümünden kalan i değişkenine atandı. Matematiksel operatörler hakkında şunu eklemek istiyorum: eğer işlem yapılan her iki operarand farklı türdeyse sonuç daha üst kapasiteli türde tutulur. Örnek:

```
using System;
class Operatorler
{
	static void Main()
	{
		float i=5/2f;
		Console.Write(i);
	}
}
```

Burada 5 sayısı da float gibi davranmış ve sonuç ondalıklı çıkmıştır. Ancak,

```
using System;
class Operatorler
{
	static void Main()
	{
		float i=2/5/2f;
		Console.Write(i);
	}
}
```

Burada sonuç 0 çıkar. Çünkü zaten işlem yapılırken sonuç daha floata (2f) gelmeden sıfırlanmıştı.

```
using System;
class Operatorler
{
	static void Main()
	{
		float i=2f/5/2;
		Console.Write(i);
	}
}
```

Burada ise zincirleme olarak her işlemin sonucu floatlaşmakta ve sonuç ondalıklı çıkmaktadır. Daha çarpıcı bir örnek:

```
using System;
class Operatorler
{
	static void Main()
	{
		float i=2f/5/2/2/4/8/4/5/3;
		Console.Write(i);
	}
}
```

Burada da sonuç ondalıklıdır.

Bir artırma ve bir eksiltme operatörleri

C#'ın en çok kullanılan operatörlerindendir. Önüne veya sonuna geldiği değişkeni bir artırır veya bir eksiltirler, sabitlere uygulanamazlar. Hem uygulandığı değişkenin değerini 1 artırır veya eksiltirler hem de tutucu vazifesi görürler. Ön ek ve son ek olmak üzere iki şekilde kullanılabilirler. Örnek;

```
using System;
class Operatorler
{
	static void Main()
	{
		int a=5;
		int i=++a;
		Console.Write(i+" "+a);
	}
}
```

Programda da gördüğünüz gibi ++ operatörü hem a'nın değerini bir artırdı hem de bu değeri tuttu ve değer i'ye atandı. İşte ön ek ve son ek şeklinde kullanımın farkı burada ortaya çıkıyor. Örneği inceleyiniz.

```
using System;
class Operatorler
{
	static void Main()
	{
		int a=5;
		int i=a++;
		Console.Write(i+" "+a);
	}
}
```

Bu programda ++ operatörü ön ek şeklinde değil de son ek şeklinde kullanılmış. Bu durumda 7. satırda önce a'nın değeri i'ye atanır, sonra a'nın değeri 1 artırılır. Halbuki ilk örneğimizde önce a'nın değeri 1 artırılmış sonra bu değer i'ye atanmıştı. İsterseniz şimdi bu ++ ve -- operatörlerini bir örnekte hep beraber görelim;

```
using System;
class Operatorler
{
	static void Main()
	{
		int a=5;
		int i=a++;
		int b=i--;
		int c=10;
		int d=--c;
		Console.Write("{0}\n{1}\n{2}\n{3}",a,i,b,d);
	}
}
```

Bu program ekrana alt alta 6, 4, 5 ve 9 yazacaktır. Dikkat ettiyseniz `Write` metodunu farklı şekilde kullandık. Siz de böyle çok fazla değişkeni daha kolay yazdırabilirsiniz. Ayrıca bu yöntemi `WriteLine` metodunda da kullanabilirsiniz.
NOT: Bir artırma ve bir azaltma operatörleri byte, sbyte, short ve ushort türlerindeki değişkenlerin türünü değiştirmez.

Karşılaştırma operatörleri

Daha önceki derslerimizden birinde 2<5 gibi bir ifadenin aslında bool türünden bir değer olduğunu söylemiştik. İşte bu operatörler bu bool türünden değerleri operandları vasıtasıyla üretmek için kullanılır. İki sayının birbirine göre büyüklüğünü ve küçüklüğünü kontrol ederler. Tabii ki yalnızca sayısal türdeki (char da dâhil) değişken ya da sabitlerle kullanılabilirler. Charla sayısal türler karşılaştırıldığında charın Unicode karşılığı hesaba katılır. Bu operatörler şunlardır: < (küçüktür), > (büyüktür), <= (küçük eşittir), >= (büyük eşittir), == (eşittir), != (eşit değildir),. Örnekler:

```
using System;
class Operatorler
{
	static void Main()
	{
		bool a=4<6;
		bool b=6>5;
		bool c=7<=7;
		bool d=9>=12;
		bool e=10==12;
		bool f=1!=8;
		Console.Write("{0}\n{1}\n{2}\n{3}\n{4}\n{5}",a,b,c,d,e,f);
	}
}
```

Bu program ekrana alt alta True, True, True, False, False ve True yazacaktır.

as operatörü

Tüm değişken türlerinden objecte ve string değerli olmak şartıyla objectten stringe dönüşüm yapar. Örnek:

```
using System;
class Operatorler
{
	static void Main()
	{
		object i="50";
		string s=i as string;
		Console.Write(s);
	}
}
```

veya

```
using System;
class Operatorler
```

```
{
	static void Main()
	{
		int i=50;
		object s=i as object;
		Console.Write(s);
	}
}
```

is operatörü

Verilen değişken, sabit ya da ifadenin türünü kontrol eder. Eğer söz konusu değişken, sabit ya da ifade verilen türle uyumluysa true değilse false değeri üretir. Eğer söz konusu değişken, sabit ya da ifadenin türü her zaman true ya da false üretiliyorsa derleyici uyarı verir, ancak bu uyarı derlemeye engel değildir. Kullanımına bir örnek:

```
using System;
class Operatorler
{
	static void Main()
	{
		int i=50;
		bool b1=i is int;
		bool b2=i is double;
		bool b3=12 is byte;
		bool b4='c' is string;
		bool b5=12f+7 is int;
		Console.Write("{0}\n{1}\n{2}\n{3}\n{4}",b1,b2,b3,b4,b5);
	}
}
```

Bu program alt alta True, False, False, False ve False yazacaktır. Dikkat ettiyseniz 12 sabitini byte saymadı ve 12f+7 sabitini de int saymadı. Çünkü C#, harfsiz tam sayı sabitleri int sayar. 12f+7 sabiti de floatlaşmıştır.

Mantıksal operatörler

Bu operatörler true veya false sabit ya da değişkenleri mantıksal ve, veya, değil işlemine sokarlar. Bunlar && (ve), || (veya) ve ! (değil) operatörleridir. Örnekler:

```
using System;
class Operatorler
{
	static void Main()
	{
		bool b1=35>10&&10==50; //false
		bool b2=35>10&&10!=50; //true
		bool b3=5 is int||12*3==200; //true
		bool b4=5<4||45/5==9; //true
		bool b5=!(5<4); //true
		Console.Write(b1+" "+b2+" "+b3+" "+b4+" "+b5);
	}
```

```
}
```

Eğer ! operatöründen sonra bir değişken ya da sabit yerine bir ifade geliyorsa ifadeyi parantez içine almak zorundayız.

Bitsel operatörler

Bitsel operatörler sayıların kendisi yerine sayıların bitleriyle ilgilenirler. Diğer bir deyişle sayıları ikilik sisteme dönüştürüp öyle işlem yaparlar. Bu operatörler yalnızca tam sayı sabit, değişken ya da ifadelerle kullanılabilirler. Eğer bitsel operatörler bool türünden değişken, sabit ya da ifadelerle kullanılırsa mantıksal operatörlerin gördüğü işin aynısını görürler. Örneğin aşağıdaki iki kullanım da birbiyle aynı sonucu doğuracaktır:

```
bool b1=false&&false;
bool b1=false&false;
```

~ (bitsel değil), & (bitsel ve), | (bitsel veya) ve ^ (bitsel özel veya) operatörleri bitsel operatörlerdir. Örnekler:

```
using System;
class Operatorler
{
      static void Main()
      {
            byte a=5&3; // 00000101&00000011 -->sonuç 00000001 çıkar.
Yani 1 çıkar.
            byte b=5|3; // 00000101|00000011 -->sonuç 00000111 çıkar.
Yani 7 çıkar.
            byte c=5^3; // 00000101^00000011 -->sonuç 00000110 çıkar.
Yani 6 çıkar.
            byte d=5;
            byte e=(byte)~d; // 00000101'in değili -->sonuç 11111010
çıkar. Yani 250 çıkar.
            byte f=(byte)~(a+b); // 00001000'ın değili -->sonuç
11110111 çıkar. Yani 247 çıkar.
            byte g=(byte)~(a+7); // 00001000'ın değili --->sonuç
11110111 çıkar. Yani 247 çıkar.
            Console.Write(a+" "+b+" "+c+" "+e+" "+f+" "+g);
      }
}
```

Bitsel operatörler sayıları ikilik sisteme dönüştürüp karşılıklı basamakları "ve", "veya" veya "özel veya" işlemine sokarlar ve sonucu operandlarıyla birlikte tutarlar. & (ve) işlemi karşılıklı basamaklardan her ikisi de 1'se ilgili basamağında 1 tutar, diğer durumlarda 0 tutar. | (veya) işlemi karşılıklı basamaklarından herhangi birisi 1'se ilgili basamağında 1 tutar, diğer durumlarda 0 tutar. ^ (özel veya) işlemi karşılıklı basamakları farklıysa ilgili basamağında 1 tutar, diğer durumlarda 0 tutar. ~ (değil) operatörü operandının her basamağını tersleştirip tutar. Bu operatörlerin operandları byte, sbyte, short, ushort olması durumunda sonuç intleşir.
NOT: Başında 0x veya 0X olan sayılar 16'lık düzende yazılmış demektir. Örnek:

```
using System;
class Operatorler
{
      static void Main()
```

```
        {
                int a=0xff;
                int b=0Xff;
                Console.Write(a+" "+b);
        }
}
```

Bu program ff sayısının onluk düzendeki karşılığı olan 255 sayısını ekrana iki kere yazacaktır.

Bitsel kaydırma operatörleri

Bu operatörler (<< ve >>) sayıların bitlerini istenildiği kadar kaydırmak için kullanılır. << sayıları sola kaydırırken, >> sağa kaydırır. Örnekler:

```
using System;
class Operatorler
{
      static void Main()
      {
            byte b=100; // 01100100
            byte c=(byte)(b<<1); //b bir kez sola kaydırıldı ve 11001000 oldu. (200 oldu.)
            byte d=50; // 00110010
            byte e=(byte)(d>>2); // d iki kez sağa kaydırıldı ve
00001100 oldu. (12 oldu.)
            Console.Write(c+" "+e);
     }
}
```

<< ve >> operatörlerinin operandları değişken olabileceği gibi sabit ya da ifade de olabilir. Operand(lar) byte, sbyte, short, ushortsa sonuç intleşir.

Atama ve işlemli atama operatörleri

Şimdiye kadar çok kullandığımız = operatörüyle ilgili bilmeniz gereken iki şey var.

- Sol tarafta kesinlikle yalnızca bir tane değişken olmalı. Örneğin aşağıdaki gibi bir kullanım hatalıdır.

```
int a=0, b=0;
a+b=20;
```

- = operatöründe işlemler sağdan sola gerçekleşir ve = operatörü operandlarıyla birlikte atadığı değeri tutar. Örnek:

```
using System;
class Operatorler
{
      static void Main()
      {
            byte b=7, a=1, c;
            c=a=b;
            Console.Write(a+" "+b+" "+c);
      }
}
```

Bu program ekrana yan yana üç tane 7 yazacaktır. c=a=b; satırında önce b a'ya atanır ve a=b ifadesi 7 değerini tutar. Çünkü a=b atamasında 7 sayısı atanmıştır. Sonra a=b ifadesinin tuttuğu 7 sayısı c'ye atanır. Sonunda üç değişkenin değeri de 7 olur. Ancak şu örnek hatalıdır:

```
using System;
class Operatorler
{
	static void Main()
	{
		byte b=7, a=1, c;
		(c=a)=b;
		Console.Write(a+" "+b+" "+c);
	}
}
```

Burada önce a c'ye atanıyor ve c=a ifadesi 1 değeri tutuyor ve ifademiz 1=b ifadesine dönüşüyor. Atama operatörünün prensibine göre böyle bir ifade hatalı olduğu için program derlenmeyecektir. Bir de işlemli atama operatörleri vardır. Aslında bunlar = operatörünün yaptığı işi daha kısa kodla yaparlar:

- `a=a+b;` yerine `a+=b;`
- `a=a-b;` yerine `a-=b;`
- `a=a*b;` yerine `a*=b;`
- `a=a/b;` yerine `a/=b;`
- `a=a%b;` yerine `a%=b;`
- `a=a<<b;` yerine `a<<=b;`
- `a=a>>b;` yerine `a>>=b;`
- `a=a&b;` yerine `a&=b;`
- `a=a^b;` yerine `a^=b;`
- `a=a|b;` yerine `a|=b;`

kullanılabilir.

?: operatörü

?: operatörü C#'ta üç operand alan tek operatördür. Verilen koşula göre verilen değerlerden (object türünden) birini tutar. Kullanımı şöyledir:

```
koşul?doğruysa_değer:yanlışsa_değer
```

Örnek:

```
using System;
class Operatorler
{
	static void Main()
	{
		Console.Write("Vikikitap'ı seviyor musunuz? (e, h): ");
		string durum=Console.ReadLine();
		Console.Write(durum=="e"?"Teşekkürler!!":"Sağlık
olsun...");
	}
```

```
}
```

Tabii ki verilen değerlerin illa ki sabit olma zorunluluğu yoktur, değişken ya da ifade de olabilir.

İşaret değiştirme operatörü (-)

- operatörü bir değişken, sabit ya da ifadenin işaretini değiştirmek için kullanılır. Örnekler:

```
using System;
class Operatorler
{
	static void Main()
	{
		int a=50;
		int b=-a;
		int c=-23;
		int d=-(a+b+c);
		int e=-12;
		Console.Write("{0}\n{1}\n{2}\n{3}\n{4}",a,b,c,d,e);
	}
}
```

Stringleri yan yana ekleme operatörü (+)

+ operatörü stringleri veya charları yan yana ekleyip operandlarıyla birlikte string olarak tutar. Ancak ifadedeki bütün bileşenlerin string olarak birleşebilmesi için ilk stringten önce charların yan yana gelmemesi gerekir. Yalnızca charlardan oluşan bir ifade stringleşmez.
NOT: + ve - operatörleri aynı zamanda sayısal (ve char) sabit, değişken ve ifadeleri toplamak ve çıkarmak için de kullanılır. Bunu daha önce görmüştük.

typeof operatörü

Herhangi bir değişken türünün CTS karşılığını Type türünden tutar. Örnek:

```
using System;
class Operatorler
{
	static void Main()
	{
		Type a=typeof(int);
		Console.Write(a);
	}
}
```

Bu program ekrana `System.Int32` yazar.

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Akış kontrol mekanizmaları

Program yazarken çoğu zaman herşey ak ve kara gibi net olmaz, çoğu zaman çeşitli koşullara göre farklı komutlar çalıştırmamız gerekir. Benzer şekilde çoğu komutun da yalnızca bir kez çalıştırılması bizim için yeterli gelmez, belli koşulları sağladığı sürece sürekli çalıştırılmasını istediğimiz komutlar olabilir. İşte bu gibi durumlar için C#'ta akış kontrol mekanizmaları vardır. Aslında en basitinden en karmaşığına kadar bütün programlama dillerinde bu mekanizmalar mevcuttur ve programlama dillerinin en önemli ögelerinden birisidir.

if else

`if else` deyimi sayesinde belli bir koşul sağlandığında söz konusu komutlar çalıştırılır, o belli koşullar sağlanmadığında çalıştırılmaz ya da başka komutlar çalıştırılır. Kullanılışı şu şekildedir:

```
if(koşul)
	komut1;
else
	komut2;
```

veya

```
if(koşul)
{
	komutlar1
}
else
{
	komutlar2
}
```

Yukarıdaki örneklerde eğer koşul sağlanırsa 1. komutlar, sağlanmazsa 2. komutlar çalıştırılır. if veya else'in altında birden fazla komut varsa bu komutları parantez içine almak gerekir. if veya else'in altında tek komut varsa bu komutları parantez içine almak gerekmez. Örnek bir program:

```
using System;
class AkisKontrolMekanizmalari
{
	static void Main()
	{
```

```
            int a=5, b=7;
            if(a<b)
                  Console.Write("a b'den küçük");
            else
                  Console.Write("a b'den küçük değil");
      }
}
```

Başka bir örnek program:

```
using System;
class AkisKontrolMekanizmalari
{
      static void Main()
      {
            int a=5, b=7;
            if(a<b)
            {
                  Console.WriteLine("a b'den küçük");
                  Console.Write(a);
            }
            else
            {
                  Console.WriteLine("a b'den küçük değil");
                  Console.Write(b);
            }
      }
}
```

if else yapılarında else kısmının bulunması zorunlu değildir. Bu durumda sadece koşul sağlandığında bir şeyler yapılacak, koşul sağlanmadığında bir şeyler yapılmayacaktır. Örnek:

```
using System;
class AkisKontrolMekanizmalari
{
      static void Main()
      {
            int a=5, b=7;
            if(a<b)
                  Console.WriteLine("a b'den küçük");
      }
}
```

if else bloklarının aşağıdaki gibi kullanımı da mümkündür:

```
using System;
class AkisKontrolMekanizmalari
{
      static void Main()
      {
```

```
            Console.Write("Cinsiyetinizi girin (e, k): ");
            char cins=Convert.ToChar(Console.ReadLine());
            if(cins=='e')
                  Console.Write("Erkeksiniz");
            else if(cins=='k')
                  Console.Write("Kızsınız");
            else
                  Console.Write("Lütfen cinsiyetinizi doğru giriniz!");
      }
}
```

Bu program kullanıcıdan cinsiyetini girmesi istemekte, eğer kullanıcının girdiği harf `e` ise ekrana `Erkeksiniz` yazmakta, eğer girdiği harf `e` değilse ise bu sefer kullanıcının girdiği harfi `k` mı değil mi diye incelemekte, eğer k girmişse ekrana `Kızsınız` yazmakta, bunların dışında bir harf girdiğinde de ekrana `Lütfen cinsiyetinizi doğru giriniz!` yazmaktadır. Bu şekilde bu bloklar daha da uzatılabilir. Yani `else if` satırının bir tane olması zorunlu değildir. Ancak tabii ki `else` satırının yalnızca bir tane olması gerekir. C# iç içe if else kullanılmasına izin verir:

```
if(koşul1)
{
      if(koşul2)
            komut1;
      else
      {
            komut2;
            komut3;
      }
}
else
      komut4;
```

switch

switch deyimi bazı if else deyimlerinin yaptığı işi daha az kodla yapar. Genellikle bazı karmaşık if else bloklarını kurmaktansa switch'i kullanmak programın anlaşılırlığını artırır. Ancak tabii ki basit if else bloklarında bu komutun kullanılması gereksizdir. Kuruluşu:

```
switch(ifade)
{
      case sabit1:
            komut1;
            break;
      case sabit2:
            komut2;
            break;
      default:
            komut3;
           break;
}
```

Bu switch deyimiyle ilgili bilmeniz gerekenler:

- İfadenin ürettiği değer hangi case sabitinde varsa o "case"deki komutlar işletilir. Eğer ifadenin ürettiği değer hiçbir case sabitinde yoksa default casedeki komutlar işletilir.
- Aynı birden fazla case sabiti olamaz. Örneğin:

```
using System;
class AkisKontrolMekanizmalari
{
	static void Main()
	{
		int a=4;
		switch(a)
		{
			case 4:
				Console.Write("deneme1");
				break;
			case 4:
				Console.Write("deneme2");
				break;
			case 5:
				Console.Write("deneme3");
				break;
			default:
				Console.Write("deneme4");
				break;
		}
	}
}
```

Bu program hatalıdır.

- C#'ta herhangi bir case'e ait komutların `break;` satırı ile sonlandırılması gerekmektedir. Eğer `break;` satırı ile sonlandırılmazsa programımız hata verir. Örneğin aşağıdaki program hata vermez:

```
using System;
class AkisKontrolMekanizmalari
{
static void Main()
{
	int a=4;
	switch(a)
	{
		case 4:
			string b=Console.ReadLine();
			Console.Write(b);
			break;
		default:
			Console.Write("deneme4");
			break;
```

```
        }
 }
```

Ancak şu program hata verir:

```
using System;
class AkisKontrolMekanizmalari
{
        static void Main()
        {
                int a=4;
                switch(a)
                {
                        case 4:
                                string b=Console.ReadLine();
                                Console.Write(b);
                        default:
                                Console.Write("deneme4");
                                break;
                }
        }
}
```

- Eğer programımızın bir case'deyken farklı bir case'e gitmesini istiyorsak `goto` anahtar sözcüğünü kullanırız. Örnek:

```
using System;
class AkisKontrolMekanizmalari
{
        static void Main()
        {
                int a=5;
                switch(a)
                {
                        case 4:
                                string b=Console.ReadLine();
                                Console.Write(b);
                                break;
                        case 5:
                                Console.Write("Şimdi case 4'e gideceksiniz.");
                                goto case 4;
                        default:
                                Console.Write("deneme4");
                                break;
                    }
         }
}
```

- goto satırı kullanılmışsa break; satırının kullanılmasına gerek yoktur.
- Eğer farklı case sabitlerinin aynı komutları çalıştırmasını istiyorsak şöyle bir program yazılabilir:

```
using System;
class AkisKontrolMekanizmalari
{
        static void Main()
        {
                int a=5;
              switch(a)
              {
                    case 4:
                    case 5:
                          string b=Console.ReadLine();
                          Console.Write(b);
                          break;
                    default:
                          Console.Write("deneme4");
                          break;
              }
       }
}
```

Bu programda a değişkeni 4 olsa da 5 olsa da aynı komutlar çalıştırılacaktır.

- case anahtar sözcüğünün yanındaki ifade mutlaka ya sabit ya da sabitlerden oluşan bir ifade olmalıdır.
- default durumunu istediğiniz yere yazabilirsiniz, aynı şekilde istediğiniz case'leri de istediğiniz yere yazabilirsiniz. Yani case'lerin sırası önemli değildir.
- Bir switch bloğunda default durumu bulunmak zorunda değildir.
- switch'in parantez içindeki ifadesi bir değişken olabileceği gibi, bir sabit ya da ifade de olabilir.

for döngüsü

Eğer programda -belli koşulları sağladığı sürece- birden fazla çalıştırılmasını istediğimiz kodlar varsa döngüler kullanılır. C#'ta en çok kullanılan döngü "for"dur. Kullanımı:

```
for(ifade1;kosul;ifade2)
        komut;
```

veya

```
for(ifade1;kosul;ifade2)
{
       komut1;
       komut2;
         .
         .
         .
}
```

for döngüsünün çalışma prensibi

```
for(ifade1;kosul;ifade2)
{
       komut1;
       komut2;
         .
         .
         .
}
```

1) ifade1 çalıştırılır.

2) Koşula bakılır. Eğer koşul sağlanıyorsa;

2.1) küme parantezleri içindeki ya da -küme parantezleri yoksa- kendinden sonra gelen ilk satırdaki komut çalıştırılır.

2.2) ifade2 çalıştırılır.

2.3) 2. adıma dönülür.

3) Eğer koşul sağlanmıyorsa küme parantezleri dışına ya da -küme parantezleri yoksa- kendinden sonra gelen ilk satırdaki komuttan hemen sonraki satıra çıkılır. Dolayısıyla döngüden çıkılmış olur.

for döngüsüyle ilgili örnekler

```
using System;
 class AkisKontrolMekanizmalari
 {
       static void Main()
       {
             int toplam=0;
             for(int i=1;i<=3;i++)
                   toplam+=i;
             Console.Write("toplam={0}",toplam);
       }
 }
```

Bu program 1'den 3'e kadar olan tam sayıları (1 ve 3 dâhil) toplayıp toplamı ekrana yazacaktır.

```
 using System;
 class AkisKontrolMekanizmalari
 {
       static void Main()
       {
             float s;
             int si,f;
             bool a=true;
             for(;a;)
             {
                   Console.Write("Lütfen faktoriyelinin alınmasını
istediğiniz sayıyı girin: ");
                   s=Convert.ToSingle(Console.ReadLine());
```

```
                    si=(int)s;
                    if(si!=s||s<1)
                    {
                          Console.WriteLine("Lütfen pozitif tam sayı
girin.");
                          a=true;
                    }
                    else
                    {
                                a=false;
                                for(f=1;si>1;si--)
                              f*=si;
                          Console.Write("Faktöriyeli={0}",f);
                         }
             }
       }
 }
```

Bu program girdiğimiz herhangi bir sayının faktöriyelini bulup ekrana yazar. Eğer girdiğimizi sayı tam sayı değilse veya 1'den küçükse "Lütfen pozitif tam sayı girin." diyerek tekrar veri girişi ister.
UYARI: C# nokta (.) işaretini sayıların ondalık kısımlarını ayırmak için kullanırken, DOS aynı amaç için virgül (,) işaretini kullanır.

for döngüsüyle ilgili kurallar

- for döngüsünün parantezleri içindeki "for(int i=0;i<5;i++)" iki ifade ve bir koşulun istenirse bir tanesi, istenirse bazıları, istenirse de tamamı boş bırakılabilir; ancak noktalı virgüller mutlaka yazılmalıdır.
- Tahmin edebileceğiniz gibi for döngüsünün içinde veya "for(int i=0;i<5;i++)" kısmında tanımlanan herhangi bir değişken döngünün dışında kullanılamaz. Bir değişkeni döngünün dışında kullanabilmemiz için o değişkenin döngüden önce tanımlanıp ilk değer verilmesi ve değişkeni kullanacağımız yerde de faaliyet alanının devam etmesi gerekmektedir. Bu bütün döngüler ve koşul yapıları için geçerlidir. Örneğin:

```
using System;
class AkisKontrolMekanizmalari
{
       static void Main()
       {
             int a=0;
              for(int i=0;i<1;i++)
                  a=5;
              Console.Write(a);
       }
}
```

Bu program ekrana `5` yazacaktır. Ancak;

```
using System;
class AkisKontrolMekanizmalari
{
       static void Main()
```

```
        {
              int a;
               for(int i=0;i<1;i++)
                   a=5;
               Console.Write(a);
        }
 }
```

Bu program çalışmaz, çünkü a değişkeni döngüden önce tanımlanmasına rağmen ilk değer verilmiyor.
HATIRLATMA: Bir değişkenin faaliyet alanı tanımlandığı en iç bloğun içidir.
NOT: `Console.Write("{0,3}",i);` gibi bir ifadede i değişkeni 3 birimlik bir genişlikte sağa yaslı olarak yazılır. `WriteLine` ile de kullanılabilir. i; değişken, sabit ya da ifade olabilirken 3 yalnızca sabit olmalıdır. i bir ifade olsa bile parantez içine almaya gerek yoktur.

while döngüsü

Komut ya da komutların bir koşul sağlandığı sürece çalıştırılmasını sağlar. Kuruluşu:

```
 while(koşul)
       komut;
veya
 while(koşul)
 {
       komut1;
       komut2;
       .
       .
       .
 }
```

while döngüsünün çalışma prensibi

```
 while(koşul)
 {
       komut1;
       komut2;
       .
       .
       .
 }
```

1) Koşula bakılır. Eğer koşul sağlanıyorsa;

1.1) küme parantezleri içindeki ya da -küme parantezleri yoksa- kendinden sonra gelen ilk satırdaki komut çalıştırılır.

1.2) 1. adıma dönülür.

2) Eğer koşul sağlanmıyorsa küme parantezleri dışına ya da -küme parantezleri yoksa- kendinden sonra gelen ilk satırdaki komuttan hemen sonraki satıra çıkılır. Dolayısıyla döngüden çıkılmış olur.

- Aslında while döngüsü for döngüsünün yalnızca koşuldan oluşan hâlidir. Yani `for(;i<0;)` ile `while(i<0)` aynı döngüyü başlatır.

do while döngüsü

Şimdiye kadar gördüğümüz döngülerde önce koşula bakılıyor, eğer koşul sağlanırsa döngü içindeki komutlar çalıştırılıyordu. Ancak bazen döngüdeki komutların koşul sağlanmasa da en az bir kez çalıştırılmasını isteyebiliriz. Bu gibi durumlar için C#'ta `do while` döngüsü vardır.

Kullanımı

```
do
     komut;
while(koşul)
veya
do
{
     komut1;
     komut2;
     .
     .
     .
}while(koşul)
```

do while döngüsünün çalışma prensibi

```
do
{
     komut1;
     komut2;
     .
     .
     .
}while(koşul)
```

1) Döngüdeki komutlar bir kez çalıştırılır.
2) Koşula bakılır.

2.1) Eğer koşul sağlanıyorsa 1. adıma dönülür.

2.2) Eğer koşul sağlanmıyorsa döngüden çıkılır.

Döngülerde kullanılan anahtar sözcükler

break

Hatırlarsanız break komutunu switch'teki case'lerden çıkmak için kullanmıştık. Benzer şekilde break komutu bütün döngülerden çıkmak için kullanılabilir. Örnek:

```
using System;
class AkisKontrolMekanizmalari
{
     static void Main()
     {
          for(char a;;)
```

```
            {
                  a=Convert.ToChar(Console.ReadLine());
                  if(a=='q')
                        break;
            }
      }
}
```

Bu program, kullanıcı "q" harfini girene kadar kapanmamaktadır.

continue

break sözcüğüne benzer. Ancak break sözcüğünden farklı olarak program continue'u gördüğünde döngüden çıkmaz, sadece döngünün o anki iterasyonu sonlanır. Örnek:

```
using System;
class AkisKontrolMekanizmalari
{
      static void Main()
      {
            for(int a=0;a<51;a++)
            {
                  if(a%2==1)
                        continue;
                  Console.WriteLine(a);
            }
      }
}
```

Bu program 0'dan 50'ye kadar (0 ve 50 dâhil) olan çift sayıları ekrana alt alta yazmaktadır.

goto

Nesneye yönelik programlamada pek hoş görülmese de kullanabileceğiniz başka bir komut "`goto`"dur. Aslında eskiden BASIC gibi dillerde her satırın bir numarası vardı ve bu sözcük satırlar arasında dolaşmayı sağlıyordu. Ancak böyle bir yöntem nesne yönelimli programlamaya terstir. O yüzden çok fazla kullanmamanız tavsiye edilir. Örnek kullanım:

```
using System;
class AkisKontrolMekanizmalari
{
      static void Main()
      {
            birinci:
                  Console.WriteLine("Birinci bölüm");
                  goto ucuncu;
            ikinci:
                  Console.WriteLine("İkinci bölüm");
            ucuncu:
                  Console.WriteLine("Üçüncü bölüm");
      }
```

```
}
```

Bu programda ikinci bölüm hiçbir zaman çalıştırılmayacaktır.

Döngülerle ilgili karışık örnekler

- 1'den 1000'e (sınırlar dâhil) kadar olan sayılar içerisinde 5'e tam bölünen, ancak 7'ye tam bölünemeyen sayıları alt alta listeleyen, bu sayıların kaç tane olduğunu ve toplamını yazan bir program yazınız.

```
using System;
class AkisKontrolMekanizmalari
{
      static void Main()
      {
            int toplam=0, sayi=0, i=5;
            for(;i<1001;i+=5)
            {
                  if(i%35==0)
                        continue;
                  sayi++;
                  toplam+=i;
                  Console.WriteLine(i);
            }
            Console.WriteLine("Sayısı: "+sayi);
            Console.WriteLine("Toplam: "+toplam);
      }
}
```

- Girilen pozitif herhangi bir tam sayıyı ikilik düzene çeviren programı yazınız.

```
using System;
class AkisKontrolMekanizmalari
{
      static void Main()
      {
            string mod="";
            Console.Write("Lütfen ikilik sisteme dönüştürülmesini
istediğiniz sayınızı girin: ");
            float a=Convert.ToSingle(Console.ReadLine());
            int b=(int)a;
            if(a<=0||a!=b)
                  mod="Bir pozitif tam sayı girmediğiniz için sayınız
ikilik sisteme dönüştürülmedi!";
            else
                  for(;b>0;b/=2)
                        mod=b%2+mod;
          Console.Write(mod);
      }
}
```

- Konsol ekranına girilen 0 ile 100 (sınırlar dâhil) arasındaki 10 notun en büyüğünü, en küçüğünü ve ortalamasını yazan programı yazınız.

```
using System;
class AkisKontrolMekanizmalari
{
      static void Main()
      {
            int bs=0, toplam=0, ks=0;
            for(int a=0, b;a<10;a++)
            {
                  Console.Write(a+1+". notu giriniz: ");
                  b=Convert.ToInt32(Console.ReadLine());
                  if(b>100||b<0)
                  {
                        Console.Write("Yanlış not girdiniz. Lütfen tekrar
");
                        a--;
                        continue;
                  }
                  if(a==0)
                  {
                        bs=b;
                        ks=b;
                  }
                  else
                  {
                        if(b>bs)
                              bs=b;
                        if(b<ks)
                              ks=b;
                  }
                  toplam+=b;
            }
            Console.Write("En büyük: {0}\nEn küçük: {1}\nOrtalama:
"+toplam/10,bs,ks);
      }
}
```

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Rastgele sayı üretme

C#'ta program yazarken (özellikle oyun programlarında) rastgele değerlere ihtiyaç duyabiliriz.

- Rastgele sayı üretebilmemiz için öncelikle `Random` sınıfı türünden bir nesne yaratmalıyız:

```
Random rnd=new Random();
```

Buradaki yarattığımız nesnenin adı `rnd`. Şimdi bu nesne üzerinden Random sınıfının metotlarına erişebileceğiz.

```
int RastgeleSayi1=rnd.Next(10,20);
int RastgeleSayi2=rnd.Next(50);
int RastgeleSayi3=rnd.Next();
double RastgeleSayi4=rnd.NextDouble();
```

Birinci örnekte: 10 ile 20 arasında int türden rastgele bir sayı üretilir, 10 dâhil ancak 20 dâhil değildir.
İkinci örnekte: 0 ile 50 arasında int türden rastgele bir sayı üretilir, 0 dâhil ancak 50 dâhil değildir.
Üçüncü örnekte: int türden pozitif herhangi bir sayı üretilir.
Dördüncü örnekte: double türden 0.0 ile 1 arasında rastgele bir sayı üretilir.

Random sınıfı System isim alanı içinde bulunduğu için programımızın başında `using System;` satırının bulunması rastgele sayı üretme metotlarını kullanabilmemiz için yeterlidir.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Diziler

Şimdiye kadar birçok değişken tanımladık ve programlarımızın içinde kullandık. Bir program içinde tanımladığımız değişken sayısı şimdiye kadar bir elin parmaklarını geçmedi. Ancak her zaman bu böyle olmayabilir. Bazı programlarda 200-300 değişkene ihtiyaç duyabiliriz. Bunların hepsinin teker teker tanımlanması oldukça zahmetlidir. İşte bu yüzden programlama dillerinde dizi diye bir kavram vardır. Aslında bir dizi, birbiriyle ilişkili değişkenlerin oluşturduğu bir gruptan başka bir şey değildir. Diyelim ki `sayi1` ve `sayi2` adlı iki değişken tanımladık. Bunların adları birbirine benzemesine rağmen birbiriyle hiçbir ilişkisi yoktur. Yani bir döngü içinde `sayiX` yazarak bu değişkenlere erişemeyiz. Halbuki dizilerde bu mümkündür.

Dizilerin tanımlanması ve elemanlarının kullanılması

```
int[] dizi=new int[25];
```

veya

```
int[] dizi;
dizi=new int[25];
```

Yukarıdaki iki kodda da int türünden 25 elemanlı `dizi` adında bir dizi tanımlandı ve dizinin her bir elemanına int türünün varsayılan değeri atandı. Varsayılan değerler, sayısal türler için 0, object türü için NULL (yokluk), string türü için "", char için ' ' (boşluk) ve bool için false değerleridir.

- Bütün dizilerin birinci elemanı 0. indeksidir. `dizi` dizisinin birinci elemanına `dizi[0]`, 10. elemanına `dizi[9]` yazarak erişebilir ve bu dizi elemanlarını bir değişkenmiş gibi kullanabiliriz. Örnek:

```
using System;
class Diziler
{
   static void Main()
   {
      int[] dizi=new int[20];
      dizi[5]=30;
      Console.Write(dizi[5]);
   }
}
```

Bu program ekrana 30 yazacaktır.

- C ve C++ programlama dillerinde olduğu gibi dizilerin elemanlarına aşağıdaki gibi de değer atayabiliriz:

```
string[] dizi1={"Bir","İki","Üç"};
int[] dizi2={2,-4,6};
float[] dizi3={2f,1.2f,7f};
```

Diziler yukarıdaki şekilde tanımlandığında söz konusu dizilerin eleman sayısı yazılan eleman sayısı olur. Örneğin yukarıdaki örneklerde üç dizinin de eleman sayısı üçtür ve dördüncü elemana ulaşmak istersek programımız çalışma zamanında hata verir. Bu şekilde dizi elemanlarına değişken ve ifadeler de atanabilir.

- `int[] dizi=new int[20];` satırında 20'nin illaki sabit olmasına gerek yoktur. Değişken ya da ifade de olabilir. Örnek:

```
using System;
class Diziler
```

```
{
   static void Main()
   {
      int a=Convert.ToInt32(Console.ReadLine());
      int[] dizi=new int[a+5];
      dizi[5]=30;
      Console.Write(dizi[5]);
   }
}
```

- Eleman sayısı belirlenen bir dizinin eleman sayısı daha sonra değiştirilemez.
- Birden fazla dizi aşağıdaki gibi tanımlanabilir:

```
int[] dizi1=new int[10], dizi2=new int[20];
```

veya

```
int[] dizi1, dizi2;
```

İkincisinde tanımlanan dizinin elemanlarına henüz erişilemez.

foreach

foreach yalnızca dizilere uygulanabilen bir döngü yapısıdır. Kullanımı şu şekildedir:

```
int[] dizi={3,2,6,7};
foreach(int eleman in dizi)
   Console.WriteLine(eleman);
```

Burada `dizi` dizisinin bütün elemanları teker teker ekrana yazdırılıyor.

- foreach döngüsüyle dizi elemanlarının değerini değiştiremeyiz, sadece ekrana yazdırmak gibi "read-only" işler yapabiliriz.

Çok boyutlu diziler

Çok boyutlu diziler kısaca her bir elemanı bir dizi şeklinde olan dizilerdir, matris dizileri (düzenli diziler) ve düzensiz diziler olmak üzere ikiye ayrılır.

Matris diziler

Her bir dizi elemanının eşit sayıda dizi içerdiği dizilerdir.

İki boyutlu matris diziler

3X2 boyutunda iki boyutlu bir matris dizi aşağıdaki gibi tanımlanabilir:

```
int[,] dizi=new int[3,2];
```

veya

```
int[,] dizi={{1,2},{3,4},{5,6}};
```

İkinci dizinin elemanları indekslerine göre aşağıdaki gibidir.
dizi[0,0] → 1
dizi[0,1] → 2

dizi[1,0] → 3
dizi[1,1] → 4
dizi[2,0] → 5
dizi[2,1] → 6
Bu diziyi matris olarak aşağıdaki gibi gösterebiliriz:

dizi[0,0]	dizi[0,1]
dizi[1,0]	dizi[1,1]
dizi[2,0]	dizi[2,1]

→

1	2
3	4
5	6

İkiden fazla boyutlu matris diziler

Üç boyutlu bir dizi:

```
int[,,] dizi=new int[3,2,2];
```

veya

```
int[,,] dizi={{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}};
```

Bu dizinin indekslerine göre elemanlarıysa;
dizi[0,0,0] → 1
dizi[0,0,1] → 2
dizi[0,1,0] → 3
dizi[0,1,1] → 4
dizi[1,0,0] → 5
dizi[1,0,1] → 6
dizi[1,1,0] → 7
dizi[1,1,1] → 8
dizi[2,0,0] → 9
dizi[2,0,1] → 10
dizi[2,1,0] → 11
dizi[2,1,1] → 12

- foreach döngüsüyle iç içe döngü kurmaya gerek kalmadan her çeşit boyutlu matris dizinin elemanlarına ulaşılabilir.
- Eğer dizilerin elemanlarını değiştirmemiz gerekiyorsa iç içe for döngüsü kurmamız gerekir. Örnek:

```
int[,,] dizi={{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}};
for(int i=0;i<3;i++)
   for(int j=0;j<2;j++)
      for(int k=0;k<2;k++)
         dizi[i,j,k]=20;
```

Bu programda dizinin bütün elemanlarının değerini 20 ile değiştirdik.

Düzensiz diziler

Her bir dizi elemanının farklı sayıda eleman içerebileceği çok boyutlu dizilerdir.

```
int[][] dizi=new int[3][];
dizi[0]=new int[3];
dizi[1]=new int[4];
dizi[2]=new int[2];
```

Birinci satırda 3 satırı olan ancak sütun sayısı belli olmayan iki boyutlu bir dizi tanımlanıyor. İkinci, üçüncü ve dördüncü satırda da bu iki boyutlu dizinin her bir satırının kaç sütun içerdiği ayrı ayrı belirtiliyor.

dizi[0][0]	dizi[0][1]	dizi[0][2]	
dizi[1][0]	dizi[1][1]	dizi[1][2]	dizi[1][3]
dizi[2][0]	dizi[2][1]		

- Düzensiz dizilerin elemanlarına, örneğin 0,0 indeksine `dizi[0][0]` yazarak erişebiliriz.
- Düzensiz dizilerde foreach döngüsü sadece dizi adını yazarak çalışmaz. Ana düzensiz dizinin her bir elemanı için farklı bir foreach döngüsü başlatılmalıdır.
- Şimdiye kadar öğrendiğimiz şekilde düzensiz dizilerin elemanlarını iç içe for döngüsüyle değiştiremeyiz. Çünkü her satır farklı sayıda sütun içerebileceği için satırların sütun sayısı dinamik olarak elde edilmelidir. Bunun için C#'ın System isim alanındaki Array sınıfına ait metotları vardır ve her diziyle kullanılabilirler.

Dizilerle kullanılabilen metotlar

GetLength()

`x.GetLength(y)` şeklinde kullanılır. Herhangi bir dizinin eleman sayısını int olarak tutar. x diziyi, y o dizinin hangi boyutunun eleman sayısının hesaplanacağını belirtir. Örnekler:

```
int[] dizi={1,4,7,9};
Console.Write(dizi.GetLength(0));
```

Bu program ekrana 4 yazar.

```
int[,] dizi={{2,4,2},{7,10,4},{7,12,6},{2,1,12}};
byte a=dizi.GetLength(1);
Console.WriteLine(a);
```

Bu program ekrana 3 yazar.

```
int[][] dizi=new int[3][];
dizi[0]=new int[]{1,2,3};
dizi[1]=new int[]{4,5,6,7};
dizi[2]=new int[]{8,9};
for(int i=0;i<dizi.GetLength(0);i++)
  for(int j=0;j<dizi[i].GetLength(0);j++)
     Console.WriteLine("dizi[{0}][{1}]={2}",i,j,dizi[i][j]);
```

Bu program dizinin bütün elemanlarını teker teker ekrana yazar.

CreateInstance metodu ile dizi tanımlama

Şimdiye kadar öğrendiğimiz dizi tanımlama yöntemlerinin yanında başka dizi tanımlama yöntemleri de vardır.

```
Array dizi=Array.CreateInstance(typeof(int),5);
```

Burada int türünden 5 elemanlı `dizi` adında bir dizi tanımlandı ve dizinin her bir elemanına int türünün varsayılan değeri atandı.

```
Array dizi=Array.CreateInstance(typeof(int),3,2,5);
```

Burada 3X2X5 boyutunda int türünden 3 boyutlu bir dizi oluşturduk.

```
int[] dizi1=new int[5]{2,3,6,8,7};
Array dizi2=Array.CreateInstance(typeof(int),dizi1);
```

Burada 2X3X6X8X7 boyutunda beş boyutlu bir dizi oluşturduk.

- CreateInstance yöntemiyle oluşturulan dizilere `DiziAdi[0,4]` gibi bir yöntemle erişilemez. Şimdi bir örnek yapalım:

```
using System;
class Diziler
{
   static void Main()
   {
      Array dizi=Array.CreateInstance(typeof(int),5,4,3);
      for(int i=0;i<=dizi.GetUpperBound(0);i++)
         for(int j=0;j<=dizi.GetUpperBound(1);j++)
            for(int k=0;k<=dizi.GetUpperBound(2);k++)
               dizi.SetValue(i+j+k,i,j,k);
      for(int i=0;i<=dizi.GetUpperBound(0);i++)
         for(int j=0;j<=dizi.GetUpperBound(1);j++)
            for(int k=0;k<=dizi.GetUpperBound(2);k++)
               Console.WriteLine(dizi.GetValue(i,j,k));
   }
}
```

Daha önce CreateInstance yöntemiyle oluşturulan dizilere `DiziAdi[0,4]` gibi bir yöntemle erişilemeyeceğini söylemiştik. İşte bunun için çeşitli metotlar vardır.
GetUpperBound: Bir dizinin son indeks numarasını verir.
SetValue: Bir dizinin belirli bir indeksini belirli bir değerle değiştirir.
GetValue: Bir dizinin belirli bir indeksini tutar.

- Bu metotların kullanımları yukarıdaki örnek programda verilmiştir.
- Bu metotlar normal şekilde oluşturulan dizilerle de kullanılabilir.

Dizileri kopyalamak

```
int[] dizi1={1,2,3,4};
 int[] dizi2=new int[10];
 dizi1.CopyTo(dizi2,3);
```

Burada dizi1'in tüm elemanları dizi2'ye 3. indeksten itibaren kopyalanıyor.

```
 int[] dizi1={1,2,3,4};
 int[] dizi2=new int[10];
 Array.Copy(dizi1,dizi2,3);
```

Burada 3 tane eleman dizi1'den dizi2'ye kopyalanır. Kopyalama işlemi 0. indeksten başlar.

```
 int[] dizi1={1,2,3,4,5,6,7};
 int[] dizi2=new int[10];
 Array.Copy(dizi1,2,dizi2,7,3);
```

Burada dizi1'in 2. indeksinden itibaren 3 eleman, dizi2'ye 7. indeksten itibaren kopyalanıyor.

Dizileri sıralama

Örnek:

```
using System;
class Diziler
{
   static void Main()
   {
      Array metinsel=Array.CreateInstance(typeof(string),8);
      metinsel.SetValue("Bekir",0);
      metinsel.SetValue("Mehmet",1);
      metinsel.SetValue("Tahir",2);
      metinsel.SetValue("Yusuf",3);
      metinsel.SetValue("Yunus",4);
      metinsel.SetValue("Gökçen",5);
      metinsel.SetValue("Şüheda",6);
      metinsel.SetValue("Arzu",7);
      Console.WriteLine("Sırasız dizi:");
      foreach(string isim in metinsel)
         Console.Write(isim+" ");
      Console.WriteLine("\n\nSıralı dizi:");
      Array.Sort(metinsel);
      foreach(string isim in metinsel)
         Console.Write(isim+" ");
   }
}
```

Başka bir örnek:

```
using System;
class Diziler
{
```

```
    static void Main()
    {
       Array sayisal=Array.CreateInstance(typeof(int),8);
       sayisal.SetValue(200,0);
       sayisal.SetValue(10,1);
       sayisal.SetValue(6,2);
       sayisal.SetValue(3,3);
       sayisal.SetValue(1,4);
       sayisal.SetValue(0,5);
       sayisal.SetValue(-5,6);
       sayisal.SetValue(12,7);
       Console.WriteLine("Sırasız dizi:");
       foreach(int sayi in sayisal)
          Console.Write(sayi+" ");
       Console.WriteLine("\n\nSıralı dizi:");
       Array.Sort(sayisal);
       foreach(int sayi in sayisal)
          Console.Write(sayi+" ");
    }
}
```

Dizilerde arama

Örnek:

```
using System;
class Diziler
{
   static void Main()
   {
      string[] dizi={"ayşe","osman","ömer","yakup","meltem"};
      Array.Sort(dizi);
      Console.Write(Array.BinarySearch(dizi,"osman"));
   }
}
```

BinarySearch metodu, bir nesneyi bir dizi içinde arar, eğer bulursa bulduğu nesnenin indeksini tutar, bulamazsa negatif bir sayı tutar. BinarySearch'ü kullanabilmek için diziyi daha önce Sort ile sıralamalıyız. Başka bir örnek:

```
using System;
class Diziler
{
   static void Main()
   {
      string[]
dizi={"ayşe","osman","ömer","yakup","meltem","rabia","mahmut","zafer","yılmaz","çağlayan"};
      Array.Sort(dizi);
      Console.Write(Array.BinarySearch(dizi,3,4,"yakup"));
   }
}
```

BinarySearch burada 3. indeksten itibaren 4 eleman içinde "yakup"u arar. Bulursa indeksini tutar. Bulamazsa negatif bir sayı tutar.
UYARI: Yalnızca tek boyutlu diziler Sort ile sıralanabilir, dolayısıyla da çok boyutlu dizilerde hem Sort ile sıralama hem de BinarySearch ile arama yapmak imkansızdır.

Diğer metotlar

```
Array.Clear(dizi,1,3);
```

Bu kod `dizi` dizisinin 1. indeksinden itibaren 3 indeksini sıfırlar (varsayılan değere döndürür).

```
 Array.Reverse(dizi);
```

Bu kod `dizi` dizisinin tamamını ters çevirir.

```
 Array.Reverse(dizi,1,3);
```

Bu kod `dizi` dizisinin 1. indeksten itibaren 3 elemanını ters çevirir.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Metotlar

Şu ana kadar yaptığımız örneklerde hep önceden hazırlanmış `ReadLine()`, `WriteLine()` vb. gibi metotları kullandık. Artık kendi metotlarımızı yapmanın zamanı geldi. Bilmem farkında mısınız, ama aslında bütün örneklerimizde birer metot yaratmıştık. O da çalışabilir her programda bulunması gereken `Main` metoduydu. Artık Main metodu gibi başka metotlar yaratıp programımızın içinde kullanabileceğiz. Metotlar oluşturarak programımızı parçalara böler ve programımızın karmaşıklığını azaltırız. Ayrıca bazı kodları bir metot içine alıp aynı kodlara ihtiyacımız olduğunda bu metodu çağırabiliriz. Bu sayede de kod hamallığı yapmaktan kurtuluruz.

Metot yaratımı ve kullanımı

```
int MetotAdi(int a,int b)
{
   return a+b;
}
```

Bu metot, iki tane int türünden girdi alır ve bu girdilerin toplamını int türünden tutar. Ancak bu metodu programımız içinde kullanabilmemiz için bu metodun içinde bulunduğu sınıf türünden bir nesne yaratıp "." operatörüyle bu nesne üzerinden metodumuza erişmeliyiz. Örnek:

```
using System;
class Metotlar
{
   int Topla(int a,int b)
   {
      return a+b;
   }
   static void Main()
   {
      Metotlar nesne=new Metotlar();
      int a=nesne.Topla(2,5);
      Console.Write(a);
   }
}
```

- static olarak tanımlanan metotlara erişmek için metodun içinde bulunduğu sınıf türünden bir nesne yaratmaya gerek yoktur. static olarak tanımlanan metotlara sadece metodun adını yazarak erişilebilir. Örnek:

```
using System;
class Metotlar
{
   static int Topla(int a,int b)
   {
      return a+b;
   }
   static void Main()
   {
      int a=Topla(2,5);
      Console.Write(a);
```

```
    }
}
```

- Bütün programlarda önce Main metodu çalışır. Diğer metotlar Main metodunun içinden çağrılmadıkça çalışmaz.
- Eğer metot, içinde bulunduğumuz sınıfta değil de, başka bir sınıf içinde yaratılmışsa o metodu kullanabilmek için önce sınıfı yazmamız gerekir. Örnek:

```
using System;
class Metotlar1
{
   public static int Topla(int a,int b)
   {
      return a+b;
   }
}
class Metotlar2
{
   static void Main()
   {
      int a=Metotlar1.Topla(2,5);
      Console.Write(a);
   }
}
```

Dikkat ettiyseniz bu metodun yaratılma satırının (4. satır) başına `public` anahtar sözcüğü konmuş. `public` sözcüğüyle derleyiciye bu metoda her sınıftan erişilebileceğini söylüyoruz. Eğer public sözcüğü yazılmamış olsaydı bu metoda sadece `Metotlar1` sınıfından erişilebilirdi.

- Şimdi de static olmayan bir metodu başka bir sınıf içinde yaratalım ve kullanalım:

```
using System;
class Metotlar1
{
   public int Topla(int a,int b)
   {
      return a+b;
   }
}
class Metotlar2
{
   static void Main()
   {
      Metotlar1 nesne=new Metotlar1();
      int a=nesne.Topla(3,9);
      Console.Write(a);
   }
}
```

- Bütün değer tutan metotlar bir değermiş gibi kullanılabilir, ancak değişkenmiş gibi kullanılamaz.
- Herhangi bir değer tutmayan (`WriteLine` gibi) metotları `void` anahtar sözcüğüyle yaratırız. Örnek:

```
using System;
class Metotlar1
{
   static void Yaz(object a,int b)
   {
      for(;b>0;b--)
         Console.Write(a);
   }
   static void Main()
   {
      Yaz("deneme",5);
   }
}
```

Burada oluşturduğumuz `Yaz` metodu aldığı ilk parametreyi ikinci parametre kere ekrana yazar. Örneğin programımızda "deneme" stringi ekrana 5 kez yazdırılıyor.

- Herhangi bir değer tutmayan metotlarda `return;` komutu, yanına herhangi bir ifade olmadan kullanılabilir. Aslında `return;` komutunun asıl görevi metottan çıkmaktır, ancak yanına bazı ifadeler koyularak metodun tuttuğu değeri belirtme vazifesi de görür. Örneğin bir `if` koşulu yaratıp, eğer koşul sağlanırsa metottan çıkılmasını, koşul sağlanmazsa başka komutlar da çalıştırılmasını sağlayabiliriz. Ancak doğal olarak bir metodun son satırında `return;` komutunun kullanılması gereksizdir. Örnek kullanım:

```
using System;
class Metotlar1
{
   static void Yaz(object a,int b)
   {
       if(b>10)
         return;
       for(;b>0;b--)
         Console.Write(a);
   }
   static void Main()
   {
      Yaz('n',10);
   }
}
```

Bu programda eğer metoda verilen ikinci parametre 10'dan büyükse metottan hiçbir şey yapılmadan çıkılıyor.

Metotlarla ilgili önemli özellikler

- Metotları kullanırken parametrelerini doğru sayıda, doğru sırada ve doğru türde vermeliyiz.
- Değer tutan metotlarda `return` satırıyla belirtilen ifade, metodu yaratırken verilen türle uyumlu olmalıdır.
- Değer tutmayan (`void` ile belirtilmiş) metotlarda `return` komutunun herhangi bir ifadeyle kullanılması yasaktır.
- Değer tutmayan metotların bir değermiş gibi kullanılması yasaktır.
- Metotlar değer tutmayabileceği gibi, parametre de almayabilirler. Örnek program:

```
using System;
class Metotlar1
{
   static void Yaz()
   {
          Console.Write("deneme");
   }
   static void Main()
   {
      Yaz();
   }
}
```

Buradaki ekrana "deneme" yazan metot herhangi bir parametre almaz. Dolayısıyla da programda kullanırken de parantezlerin içine hiçbir şey yazılmaz.

- Bir metodun içinde başka bir metot yaratılamaz. Örneğin aşağıdaki gibi bir kullanım hatalıdır:

```
using System;
class Metotlar1
{
   static void Yaz()
   {
          Console.Write("deneme");
          static void Ciz()
          {
             Console.Write("\n");
          }
   }
   static void Main()
   {
      Yaz();
   }
}
```

- Metot yaratılırkenki metot parantezinde "`static void Yaz(object a,int b)`" veya metot küme parantezlerinin içinde tanımlanan değişkenler metottan çıkıldığında bellekten silinirler. Eğer aynı metot tekrar çağrılırsa söz konusu değişkenler tekrar tanımlanıp tekrar değerler atanır.
- Metot yaratılırkenki metot parantezindeki değişkenlerin türleri tek tek belirtilmelidir. Virgül ile ortak tür belirtimi yapılamaz. Yani `static void Ornek(int a,int b)` yerine `static void Ornek(int a, b)` yazılamaz.

Metot parametresi olarak diziler

Örnek:

```
using System;
class Metotlar1
{
   static void Yaz(int[] dizi)
   {
         foreach(int i in dizi)
            Console.WriteLine(i);
   }
   static void Main()
   {
         int[] dizi={1,2,4,7,9};
         Yaz(dizi);
   }
}
```

Buradaki `Yaz` metodu kendisine parametre olarak verilen dizinin elemanlarını alt alta yazdı. Eğer yalnızca int[] türündeki değil bütün türlerdeki dizileri ekrana yazan bir metot yazmak istiyorsak:

```
using System;
class Metotlar1
{
   static void Yaz(Array dizi)
   {
         foreach(object i in dizi)
            Console.WriteLine(i);
   }
   static void Main()
   {
         int[] dizi={1,2,4,7,9};
         Yaz(dizi);
   }
}
```

Bu kullanım doğrudur. Ancak aşağıdaki kullanım hatalıdır:

```
using System;
class Metotlar1
{
   static void Yaz(object[] dizi)
   {
         foreach(object i in dizi)
            Console.WriteLine(i);
   }
   static void Main()
   {
         int[] dizi={1,2,4,7,9};
```

```
          Yaz(dizi);
    }
}
```

Çünkü dizilerde değişkenlerdeki gibi bir bilinçsiz tür dönüşümünden bahsetmek imkansızdır.

Dizi ve değişken parametreler arasındaki fark

Aşağıdaki iki programı karşılaştırın:

```
using System;
class Metotlar1
{
  static void Degistir(int[] dizi)
  {
        for(int i=0;i<5;i++)
            dizi[i]=10;
  }
  static void Yaz(Array dizi)
  {
        foreach(object a in dizi)
            Console.WriteLine(a);
  }
  static void Main()
  {
        int[] dizi={1,2,4,7,8};
        Degistir(dizi);
        Yaz(dizi);
  }
}
```

```
using System;
class Metotlar1
{
  static void Degistir(int sayi)
  {
        sayi=10;
  }
  static void Yaz(int sayi)
  {
        Console.WriteLine(sayi);
  }
  static void Main()
  {
        int sayi=1;
        Degistir(sayi);
        Yaz(sayi);
  }
}
```

Bu iki programı çalıştırdığınızda göreceksiniz ki metoda parametre olarak aktarılan dizinin metot içinde bir elemanının değiştirilmesi esas diziyi etkiliyor. Ancak metoda parametre olarak aktarılan değişkenin metot içinde değiştirilmesi esas değişkeni etkilemiyor. Çünkü bir metoda parametre olarak bir dizi verildiğinde derleyici metoda dizinin bellekteki adresini verir; metot o adresteki verilerle çalışır. Dolayısıyla da dizinin herhangi bir elemanındaki değişiklik esas diziyi etkileyecektir. Çünkü gerek esas program, gerekse de metot aynı adresteki verilere erişir. Halbuki bir metoda parametre olarak bir değişken verdiğimizde metot için değişkenin bellekteki adresi önemli değildir, metot için önemli olan değişkenin değeridir. Metot, değişkeni kullanabilmek için geçici bir bellek bölgesi yaratır ve parametre olarak aldığı değişkenin değerini bu geçici bellek bölgesine kopyalar ve o geçici bellek bölgesiyle çalışır. Metottan çıkıldığında da o geçici bellek bölgesi silinir.

Peki bir metoda aktarılan bir değişkende yapılan bir değişikliğin tıpkı dizilerdeki gibi esas değişkeni etkilemesini ister miydiniz? İşte bunun için C#'ın `ref` ve `out` olmak üzere iki anahtar sözcüğü vardır.

ref anahtar sözcüğü

Örnek:

```
using System;
class Metotlar1
{
  static void Degistir(ref int sayi)
  {
        sayi=10;
  }
  static void Yaz(int sayi)
  {
        Console.WriteLine(sayi);
  }
  static void Main()
  {
        int sayi=1;
        Degistir(ref sayi);
        Yaz(sayi);
  }
}
```

`ref` anahtar sözcüğü değişkenlerin metotlara *adres gösterme* yoluyla aktarılmasını sağlar. Gördüğünüz gibi `ref` sözcüğünün hem metodu çağırırken , hem de metodu yaratırken değişkenden önce yazılması gerekiyor. Bu program ekrana 10 yazacaktır. Ayrıca `ref` sözcüğüyle bir değişkenin metoda adres gösterme yoluyla aktarılabilmesi için esas programda değişkene bir ilk değer verilmelidir. Yoksa program hata verir. Örneğin aşağıdaki program hata verir:

```
using System;
class Metotlar1
{
  static void Degistir(ref int sayi)
  {
        sayi=10;
  }
  static void Yaz(int sayi)
```

```
  {
        Console.WriteLine(sayi);
  }
  static void Main()
  {
        int sayi;
        Degistir(ref sayi);
        Yaz(sayi);
  }
}
```

out anahtar sözcüğü

Kullanımı `ref` anahtar sözcüğüyle tamamen aynıdır. Tek farkı `out` ile belirtilen değişkenlere esas programda bir ilk değer verilmesinin zorunlu olmamasıdır. Örneğin aşağıdaki kullanım tamamen doğrudur.

```
using System;
class Metotlar
{
  static void Degistir(out int sayi)
  {
        sayi=10;
  }
  static void Yaz(int sayi)
  {
        Console.WriteLine(sayi);
  }
  static void Main()
  {
        int sayi;
        Degistir(out sayi);
        Yaz(sayi);
  }
}
```

NOT: `ref` sözcüğünün dizilerle kullanımı gereksiz olmasına rağmen C# bunu kısıtlamamıştır. Ancak `out` sözcüğü dizilerle kullanılamaz.

Metotların aşırı yüklenmesi

C#'ta parametre sayısı ve/veya parametrelerin türleri farklı olmak şartıyla aynı isimli birden fazla metot yaratılabilir. Buna metotların aşırı yüklenmesi denir.

C#, bir metot çağrıldığında ve çağrılanla aynı isimli birden fazla metot bulunduğunda metodun çağrılış biçimine bakar. Yani ana programdaki metoda girilen parametrelerle yaratılmış olan metotların parametrelerini kıyaslar. Önce parametre sayısına bakar. Eğer aynı isimli ve aynı sayıda parametreli birden fazla metot varsa bu sefer parametre türlerinde tam uyumluluk arar, parametre türlerinin tam uyumlu olduğu bir metot bulamazsa bilinçsiz tür dönüşümünün mümkün olduğu bir metot arar, onu da bulamazsa programımız hata verir. Örnekler:

```
using System;
class Metotlar
```

```
{
  static void Metot1(int x,int y)
  {
        Console.WriteLine("1. metot çağrıldı.");
  }
  static void Metot1(float x,float y)
  {
        Console.WriteLine("2. metot çağrıldı.");
  }
  static void Metot1(string x,string y)
  {
        Console.WriteLine("3. metot çağrıldı.");
  }
  static void Main()
  {
        Metot1("deneme","deneme");
        Metot1(5,6);
        Metot1(10f,56f);
  }
}
```

Bu programda üç metot da aynı sayıda parametre almış. Bu durumda parametrelerin türlerine bakılır. Ana programdaki `Metot1("deneme","deneme");` satırıyla üçüncü metot çağrılır. `Metot1(5,6);` metot çağrımının parametre türlerinin tam uyumlu olduğu metot birinci metottur, o yüzden birinci metot çağrılır. Eğer birinci metot yaratılmamış olsaydı ikinci metot çağrılacaktı. Son olarak `Metot1(10f,56f);` satırıyla da ikinci metot çağrılır. Başka bir örnek:

```
using System;
class Metotlar
{
  static void Metot1(float x,float y)
  {
        Console.WriteLine("1. metot çağrıldı.");
  }
  static void Metot1(double x,double y)
  {
        Console.WriteLine("2. metot çağrıldı.");
  }
  static void Main()
  {
        Metot1(5,6);
  }
}
```

Bu programda iki metodun da parametre sayısı eşit, iki metotta da tam tür uyumu yok ve iki metotta da bilinçsiz tür dönüşümü mümkün. Bu durumda en az kapasiteli türlü metot çağrılır. Yani bu programda birinci metot çağrılır. Başka bir örnek:

```
using System;
class Metotlar
{
  static void Metot1(float x,float y)
  {
        Console.WriteLine("1. metot çağrıldı.");
  }
  static void Metot1(int x,int y)
  {
        Console.WriteLine("2. metot çağrıldı.");
  }
  static void Main()
  {
        Metot1(5,6.4f);
  }
}
```

Bu durumda birinci metot çağrılır. Başka bir örnek:

```
using System;
class Metotlar
{
  static void Metot1(float x,float y)
  {
        Console.WriteLine("1. metot çağrıldı.");
  }
  static void Metot1(int x,int y)
  {
        Console.WriteLine("2. metot çağrıldı.");
  }
  static void Main()
  {
        Metot1('f','g');
  }
}
```

Bu durumda ikinci metot çağrılır. Çünkü char hem inte hem de floata bilinçsiz olarak dönüşebilir. Ancak int daha az kapasitelidir.

NOT: Metotların geri dönüş tipi (tuttuğu değerin tipi) faydalanılabilecek ayırt edici özelliklerden değildir. Yani iki metodun parametre sayısı ve parametre türleri aynı ise tuttuğu değer tipleri farklı olsa bile bunların herhangi biri çağrılmak istendiğinde programımız derlenmeyecektir.

```
using System;
class Metotlar
{
  static void Metot1(int x,int y,int z)
  {
        Console.WriteLine("1. metot çağrıldı.");
  }
```

```
  static void Metot1(int x,int y)
  {
        Console.WriteLine("2. metot çağrıldı.");
  }
  static void Metot1(float x,int y)
  {
        Console.WriteLine("3. metot çağrıldı.");
  }
  static void Main()
  {
        Metot1(3,3,6);
        Metot1(3.4f,3);
        Metot1(1,'h');
  }
}
```

Burada sırasıyla 1., 3. ve 2. metotlar çağrılacaktır.

Değişken sayıda parametre alan metotlar

Şimdiye kadar metotlarımıza gönderdiğimiz parametre sayısı belliydi ve bu parametre sayısından farklı bir miktarda parametre girersek programımız hata veriyordu. Artık istediğimiz sayıda parametre girebileceğimiz metotlar yapmasını göreceğiz. Örnek:

```
using System;
class Metotlar
{
   static int Toplam(params int[] sayilar)
   {
      if(sayilar.Length==0)
         return 0;
      int toplam=0;
      foreach(int i in sayilar)
         toplam+=i;
      return toplam;
   }
   static void Main()
   {
      Console.WriteLine(Toplam());
      Console.WriteLine(Toplam(5));
      Console.WriteLine(Toplam(5,10));
      Console.WriteLine(Toplam(2,9,12));
      Console.WriteLine(Toplam(7,12,45));
      Console.WriteLine(Toplam(123,12,5,7,9,4,12));
   }
}
```

Burada aldığı parametrelerin toplamını tutan bir metot yazdık. Eğer metoda hiç parametre girilmemişse 0 değerini döndürmektedir. Gördüğünüz gibi `params` anahtar sözcüğüyle girilen tüm parametreleri int[] türündeki sayilar

dizisine aktardık ve bu diziyi metotta gönlümüzce kullandık. Bir de bu programımızda `Length` özelliğini gördünüz. Özellikler metotlara benzerler, metotlardan tek farkları "()" kısımları olmamasıdır. Dolayısıyla parametre almazlar. Örneğimizde `sayilar.Length` satırı sayilar dizisinin eleman sayısını int türünden tutar. Başka bir örnek:

```
using System;
class Metotlar
{
   static int Islem(string a,params int[] sayilar)
   {
      if(a=="carp")
      {
         if(sayilar.Length==0)
            return 1;
         int carpim=1;
         foreach(int i in sayilar)
            carpim*=i;
         return carpim;
      }
      else if(a=="topla")
      {
         if(sayilar.Length==0)
            return 0;
         int toplam=0;
         foreach(int i in sayilar)
            toplam+=i;
         return toplam;
      }
      else
         return 0;
   }
   static void Main()
   {
      Console.WriteLine(Islem("topla",3,4,7,8));
      Console.WriteLine(Islem("carp",5,23,6));
   }
}
```

Bu programdaki gibi metodumuzda değişken parametre yanında bir ya da daha fazla normal sabit parametre de olabilir. Ancak değişken parametre mutlaka en sonda yazılmalıdır.

NOT: Değer döndüren metotlarımız mutlaka her durumda değer döndürmelidir. Örneğin metodumuzda sondaki `else` kullanılmazsa programımız derlenmez. Çünkü else'i kullanmasaydık birinci parametre yalnızca "carp" veya "topla" olduğunda metodumuz bir değer döndürecekti. Ve bu da C# kurallarına aykırı.

NOT: Değişken sayıda parametre alan metotlar aşırı yüklenmiş metotlar olduğunda değerlendirilmeye alınmaz. Örnek:

```
using System;
class Metotlar
```

```
{
  static void Metot1(int x,int y)
  {
        Console.WriteLine("1. metot çağrıldı.");
  }
  static void Metot1(int x,params int[] y)
  {
        Console.WriteLine("2. metot çağrıldı.");
  }
  static void Main()
  {
        Metot1(3,6);
  }
}
```

Burada 1. metot çağrılır. Ancak,

```
using System;
class Metotlar
{
  static void Metot1(int x,int y)
  {
        Console.WriteLine("1. metot çağrıldı.");
  }
  static void Metot1(int x,params int[] y)
  {
        Console.WriteLine("2. metot çağrıldı.");
  }
  static void Main()
  {
        Metot1(3,6,8);
  }
}
```

Burada 2. metot çağrılır.

Kendini çağıran metotlar

C#'ta bir metodun içinde aynı metot çağrılabilir. Örnek:

```
using System;
class Metotlar
{
   static int Faktoriyel(int a)
   {
      if(a==0)
         return 1;
      return a*Faktoriyel(a-1);
   }
   static void Main()
```

```
    {
        Console.WriteLine(Faktoriyel(0));
        Console.WriteLine(Faktoriyel(1));
        Console.WriteLine(Faktoriyel(4));
        Console.WriteLine(Faktoriyel(6));
    }
}
```

Programlamadaki metot kavramı aslında matematikteki fonksiyonlar konusunun aynısıdır. Matematikteki fonksiyonlar konusunda öğrendiğiniz bütün kuralları metotlarda uygulayabilirsiniz. Örneğin yukarıdaki örnek, matematikteki fonksiyonları iyi bilen birisi için fazla karmaşık gelmeyecektir. Örneğin:

- Matematikteki fonksiyonlar konusunu bilen birisi bilir ki bir fonksiyona parametre olarak o fonksiyonun tersini verirsek sonuç x çıkacaktır. Örneğin f(x)=2x+5 olsun. f(x) fonksiyonunun tersi $f^{-1}(x)=(x-5)/2$'dir. Şimdi f(x) fonksiyonuna parametre olarak (x-5)/2 verirsek sonuç x olacaktır. Yani $f(x)\bigcirc f^{-1}(x)=x$'tir. Şimdi bu kuralı bir metotla doğrulayalım.

```
using System;
class Metotlar
{
    static float Fonksiyon(float x)
    {
        return 2*x+5;
    }
    static float TersFonksiyon(float x)
    {
        return (x-5)/2;
    }
    static void Main()
    {
        float x=10;
        Console.WriteLine(Fonksiyon(x));
        Console.WriteLine(TersFonksiyon(x));
        Console.WriteLine(Fonksiyon(TersFonksiyon(x)));
    }
}
```

Bu program ekrana sırasıyla 25, 2.5 ve 10 yazacaktır. Eğer ortada bir bölme işlemi varsa ve matematiksel işlemler yapmak istiyorsak değişkenler int türünden değil, float ya da double türünden olmalıdır. Çünkü int türü ondalık kısmı almaz, dolayısıyla da int türüyle yapılan bölme işlemlerinde hatalı sonuçlar çıkabilir. Çünkü örneğin 2 sayısı 2.5 sayısına eşit değildir.

Main metodu

Bildiğiniz gibi çalışabilir her programda Main metodunun bulunması gerekiyor. Bütün programlarda önce Main metodu çalıştırılır. Diğer metotlar Main metodunun içinden çağrılmadıkça çalışmaz. Bu özellikler dışında Main metodunun diğer metotlardan başka hiçbir farkı yoktur. Ancak şimdiye kadar Main metodunu yalnızca `static void Main()` şeklinde yarattık. Yani herhangi bir parametre almadı ve herhangi bir değer tutmadı. Ancak Main metodunun bir değer tutmasını veya parametre almasını sağlayabiliriz.

Main metodunun değer tutması

Main metodunu `static int Main()` şeklinde de yaratabiliriz. Peki bu ne işimize yarayacak? Buradaki kritik soru "Main metodunu kim çağırıyor?"dur. Biraz düşününce bu sorunun cevabının işletim sistemi olduğunu göreceksiniz. Peki işletim sistemi Main metodunun tuttuğu değeri ne yapacak? Programlar çeşitli şekillerde sonlanabilirler. Örneğin Main metodu 0 değerini döndürürse işletim sistemi programın düzgün şekilde sonlandırıldığını, 1 değerini döndürürse de hatalı sonlandırıldığını anlayabilecektir. Ancak şimdilik bunları kullanmamıza gerek yok.

Main metodunun parametre alması

Tahmin edeceğiniz gibi Main metoduna parametreler işletim sisteminden verilir. Main metodu, komut satırından girilen argümanları string türünden bir diziye atayıp programımızda gönlümüzce kullanmamıza izin verir.

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

Burada komut satırından girilen argümanlar string[] türündeki args dizisine aktarılıyor. Bu diziyi programımızda gönlümüzce kullanabiliriz. Örneğin programımız deneme.exe olsun. Komut satırından

```
deneme ayşe bekir rabia
```

girilirse ilk sözcük olan `deneme` ile programımız çalıştırılır ve ayşe, bekir ve rabia sözcükleri de string[] türündeki args dizisine aktarılır. Örnek bir program:

```
using System;
 class Metotlar
 {
    static void Main(string[] args)
    {
       Console.WriteLine("Komut satırından şunları girdiniz: ");
       foreach(string i in args)
          Console.WriteLine(i);
    }
 }
```

Bu program komut satırından girilen argümanları ekrana alt alta yazacaktır. Komut satırında program adından sonra girilen ilk sözcük args dizisinin 0. indeksine, ikinci sözcük 1. indeksine, üçüncü sözcük 2. indeksine vs. aktarılır.

System.Math sınıfı ve metotları

.Net sınıf kütüphanesinde belirli matematiksel işlemleri yapan birçok metot ve iki tane de özellik vardır. System.Math sınıfındaki metotlar static oldukları için bu metotları kullanabilmek için içinde bulundukları sınıf türünden bir nesne yaratmaya gerek yoktur. System.Math sınıfındaki iki özellik matematikteki pi ve e sayılarıdır. Şimdi bu iki özelliği örnek bir programda görelim:

```
 using System;
 class Metotlar
 {
    static void Main()
    {
       double e=Math.E;
       double pi=Math.PI;
       Console.Write("e->"+e+" pi->"+pi);
```

```
    }
}
```

PI ve E özellikleri double türünden değer tutarlar. Şimdi System.Math sınıfındaki bütün metotları bir tablo hâlinde verelim:

Metot	Açıklama
Abs(x)	Bir sayının mutlak değerini tutar.
Cos(x)	Bir sayının kosinüsünü tutar.
Sin(x)	Bir sayının sinüsünü tutar.
Tan(x)	Bir sayının tanjantını tutar.
Ceiling(x)	x sayısından büyük en küçük tam sayıyı tutar (yukarı yuvarlama).
Floor(x)	x sayısından küçük en büyük tam sayıyı tutar (aşağı yuvarlama).
Max(x,y)	x ve y sayılarının en büyüğünü tutar.
Min(x,y)	x ve y sayılarının en küçüğünü tutar.
Pow(x,y)	x üzeri y'yi tutar.
Sqrt(x)	x'in karekökünü tutar.
Log(x)	x sayısının e tabanında logaritmasını tutar.
Exp(x)	e üzeri x'in değerini tutar.
Log10(x)	x sayısının 10 tabanındaki logaritmasını tutar.

Şimdi bir örnek verelim:

```
using System;
class Metotlar
{
   static void Main()
   {
      int a=Math.Max(10,34);
      int b=Math.Abs(-3);
      double c=Math.Ceiling(12.67);
      Console.Write("Max:"+a+" Abs"+b+" Ceiling:"+c);
   }
}
```

Bu metotlar çeşitli türlerden değer döndürebilirler. Örneğin Ceiling metodu double türünden değer döndürürken başka bir metot int türünden değer döndürebilir. Bunları deneyerek bulabilirsiniz. Açıklama yapma gereksinimi görmüyorum.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Sınıflar

Sınıflar nesne yönelimli programlamanın en önemli ögesidir. Sınıflar sayesinde programlarımızı parçalara bölüp karmaşıklığını azaltırız.

C#'ta metotlar ve özellikler mutlaka bir sınıfın içinde olmalıdır. Metotlar bir veya daha fazla komutun bir araya getirilmiş halidir; parametre alabilirler, geriye değer döndürebilirler. Özellikler ise bellek gözeneklerinin programlamadaki karşılıklarıdır. Bu bakımdan özellikler değişkenlere benzerler. Aradaki en temel fark değişkenlerin bir metot içinde tanımlanıp yalnızca tanımlandığı metot içinde etkinlik gösterebilmesine rağmen özelliklerin tıpkı metotlar gibi bir üye eleman olmasıdır. Bu bakımdan özelliklerin tuttuğu değerlere daha fazla yerden erişilebilir.

Sınıf Oluşturmak

Üç özelliği ve iki metodu olan örnek bir sınıf oluşturulması:

```
class SinifIsmi
{
   public int ozellik1;
   private string ozellik2;
   float ozellik3;
   public int metot1(int a,int b)
   {
      return a+b;
   }
   private void metot2(string a)
   {
      Console.WriteLine(a);
   }
}
```

Burada dikkatinizi `public` ve `private` anahtar sözcükleri çekmiş olmalı. Bir metot ya da özelliğe bulunduğu sınıfın dışından da erişilebilmesini istiyorsak `public` sözcüğü kullanılır. `private` sözcüğü kullanmakla hiçbir şey yazmamak aynı etkiye sahiptir ve iki durumda da metot ya da özelliğe yalnızca bulunduğu sınıfın içinden erişilebilir. Bu sınıftaki yalnızca `ozellik1` özelliğine ve `metot1` metoduna `SinifIsmi` sınıfının dışından erişilebilir. Diğer özellik ve metotlara erişilemez. Şimdi bu sınıfı programımızda kullanalım:

```
using System;
class SinifIsmi
```

```
{
   public int ozellik1;
   public string ozellik2;
   public float ozellik3;
   public int metot1(int a,int b)
   {
      return a+b;
   }
   public void metot2(string a)
   {
      Console.WriteLine(a);
   }
}
class EsasSinif
{
   static void Main()
   {
      SinifIsmi nesne=new SinifIsmi();
      Console.WriteLine(nesne.ozellik1);
      Console.WriteLine(nesne.ozellik2);
      Console.WriteLine(nesne.ozellik3);
      Console.WriteLine(nesne.metot1(2,5));
      nesne.metot2("deneme");
   }
}
```

Bu programda `SinifIsmi` sınıfındaki bütün özellik ve metotların `public` anahtar sözcüğü ile belirtildiğine dikkat edin. Oluşturduğumuz`nesne` nesnesiyle sınıfın bütün özellik ve metotlarına eriştik. `ozellik1`, `ozellik2` ve `ozellik3` özelliklerine hiçbir değer atanmamış, ancak programımız hata vermedi, sadece derleme sonunda uyarı verdi. Çünkü `SinifIsmi nesne=new SinifIsmi();` satırındaki `new` anahtar sözcüğü sayesinde sınıftaki bütün özellikler `nesne` nesnesi için türlerine göre varsayılan değere atandı. Eğer `SinifIsmi nesne=new SinifIsmi();` yerine `SinifIsmi nesne;` yazsaydık programımız hata verirdi. Ayrıca programımızda `metot2` metodu değer tutan değil, iş yapan bir metot, o yüzden tek başına kullanıldı. Başka bir örnek:

```
using System;
class SinifIsmi
{
   public int ozellik1=55;
   public string ozellik2="deneme";
   public float ozellik3=123.78f;
   public int metot1(int a,int b)
   {
      return a+b;
   }
   public void metot2(string a)
   {
      Console.WriteLine(a);
```

```
    }
}
class EsasSinif
{
    static void Main()
    {
        SinifIsmi nesne=new SinifIsmi();
        Console.WriteLine(nesne.ozellik1);
        Console.WriteLine(nesne.ozellik2);
        Console.WriteLine(nesne.ozellik3);
        Console.WriteLine(nesne.metot1(2,5));
        nesne.metot2("Vikikitap");
    }
}
```

Bu programda ise özelliklere önce değer verdik ve esas programda da bu değerler ekrana yazıldı.

Aynı sınıf türünden birden fazla nesne oluşturma

Tahmin edebileceğiniz gibi C#'ta aynı sınıf türünden birden fazla nesne oluşturulabilir ve bu nesnelerle sınıfın özellik ve metotlarına erişilebilir. Örnek:

```
using System;
class KrediHesabi
{
    public ulong HesapNo;
}
class AnaSinif
{
    static void Main()
    {
        KrediHesabi hesap1=new KrediHesabi();
        KrediHesabi hesap2=new KrediHesabi();
        hesap1.HesapNo=3456;
        hesap2.HesapNo=1111;
        Console.WriteLine(hesap1.HesapNo);
        Console.WriteLine(hesap2.HesapNo);
    }
}
```

Burada `KrediHesabi` sınıfı türünden `hesap1` ve `hesap2` adlı iki nesne tanımlandı ve bu nesnelerle `KrediHesabi` sınıfının `HesapNo` özelliğine erişildi. Burada bilmemiz gereken şey farklı nesnelerle bir sınıfın özelliğine erişip özelliğin değerini değiştirdiğimizde aslında birbirinden farklı değişkenlerin değerini değiştirmiş olmamızdır. Bu program ekrana alt alta 3456 ve 1111 yazacaktır. Çünkü `KrediHesabi hesap1=new KrediHesabi();` ve `KrediHesabi hesap2=new KrediHesabi();` satırlarıyla birbirinden farklı iki nesne oluşturduk ve bu iki nesne için bellekte ayrı ayrı alan tahsisatı yaptık. Eğer programı şöyle değiştirirsek,

```
using System;
class KrediHesabi
{
```

```
    public ulong HesapNo;
}
class AnaSinif
{
    static void Main()
    {
        KrediHesabi hesap1=new KrediHesabi();
        KrediHesabi hesap2=hesap1;
        hesap1.HesapNo=3456;
        hesap2.HesapNo=1111;
        Console.WriteLine(hesap1.HesapNo);
        Console.WriteLine(hesap2.HesapNo);
    }
}
```

Bu durumda alt alta 1111 ve 1111 yazılacaktır. Çünkü `KrediHesabi hesap2=hesap1;` satırıyla `KrediHesabi` sınıfı türünden yeni bir nesne oluşturduk ancak bu nesne için bellekte ayrı bir alan tahsisatı yapmadık. Nesneyi `hesap1` nesnesinin bellekteki adresine yönlendirdik. Dolayısıyla `hesap1`'in özelliğinde yapılan bir değişiklik `hesap2`'yi, `hesap2`'nin özelliğinde yapılan bir değişiklik de `hesap1`'i etkileyecektir. Başka bir örnek program:

```
using System;
class EsasSinif
{
    int a;
    static void Main()
    {
        EsasSinif nesne=new EsasSinif();
        Console.WriteLine(nesne.a);
    }
}
```

Gördüğünüz gibi özellik ve metot tanımlamaları için ayrı bir sınıf oluşturmak zorunda değiliz. Main() metodunun olduğu sınıfa istediğimiz üye elemanları ekleyebiliriz. Bu durumda a özelliğine erişebilmek için a özelliğini public olarak belirtme zorunluluğundan da kurtulmuş olduk.

NOT: Farkındaysanız şimdiye kadar hiçbir metot ya da özelliğimizi `static` anahtar sözcüğüyle belirtmedik. `static` anahtar sözcüğüyle belirttiğimiz metot ya da özellikleri direkt olarak söz konusu metot ya da özelliğin adını yazarak kullanabilriz. Örnek:

```
using System;
class EsasSinif
{
    static int a;
    static void Main()
    {
        Console.WriteLine(a);
    }
}
```

Bu program ekrana 0 yazacaktır. Eğer a'ya bir değer verseydik o değer yazacaktı.

NOT: Diziler tıpkı değişkenler gibi birbirine atanabilirler. Örnek:

```
int[] a={1,6,7,12};
int[] b=a;
```

Burada b dizisine a dizisi adres gösterme yoluyla aktarılmıştır, dolayısıyla da a dizisinin bir elemanında yapılan bir değişiklik b dizisini, b dizisinin bir elemanında yapılan bir değişiklik a dizisini etkileyecektir. Çünkü diziler birbirlerine adres gösterme yoluyla atanır. Ayrıca aşağıdaki gibi bir kullanım da doğrudur:

```
int[] a={1,6,7,12};
int[] b={1,5};
b=a;
```

Burada b dizisi sahip olduğu elemanları kaybedip a dizisinin adresine yönlendi. Bu ve benzer şekilde diziler birbirlerine atanabilir. Yani aynı adrese yönlendirilebilirler.

Örnekler

Bu program bir metoda girilen iki parametreyi ve bunların çarpımını ekrana yazar. Eğer bir dikdörtgen söz konusu olduğunu düşünürsek dikdörtgenin enini, boyunu ve alanını ekrana yazar.

```
using System;
class Dortgen
{
   public int En;
   public int Boy;
   public int Alan()
   {
      int Alan=En*Boy;
      return Alan;
   }
   public void EnBoyBelirle(int en,int boy)
   {
      En=en;
      Boy=boy;
   }
   public void Yaz()
   {
      Console.WriteLine("***************");
      Console.WriteLine("En:{0,5}",En);
      Console.WriteLine("Boy:{0,5}",Boy);
      Console.WriteLine("Alan:{0,5}",Alan());
      Console.WriteLine("***************");
   }
}
class AnaSinif
{
   static void Main()
   {
```

```
        Dortgen d1=new Dortgen();
        d1.EnBoyBelirle(20,50);
        d1.Yaz();
        Dortgen d2=new Dortgen();
        d2.EnBoyBelirle(25,12);
        d2.Yaz();
    }
}
```

Bu program Main bloğundan çalışmaya başlar. Önce `Dortgen` sınıfı türünden d1 nesnesi oluşturulur. Bu nesneyle Dortgen sınıfındaki `EnBoyBelirle` metodu çalıştırılır. Bu metot geçerli nesne için `En` ve `Boy` özelliğini metoda girilen parametreler yapar. Sonra aynı nesneyle `Yaz` metodu çalıştırılıyor. `Yaz` metodu da geçerli nesnenin özelliklerini ve `Alan` metodunun sonucunu ekrana yazdırıyor. Sonra aynı olaylar d2 nesnesi için de gerçekleşiyor. Burada önemli olan bir sınıf türünden oluşturduğumuz her bir nesne için bellekte o sınıftaki tüm metot ve özellikler için bir yer açılması ve tüm işlemlerin o bellek bölgesinde yapılması olayını kavramamız. Ayrıca gördüğünüz gibi static olmayan metotların gövdesinde aynı sınıftaki özellik ve metotlar static olsun olmasın direkt olarak kullanılabiliyor. Ancak programı şöyle değiştirseydik hata çıkacaktı.

```
using System;
class Dortgen
{
    public int En=20;
    public int Boy=5;
    public int Alan()
    {
        int Alan=En*Boy;
        return Alan;
    }
    static void Main()
    {
        Console.WriteLine("***************");
        Console.WriteLine("En:{0,5}",En);
        Console.WriteLine("Boy:{0,5}",Boy);
        Console.WriteLine("Alan:{0,5}",Alan());
        Console.WriteLine("***************");
    }
}
```

Çünkü burada `En` ve `Boy` özellikleriyle `Alan()` metodu tek başına kullanılamaz. Önce `Dortgen` sınıfı türünden bir nesne oluşturup bu nesne üzerinden bu özellik ve metotlara erişilmelidir. Çünkü Main metodu static bir metottur. Başka bir örnek:

```
using System;
class Dortgen
{
    public int En;
    public int Boy;
    public int Alan()
    {
```

```
      int Alan=En*Boy;
      return Alan;
   }
   public void EnBoyBelirle(int en,int boy)
   {
      En=en;
      Boy=boy;
   }
   public void Yaz()
   {
      Console.WriteLine("***************");
      Console.WriteLine("En:{0,5}",En);
      Console.WriteLine("Boy:{0,5}",Boy);
      Console.WriteLine("Alan:{0,5}",Alan());
      Console.WriteLine("***************");
   }
   static void Main()
   {
      Dortgen d1=new Dortgen();
      d1.EnBoyBelirle(20,50);
      d1.Yaz();
      Dortgen d2=new Dortgen();
      d2.EnBoyBelirle(25,12);
      d2.Yaz();
  }
}
```

Bu program da geçerlidir. Başka bir örnek:

```
using System;
class Dortgen
{
   public int En=30;
   public int Boy=23;
   public int Alan()
   {
      int Alan=En*Boy;
      return Alan;
   }
   static void Main()
   {
      yaz d1=new yaz();
      d1.Yaz();
  }
}
class yaz
{
    public void Yaz()
```

```
    {
        Console.WriteLine("***************");
        Console.WriteLine("En:{0,5}",Dortgen.En);
        Console.WriteLine("Boy:{0,5}",Dortgen.Boy);
        Console.WriteLine("Alan:{0,5}",Dortgen.Alan());
        Console.WriteLine("***************");
    }
}
```

Bu program çalışmaz. Çünkü, static olmayan metotların gövdesinde sadece aynı sınıftaki özellik ve metotlar static olsun olmasın direkt olarak kullanılabilir.

Şimdi evimizin üyelerinin (annemizin, babamızın vs.) özelliklerini gireceğimiz, yaşlarını hesaplayacağımız daha zevkli bir örnek yapalım. Yani evimizi bir sınıfmış gibi düşünelim.

```
using System;
class EvHalki
{
    public int DogumYili;
    public string Meslek;
    public string Ad;
    public string GozRengi;
    public int Yas()
    {
        return 2008-DogumYili;
    }
    public void OzellikleriYaz()
    {
        Console.WriteLine("---------------");
        Console.WriteLine("Adı: "+Ad);
        Console.WriteLine("Yaşı: "+Yas());
        Console.WriteLine("Mesleği: "+Meslek);
        Console.WriteLine("Göz rengi: "+GozRengi);
        Console.WriteLine("---------------");
    }
}
class AnaProgram
{
    static void Main()
    {
        EvHalki annem=new EvHalki(); //EvHalki sınıfı türünden yeni bir
nesne oluşturuldu.
        annem.DogumYili=1964; //Oluşturulan nesnenin bir özelliği
girildi.
        annem.Meslek="Ev hanımı";
        annem.Ad="Hatice";
        annem.GozRengi="Mavi";
        EvHalki babam=new EvHalki();
        babam.DogumYili=1950;
```

```
        babam.Meslek="Emekli";
        babam.Ad="Kenan";
        babam.GozRengi="Kahverengi";
        EvHalki kardesim=new EvHalki();
        kardesim.DogumYili=1987;
        kardesim.Meslek="Öğrenci";
        kardesim.Ad="Uğur";
        kardesim.GozRengi="Yeşil";
        annem.OzellikleriYaz();
        babam.OzellikleriYaz();
        kardesim.OzellikleriYaz();
    }
}
```

Şimdi bu programda ufak bir oynama yapalım:

```
using System;
class EvHalki
{
    public int DogumYili;
    public string Meslek;
    public string Ad;
    public string GozRengi;
    public int Yas()
    {
        return 2008-DogumYili;
    }
    public void OzellikleriYaz()
    {
        Console.WriteLine("---------------");
        Console.WriteLine("Adı: "+Ad);
        Console.WriteLine("Yaşı: "+Yas());
        Console.WriteLine("Mesleği: "+Meslek);
        Console.WriteLine("Göz rengi: "+GozRengi);
        Console.WriteLine("---------------");
    }
}
class AnaProgram
{
    static void Main()
    {
        EvHalki annem=new EvHalki();
        annem.DogumYili=1964;
        annem.Meslek="Ev hanımı";
        annem.Ad="Hatice";
        annem.GozRengi="Mavi";
        EvHalki babam=new EvHalki();
        babam.OzellikleriYaz();
```

```
    }
}
```

Gördüğünüz gibi annemin özellikleri girilmesine rağmen `OzellikleriYaz` metodu çağrılmadığı için özellikler ekrana yazılmadı. Ayrıca da babama da herhangi bir özellik girilmemesine rağmen `OzellikleriYaz` metoduyla ekrana yazdırdık. Ekrana bütün özelliklerin varsayılanları yazıldı. Çünkü `EvHalki babam=new EvHalki();` satırındaki altı çizili kısım sayesinde bellekte babam nesnesi için ayrı bir yer ayrıldı ve bütün özellikler varsayılan değere atandı. Yalnızca `EvHalki babam;` satırı olsaydı Evhalki sınıfı türünden babam nesnesi oluşturulurdu, ancak bellekte bu nesne için ayrı bir yer ayrılmazdı.

this anahtar sözcüğü

Şimdi şöyle bir program yazalım:

```
using System;
class Dortgen
{
   public int En;
   public int Boy;
   void EnBoyBelirle(int en,int boy)
   {
      En=en;
      Boy=boy;
   }
   static void Main()
   {
      Dortgen d1=new Dortgen();
      d1.EnBoyBelirle(20,50);
      Console.WriteLine(d1.En+"\n"+d1.Boy);
   }
}
```

Tahmin edebileceğiniz bu programda önce d1 nesnesinin En ve Boy özelliğini sırasıyla 20 ve 50 olarak ayarlıyoruz, sonra da bu d1 nesnesinin özelliklerini ekrana yazdırıyoruz. Bu programı şöyle de yazabilirdik:

```
using System;
class Dortgen
{
   public int En;
   public int Boy;
   void EnBoyBelirle(int en,int boy)
   {
      this.En=en;
      this.Boy=boy;
   }
   static void Main()
   {
      Dortgen d1=new Dortgen();
      d1.EnBoyBelirle(20,50);
      Console.WriteLine(d1.En+"\n"+d1.Boy);
```

```
    }
}
```

Buradaki this anahtar sözcüklerinin Türkçe karşılığı "beni çağıran nesnenin"dir. Peki ama zaten this'i yazmasak da aynı şey olurdu. Şimdi başka bir örnek yapalım:

```
using System;
class Dortgen
{
   public int En;
   public int Boy;
   void EnBoyBelirle(int En,int Boy)
   {
      En=En;
      Boy=Boy;
   }
   static void Main()
   {
      Dortgen d1=new Dortgen();
      d1.EnBoyBelirle(20,50);
      Console.WriteLine(d1.En+"\n"+d1.Boy);
   }
}
```

C# bu şekilde bir kullanıma izin verir. Çünkü Main bloğunda `EnBoyBelirle` metodunu çalıştırdığımızda sadece `EnBoyBelirle` metodu çalışır, dolayısıyla da derleyici bu metottan önce tanımlanmış olan En ve Boy özelliklerini görmez. Burada alınan parametreler En ve Boy değişkenlerine atanıyor. Sonra `En=En;` ve `Boy=Boy;` satırlarıyla da aslında alınan parametreler yine aynı değişkenlere atanıyor. Tahmin edebileceğiniz gibi aslında burada bizim yapmak istediğiniz aldığı parametreleri d1 nesnesinin En ve Boy özelliklerine aktarmak. İşte bunun için this anahtar sözcüğünü kullanılırız:

```
using System;
class Dortgen
{
   public int En;
   public int Boy;
   void EnBoyBelirle(int En,int Boy)
   {
      this.En=En;
      this.Boy=Boy;
   }
   static void Main()
   {
      Dortgen d1=new Dortgen();
      d1.EnBoyBelirle(20,50);
      Console.WriteLine(d1.En+"\n"+d1.Boy);
   }
}
```

Bu programda artık EnBoyBelirle metodunun aldığı parametreler d1 nesnesinin özelliklerine atanacaktır. Benzer şekilde C# aşağıdaki gibi bir kullanıma da izin verir.

```
using System;
 class Dortgen
 {
    int En;
    int Boy;
    static void Main()
    {
       int En=50;
       int Boy=100;
       Console.WriteLine(En+"\n"+Boy);
    }
 }
```

Eğer En ve Boy değişkenleri Main bloğunun içinde tanımlanmasaydı programımız hata verecekti. Başka bir örnek:

```
 using System;
 class Dortgen
 {
    static int En=8;
    static int Boy=3;
    static void Main()
    {
       int En=50;
       int Boy=100;
       Console.WriteLine(En+"\n"+Boy);
    }
 }
```

Burada ekrana sırasıyla 50 ve 100 yazılacaktır. Ancak;

```
 using System;
 class Dortgen
 {
    static int En=8;
    static int Boy=3;
    static void Main()
    {
       int Boy=100;
       Console.WriteLine(En+"\n"+Boy);
    }
 }
```

Bu sefer de ekrana sırasıyla 8 ve 100 yazıldı. Çünkü derleyici En değişkenini Main bloğu içinde bulamayınca sınıfın bir özelliği olan En'i ekrana yazdı. En ve Boy özellikleri static olarak tanımlandığı için bu özelliklere nesne oluşturmaya gerek kalmadan direkt erişebildik.

get ve set anahtar sözcükleri

Şimdi aşağıdaki küçük programı yazalım:

```
using System;
class YardimciSinif
{
   int Sayi;
   public void SayiBelirle(int sayi)
   {
      Sayi=sayi;
   }
   public int SayiAl()
   {
      return Sayi;
   }
}
class AnaSinif
{
   static void Main()
   {
      YardimciSinif nesne=new YardimciSinif();
      nesne.SayiBelirle(34);
      Console.WriteLine(nesne.SayiAl());
   }
}
```

Bu program oldukça basit. `YardimciSinif` sınıfının iki tane metodu ve bir tane de özelliği var. Ancak özellik private olduğu için bu özelliğe başka bir sınıftan erişmek veya değiştirmek mümkün değil. Bu yüzden bu özelliği değiştirmek veya bu özelliğe erişmek istediğimizde public olan `SayiBelirle` ve `SayiAl` metotlarını kullandık. Peki bunun bize ne faydası var? Aslında `Sayi` özelliğini public olarak ayarlayıp direkt olarak özellik üzerinde işlem yapılabilirdi. İşte faydası:

```
using System;
class YardimciSinif
{
   int Sayi;
   public void SayiBelirle(int sayi)
   {
      if(sayi<0)
         Sayi=0;
      else
         Sayi=sayi;
   }
   public int SayiyiAl()
   {
      if(Sayi>100)
         return Sayi/100;
      else
```

```
            return Sayi;
        }
    }
    class AnaSinif
    {
        static void Main()
        {
            YardimciSinif nesne=new YardimciSinif();
            nesne.SayiBelirle(34);
            Console.WriteLine(nesne.SayiAl());
        }
    }
```

Gördüğünüz gibi özelliğe değer atamayı ve özelliğin değerini ekrana yazdırmayı metotlar sayesinde koşullandırabildik. Eğer direkt özellik üzerinde işlem yapmaya kalkışsaydık böyle bir şansımız olmazdı. Şimdi benzer bir durumu set ve get sözcükleriyle oluşturalım:

```
using System;
class YardimciSinif
{
    int Sayi;
    public int SahteOzellik
    {
        set
        {
            if(value<0)
                Sayi=0;
            else
                Sayi=value;
        }
        get
        {
            if(Sayi>100)
                return Sayi/100;
            else
                return Sayi;
        }
    }
}
class AnaSinif
{
    static void Main()
    {
        YardimciSinif nesne=new YardimciSinif();
        nesne.SahteOzellik=110;
        Console.WriteLine(nesne.SahteOzellik);
    }
}
```

Gördüğünüz gibi önce `YardimciSinif` sınıfında `SahteOzellik` adlı bir özellik oluşturduk. Bu özellik gerçekten de sahtedir. Özelliğe bir değer atanmaya çalışıldığında set bloğundaki, özellik kullanılmaya çalışıldığında da get bloğundaki komutlar çalıştırılır. Aslında C# kütüphanesindeki özelliklerin çoğu bu yöntemle oluşturulmuştur. Örneğin ileride göreceğimiz Windows programlamada bir buton nesnesinin Text özelliğini değiştirdiğimizde butonun üzerindeki yazı değişir. Halbuki klasik özelliklere değer atama yönteminde sadece özelliğin değeri değişirdi. Bu yöntemde ise bir özelliğe değer atadığımızda çalışacak komutlar yazabiliyoruz. Unutmadan söyleyeyim; programdaki `value` sözcüğü özelliğe girilen değeri tutar. Özelliğe girilen değer hangi türdeyse o türde tutar. Ayrıca bu oluşturulan SahteOzellik özelliğinin metotlara oldukça benzediğini de dikkatinizi çekmiş olmalı.

NOT: Aslında tanımladığımız her dizi Array sınıfı türünden bir nesnedir. İşte bu yüzden tanımladığımız dizilerle Array sınıfının metot ve özelliklerine erişebiliriz. Örneğin `Length` Array sınıfı türünden bir özelliktir ve dizinin eleman sayısını verir. `DiziAdi.Length` yazılarak bu özelliğe erişilebilir.

NOT: Metot ve özelliklerin geri dönüş tipi bir dizi olabilir. Örnek:

```
using System;
class YardimciSinif
{
   public int[] Dizi={7,4,3};
   public int[] Metot()
   {
      int[] a={23,45,67};
      return a;
   }
}
class AnaSinif
{
   static void Main()
   {
      YardimciSinif nesne=new YardimciSinif();
      Console.WriteLine(nesne.Dizi[0]);
      Console.WriteLine(nesne.Metot()[2]);
   }
}
```

Şimdi isterseniz bir dizinin türünü istenilen türe dönüştürmeye yarayan bir sınıf yazalım. Yani sınıftaki metotlar dizinin bütün elemanlarını istenilen türe dönüştürüp bu oluşturulan yeni diziyi tutacak. C#'ta diziler arasında bilinçsiz tür dönüşümü mümkün olmadığı gibi, bu işi yapacak metot da yok. Yani yapacağımız metotlar oldukça faydalı olacaktır.

```
using System;
class Donustur
{
   public static int[] Inte(Array dizi)
   {
      int[] gecici=new int[dizi.Length];
      for(int i=0;i<dizi.Length;i++)
         gecici[i]=Convert.ToInt32(dizi.GetValue(i));
      return gecici;
   }
```

```
    public static string[] Stringe(Array dizi)
    {
        string[] gecici=new string[dizi.Length];
        for(int i=0;i<dizi.Length;i++)
            gecici[i]=dizi.GetValue(i).ToString();
        return gecici;
    }
}
```

Sınıfımız bu şekilde. Ancak sınıfımızın yalnızca inte ve stringe dönüştüm yapan iki metodu var. İsterseniz dönüşüm yapılabilecek tür sayısını yeni metotlar ekleyerek artırabilirsiniz. Şimdi bu sınıfı bir program içinde kullanalım.

```
using System;
class Donustur
{
    public static int[] Inte(Array dizi)
    {
        int[] gecici=new int[dizi.Length];
        for(int i=0;i<dizi.Length;i++)
            gecici[i]=Convert.ToInt32(dizi.GetValue(i));
        return gecici;
    }
    public static string[] Stringe(Array dizi)
    {
        string[] gecici=new string[dizi.Length];
        for(int i=0;i<dizi.Length;i++)
            gecici[i]=dizi.GetValue(i).ToString();
        return gecici;
    }
}
class AnaProgram
{
    static void Main()
    {
        string[] a={"2","5","7","9"};
        int[] b=Donustur.Inte(a);
        Console.WriteLine(b[1]+b[3]);
        int[] c={2,7,9,4};
        string[] d=Donustur.Stringe(c);
        Console.WriteLine(d[0]+d[3]);
    }
}
```

HATIRLATMA: Gördüğünüz gibi metotların parametresindeki dizi, `Array DiziAdi` yöntemiyle oluşturulduğu için bu dizinin elemanlarına klasik indeksleme yöntemiyle erişemeyiz. Bu yüzden bu dizinin elemanlarına ulaşmak için Array sınıfının `GetValue()` metodunu kullandık.

Convert sınıfının diziler için olanını oluşturduktan sonra şimdi de DOS ekranına bir kutu çizen sınıf hazırlayalım. Kutuyu ╔, ═, ╗, ║, ╚, ve ╝ karakterleriyle oluşturacağız. Bu karakterlerin Unicode karşılıklarıysa:

Karakter	Unicode karşılığı (16'lık sistemde)
╔	2554
═	2550
╗	2557
║	2551
╚	255A
╝	255D

Microsoft Word'u açıp "Ekle" menüsünden "Simge"yi seçtğinizde bunlar gibi daha birçok simgeye erişebilirsiniz. Simgeleri direkt kopyala-yapıştır yapabileceğiniz gibi Unicode karşılıklarını öğrenip programınızda bu kodları da kullanabilirsiniz. Ancak eğer kopyala-yapoştır yapmışsanız kod dosyanızı (program.cs'nizi) kaydederken kodlamasını Unicode olarak değiştirin. Kodu Not Defteri'nde yazıyorsanız kaydetmek istediğinizde zaten sistem sizi uyaracaktır. Şimdi sınıfı yazmaya başlayalım:

```
using System;
class Buton
{
   public int Genislik;
   public int Yukseklik;
   public void Ciz()
   {
      string[,] dizi=new string[Yukseklik+2,Genislik+3];
      dizi[0,0]="╔";
      dizi[0,Genislik+1]="╗";
      dizi[Yukseklik+1,0]="╚";
      dizi[Yukseklik+1,Genislik+1]="╝";
      for(int i=1;i<Genislik+1;i++)
      {
         dizi[0,i]="═";
         dizi[Yukseklik+1,i]="═";
      }
      for(int i=0;i<=Yukseklik;i++)
         dizi[i,Genislik+2]="\n";
      for(int i=1;i<=Yukseklik;i++)
      {
         dizi[i,0]="║";
         dizi[i,Genislik+1]="║";
      }
      for(int j=1;j<=Yukseklik;j++)
         for(int i=1;i<=Genislik;i++)
            dizi[j,i]=" ";
     foreach(string i in dizi)
        Console.Write(i);
   }
}
class AnaProgram
```

```
{
   static void Main()
   {
      Buton buton1=new Buton();
      buton1.Genislik=20;
      buton1.Yukseklik=13;
      buton1.Ciz();
   }
}
```

Buton sınıfımız verilen ölçülerde butona benzeyen bir kutu oluşturmaktadır. Programdaki `Genislik` ve `Yukseklik` özelliklerimiz butona sığacak yatayda ve dikeydeki karakter sayısını belirtiyor. Tabii ki bu programı daha da geliştirebilirsiniz. Örneğin Buton sınıfına bir de metin özelliği ekleyebilirsiniz ve bu özellik butonun üstündeki yazıyı belirtebilir. Veya verilen ölçülerde bir tablo yapan veya kendisine parametre olarak verilen iki boyutlu bir diziyi ekrana bir tablo gibi çizen bir sınıf tasarlayabilirsiniz. Yani her şey hayal gücünüze kalmış.

NOT: Aslında bir stringi char türünden bir diziymiş gibi düşünebiliriz. Örneğin aşağıdaki ifadeler mümkündür:

```
string a="deneme";
char b=a[2];
Console.Write(b);
```

Bu program ekrana n yazar. Ancak stringin herhangi bir karakteri bu yöntemle değiştirilemez. Bu yöntemle string karakterlerine sadece okuma (read-only) amaçlı erişebiliriz. Örneğin aşağıdaki örnek hatalıdır:

```
string a="deneme";
a[2]='ş';
Console.Write(a);
```

string türündeki bir sabit ya da değişken dizilerin bazı özelliklerini sağlarken bazı özelliklerini sağlamaz:

- Tıpkı dizilerdeki gibi stringlerde de `foreach` deyimi kullanılabilir.
- Dizilerdeki Length özelliği stringlerle de kullanılabilir. Bunun dışındaki Array sınıfına ait hiçbir metot ve özellik stringlerle kullanılamaz.

NOT: Herhangi bir sahte özelliğin set veya get bloklarından yalnızca birini yazarak o özelliği salt okunur veya salt yazılır hâle getirebiliriz. Örneğin Array sınıfının Length özelliği salt okunur bir özelliktir.

NOT: C# 2.0'da ya da daha üst versiyonlarda bir sahte özelliğin set ve get bloklarını ayrı ayrı `private` veya `public` anahtar sözcükleriyle belirtebiliyoruz. Yani bir özelliğin bulunduğu sınıfın dışında salt okunur ya da salt yazılır olmasını sağlayabiliyoruz. Ancak az önce de söylediğim gibi bu özellik C# 1.0 veya C# 1.1 gibi daha önceki versiyonlarda geçerli değildir.

NOT: get ve set anahtar sözcükleriyle erişim belirleyiciler kullanırken uymamız gereken bazı özellikler vardır:

1. Daima özellik bildiriminde kullanılan erişim belirleyicisi get veya set satırında kullanılan erişim belirleyicisinden daha yüksek seviyeli olmalıdır. Örneğin özellik bildiriminde kullanılan erişim belirleyici `private` ise get veya set satırında kullanılan erişim belirleyici `public` olamaz.
2. get veya set satırında kullanılan erişim belirleyici özellik bildiriminde kullanılan erişim belirleyiciyle aynı olmamalıdır. (Zaten gereksizdir.)
3. Yani işin özü set ve get satırlarındaki erişim belirleyicileri yalnızca, özelliği public olarak belirtmiş ancak özelliğin set veya get bloklarının herhangi birisini private yapmak istediğimizde kullanılabilir.
4. get veya set için erişim belirleyicisi kullanacaksak sahte özelliğin bloğu içinde hem get hem de set bloğunun olması gerekir.

Yapıcı metotlar

Şimdiye kadar bir sınıfın üye elemanlarını (metotlar ve özellikleri) kullanabilmek için o sınıf türünden bir nesne oluşturuyorduk ve bu nesne üzerinden o sınıfın üye elemanlarına erişebiliyorduk. Bir sınıf türünden bir nesne oluşturduğumuzda -new anahtar sözcüğü sayesinde- o sınıftaki bütün özellikler ve metotlar başka bir bellek bölümüne kopyalanıyor ve bütün özellikler -sınıfta bir değer atanmamışsa- varsayılan değeri tutuyordu. Peki bir sınıf türünden bir nesne oluşturduğumuzda sınıftaki bütün özelliklerin varsayılan değeri tutması yanında başka şeyler de yapılmasını ister miydiniz? İşte bunun için yapıcı metotları kullanıyoruz. Yapıcı metotlarla ilgili bilmemiz gereken şeylerse:

- Yapıcı metotların adı sınıfın adıyla aynı olmalıdır.
- Yapıcı metotlar bir değer tutamaz. Ancak normal metotlardan farklı olarak `void` anahtar sözcüğü de kullanılmaz. Örnek program:

```
using System;
class Deneme
{
   public Deneme()
   {
      Console.WriteLine("Deneme sınıf türünden bir nesne
oluşturuldu.");
   }
}
class AnaProgram
{
   static void Main()
   {
      Deneme a=new Deneme();
   }
}
```

Bu program ekrana `Deneme sınıf türünden bir nesne oluşturuldu.` yazacaktır. Bunu bu şekilde kullanabilirsiniz. Ancak bunun asıl kullanımı sınıftaki özellikleri varsayılan değerden farklı bir değere çekmektir. Yapıcı metotların şu şekilde kullanımı da mümkündür.

```
using System;
class Deneme
{
   public Deneme(int a,int b,int c)
   {
      Console.WriteLine(a+b+c);
   }
}
class AnaProgram
{
   static void Main()
   {
      Deneme a=new Deneme(2,5,6);
   }
}
```

Gördüğünüz gibi yapıcı metot parametre de alabiliyor. Yapıcı metoda parametreler ise nesne oluşturulurken veriliyor.

NOT: İllaki yapıcı metodun asıl sınıf dışında bulunmasına gerek yoktur. Örneğin aşağıdaki gibi bir kullanım da mümkündür:

```
using System;
class Deneme
{
   Deneme()
   {
      Console.WriteLine("Bu sınıf türünden bir nesne oluşturuldu.");
   }
   static void Main()
   {
      Deneme a=new Deneme();
   }
}
```

Bu durumda yapıcı metodu `public` olarak belirtmeye gerek yok.

NOT: Yapıcı metotların erişim belirleyicilerini private yaparak sınıfın dışından o sınıf türünden nesne oluşturulmasını engellemiş oluruz.

Varsayılan yapıcı metot

Biz herhangi bir yapıcı metot oluşturmamışsak C# otomatik olarak bir yapıcı metot oluşturur. Bu metodun içi boştur, yani bir şey yapmaz. Bu metoda varsayılan yapıcı metot denir. Bu varsayılan yapıcı metodun aldığı herhangi bir parametre yoktur. Varsayılan yapıcı metotla ilgili bilmemiz gerekenler:

- Eğer bir sınıfta herhangi bir yapıcı metot oluşturmuşsak varsayılan yapıcı metot oluşturulmaz.
- Yapıcı metotlar da aşırı yüklenebilir. Örneğin `public Deneme();`, `public Deneme(int a);`, `public Deneme(int a, int b);` ve `public Deneme(int a,int b,int c);` şeklinde üç satır oluşturabilir ve nesnenin oluşturulma şekline göre bu metotlardan yalnızca birinin çalıştırılmasını sağlayabiliriz. Örnek:

```
using System;
class Deneme
{
   Deneme()
   {
      Console.WriteLine(0);
   }
   Deneme(int a)
   {
      Console.WriteLine(a);
   }
   Deneme(int a,int b)
   {
      Console.WriteLine(a+b);
   }
   Deneme(int a,int b,int c)
```

```
    {
        Console.WriteLine(a+b+c);
    }
    static void Main()
    {
        Deneme a=new Deneme(5,6);
    }
}
```

Bu örnekte üçüncü metot çalıştırılacaktır. Bu örneği şöyle değiştirebiliriz.

```
using System;
class Deneme
{
    Deneme(int a,int b,int c)
    {
        Console.WriteLine(a+b+c);
    }
    Deneme():this(0,0,0)
    {
    }
    Deneme(int a):this(a,0,0)
    {
    }
    Deneme(int a,int b):this(a,b,0)
    {
    }
    static void Main()
    {
        Deneme a=new Deneme(5,6);
    }
}
```

Bu örnekte `this` anahtar sözcüğü sayesinde ikinci, üçüncü ve dördüncü yapıcı metotlar içeriğini aynı isimli ve üç parametre alan metottan alıyor. this anahtar sözcüğüyle kullanılan ikinci, üçüncü ve dördüncü yapıcı metotların yaptığı tek iş, birinci yapıcı metoda belirli parametreleri göndermek oluyor.

NOT: this anahtar sözcüğü bu şekilde yalnızca yapıcı metotlarla kullanılabilir.

NOT: Eğer bir sınıfta parametre alan bir veya daha fazla yapıcı metot varsa ve parametre almayan yapıcı metot yoksa -bu durumda varsayılan yapıcı metot oluşturulmayacağı için- `Sinif nesne=new Sinif();` gibi bir satırla parametre vermeden nesne oluşturmaya çalışmak hatalıdır. Çünkü `Sinif nesne=new Sinif();` satırı Sinif sınıfında parametre almayan bir yapıcı metot arar, bulamaz ve hata verir. Örnek:

```
class A
{
    public A(int a){}
}
class B
{
```

```
    A a=new A();
}
```

Bu program derleme zamanında hata verir.

Yıkıcı metotlar

Değişkenlerdeki faaliyet alanı kuralları aynen nesnelerde de geçerlidir. Yani `Deneme a=new Deneme();` satırıyla tanımladığımız a nesnesi üzerinden Deneme sınıfının üye elemanlarına, yalnızca `Deneme a=new Deneme();` satırının içinde bulunduğu en iç bloktan erişilebilir. C++ gibi alt seviye programlama dillerinde bir nesnenin faaliyet alanı bittiğinde manuel olarak bunun bilgisayara söylenmesi gerekebiliyordu. Bu söyleme işi ise yıkıcı metotlarla oluyordu. Aksi bir durumda bellekten söz konusu nesneler silinmediği için karışıklıklar çıkabiliyordu. Halbuki C#'ta bizim böyle bir şey yapmamıza gerek yok. C# Garbage Collection (çöp toplama) mekanizması sayesinde gereksiz nesneleri bellekten siliyor. Ancak bunun illaki nesnenin faaliyet alanı sonlanır sonlanmaz yapılacağı garantisi yok. Örnek bir program:

```
using System;
class Deneme
{
    ~Deneme()
    {
        Console.WriteLine("Yıkıcı metot şimdi çalıştırıldı.");
    }
    static void Main()
    {
        {
            Deneme a=new Deneme();
        }
        Console.WriteLine("Vikikitap");
    }
}
```

Gördüğünüz gibi yıkıcı metotlar sınıf adının başına ~ işareti getirilerek oluşturuluyor. Yıkıcı metotlar herhangi bir değer tutamazlar (void de almazlar) ve herhangi bir parametre almazlar. Bu program a nesnesi bellekten silindiği anda yani yıkıcı metot çalıştırıldığı anda ekrana `Yıkıcı metot şimdi çalıştırıldı.` yazacaktır.

Static üye elemanlar

Bu konuya daha önceden değinmiştik. Ancak bu bölümde biraz hatırlatma ve biraz da ek bilgi verme gereksinimi görüyorum.

- Bir nesneye bağlı olarak çalışmayacak üye elemanları static olarak belirtiriz. Örneğin Console sınıfındaki WriteLine() metodu veya Math sınıfındaki PI özelliği static üye elemanlardır.
- Static üye elemanlara nesne oluşturarak ulaşmaya çalışmak hatalıdır.
- Static üye elemanları oluştururken `static` sözcüğünün erişim belirleyici sözcükle sırası önemli değildir. Yani `static public int Topla()` veya `public static int Topla()` kullanımlarının ikisi de doğrudur. Ancak tabii ki geri dönüş tipinin üye eleman adının hemen öncesinde gelmesi gerekir.
- Şimdiye kadar fark ettiğiniz gibi Main metodunu hep static olarak oluşturduk. Eğer static olarak oluşturmasaydık bu metodun çalışabilmesi için içinde bulunduğu sınıf türünden bir nesne oluşturmamız gerekecekti. Bu durumda da Main metodu işlevi ile çelişecekti. Çünkü Main metodu programımızın çalışmaya başladığı yerdir.

- static olarak tanımlanan üye elemanlara çağrı yapıldığı an bu üye elemanlar için dinamik olarak bellekte yer ayrılır. Bu sayede static olarak tanımlanan ama değer atanmayan özelliklerin çağrı yapıldığı an varsayılan değeri tutması sağlanır.
- Bir sınıf nesnesi oluşturulduğunda static üye elemanlar için bellekte ayrı bir yer ayrılmaz.
- Normal metotlar gibi yapıcı metotlar da static olabilirler. örnek:

```
using System;
class Deneme
{
   static Deneme()
   {
      Console.WriteLine("Static metot çağrıldı.");
   }
   Deneme()
   {
      Console.WriteLine("Static olmayan metot çağrıldı.");
   }
   static void Main()
   {
      Deneme a=new Deneme();
      Deneme b=new Deneme();
   }
}
```

Bu program ekrana şunları yazacaktır:

```
Static metot çağrıldı.
Static olmayan metot çağrıldı.
Static olmayan metot çağrıldı.
```

Gördüğünüz gibi bir sınıf türünden bir nesne oluşturuldığında önce static metot sonra (varsa) static olmayan metot çalıştırılıyor. Sonraki nesne oluşturumlarında ise static metot çalıştırılmıyor. Static yapıcı metotlar parametre veya erişim belirleyicisi almazlar. Şimdi biraz kapsamlı bir örnek yapalım:

```
using System;
class Oyuncu
{
   static int Toplam;
   Oyuncu()
   {
      Toplam++;
      Console.WriteLine("Toplam oyuncu: "+Toplam);
   }
   static Oyuncu()
   {
      Console.WriteLine("Oyun başladı");
   }
   ~Oyuncu()
   {
      Console.WriteLine("Bir oyuncu ayrıldı...");
```

```
        Toplam--;
    }
    static void Main()
    {
        {
            Oyuncu ahmet=new Oyuncu();
            Oyuncu osman=new Oyuncu();
        }
        Oyuncu ayse=new Oyuncu();
        Oyuncu mehmet=new Oyuncu();
    }
}
```

Bu programın ekran çıktısı şöyle olacaktır.

```
Oyun başladı
Toplam oyuncu: 1
Toplam oyuncu: 2
Toplam oyuncu: 3
Toplam oyuncu: 4
Bir oyuncu ayrıldı...
Bir oyuncu ayrıldı...
Bir oyuncu ayrıldı...
Bir oyuncu ayrıldı...
```

Daha önce yıkıcı metodun, nesnenin kapsama alanı sonlanır sonlanmaz çalıştırılma garantisinin olmadığını söylemiştim. Bu örnekte ancak program kendini sonlandırırken her bir nesne için yıkıcı metotlar çalıştırılıyor.

Static sınıflar

Eğer bir sınıf sadece static elemanlar içeriyorsa o sınıfı static olarak tanımlayabiliriz. Böylelikle derleyici bize o sınıf türünden bir nesne oluşturmamıza izin vermeyecektir. Açıkçası pratikte pek bir faydası yoktur. Çünkü static olarak tanımladığımız sınıfların üye elemanlarını da ayrıca static olarak tanımlamak zorundayız. Static sınıfların yapıcı metotları olmaz. Dolayısıyla yapıcı metot tanımlamaya çalışırsak derleyici hata verir. Klasik bir static sınıf bildirimi şöyle yapılır:

```
static class SinifAdi
{
...
}
```

NOT: Değişkenler konusunda gördüğümüz `const` anahtar sözcüğü özelliklerle de kullanılabilir. const olarak tanımlanan özellikler aynı zamanda static'tir. Dolayısıyla const özellikleri tekrar static anahtar sözcüğüyle belirtmek hatalıdır. const özellikler static özelliklerin taşıdığı tüm özellikleri taşır.

NOT: const anahtar sözcüğü yalnızca object dışındaki özelliklerle (ve değişkenlerle) kullanılabilir. Dizilerle vs. kullanılamaz. const özellikler tanımlanır tanımlanmaz bir ilk değer verilmelidir.

UYARI: Bir özelliğe değer verirken static olmayan bir özellik ya da metot kullanılamaz (nesne oluşturulsa bile).

UYARI: Bir sınıfın içinde (bir metodun içinde olmadan) metot ve özellik oluşturulmasından başka bir şey yapılamaz. Yapılacak diğer bütün şeyler bir metot bloğunun içinde olmalıdır.

readonly anahtar sözcüğü

readonly anahtar sözcüğü const anahtar sözcüğünün yaptığı işi object türü, diziler, nesneler vs. için yapar. Yani bu ögeleri salt okunur hâle getirir. Örnek:

```
using System;
class Sinif
{
   readonly int[] a={1,5,8};
   readonly object b=5;
   static void Main()
   {
      Sinif n=new Sinif();
      Console.WriteLine(n.a[0]);
      Console.WriteLine(n.b);
   }
}
```

Şimdi bir de salt okunur nesne oluşturalım:

```
using System;
class Sinif
{
   int a=5;
   static readonly Sinif nesne=new Sinif();
   static void Main()
   {
      Console.WriteLine(nesne.a);
   }
}
```

Şimdilik nesnelerin salt okunur olması saçma gelebilir. Çünkü salt okunur nesnelerle ulaştığımız özellikleri değiştirebiliriz. Salt okunur yaptığımız nesnenin kendisidir. Ancak şimdilik buna kafanızı takmanıza gerek yok. readonly ile ilgili önemli bilgiler:

- readonly anahtar sözcüğü bir metot bloğunun içinde kullanılamaz.
- readonly anahtar sözcüğü bir nesne için kullanılacaksa static sözcüğü de kullanılmalıdır.
- readonly ve static sözcüklerinin sırası önemli değildir.
- readonly bir dizi ya da object türü ayrıca static olarak belirtilebilir.
- readonly anahtar sözcüğü constun kullanılabildiği int gibi türlerle de kullanılabilir.
- readonly'nin constun aksine özellikleri staticleştirme özelliği yoktur.
- static sözcüğü de readonly gibi bir metot içinde kullanılamaz.

NOT: Aslında `int a;` satırı int türünden bir nesne bildirimidir ve `int a=new int();` satırı mümkündür. Bu durumda a'ya int türünün varsayılan değeri atanır. Bunlarla ilgili daha fazla bilgiyi ileride göreceğiz.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Operatör aşırı yükleme

Şimdiye kadar kendi sınıflarımızı, metotlarımızı, özelliklerimizi yazdık. Şimdi kendi operatörlerimizi yazmaya sıra geldi. Şimdi içinde saat bilgileri olan bir sınıf tasarlayacağız:

```
using System;
class Zaman
{
   public int Saat;
   public int Dakika;
   public int Saniye;
   public Zaman(int saat, int dakika, int saniye)
   {
       Dakika=dakika+saniye/60;
       Saniye=saniye%60;
       Saat=saat+Dakika/60;
       Dakika=Dakika%60;
   }
}
```

Gördüğünüz gibi Zaman sınıfının Saat, Dakika ve Saniye olmak üzere üç özelliği var ve bu özellikler nesne yaratılırken girilmek zorunda. Dakika ve saniye 60'tan büyük olamayacağı için yapıcı metotta gerekli düzenlemeler yapılıyor. Bu sınıftan bir nesne yarattığımızda söz konusu nesnenin Saat, Dakika ve Saniye olmak üzere üç özelliği olacak. Başka bir nesne yarattığımızda da aynı şekilde bunun da Saat, Dakika ve Saniyesi olacak. Peki Zaman türünden iki nesnenin + operatörüne sokularak bu iki zaman diliminin toplanmasını ister miydiniz? Örneğin `zaman1` ve `zaman2` Zaman türünden iki nesne olsun. `zaman1+zaman2` yazdığımızda karşılıklı dakika ve saniyeler toplansın ve saniyeden dakikaya ve dakikadan saate gerekli aktarımlar yapılsın. İşte şimdi bunu yapacağız:

```
using System;
class Zaman
{
   public int Saat;
   public int Dakika;
   public int Saniye;
   public Zaman(int saat, int dakika, int saniye)
   {
```

```
        Dakika=dakika+saniye/60;
        Saniye=saniye%60;
        Saat=saat+Dakika/60;
        Dakika=Dakika%60;
    }
    public static Zaman operator+(Zaman a, Zaman b)
    {
        int ToplamSaniye=a.Saniye+b.Saniye;
        int ToplamDakika=a.Dakika+b.Dakika;
        int ToplamSaat=a.Saat+b.Saat;
        return new Zaman(ToplamSaat,ToplamDakika,ToplamSaniye);
    }
}
class AnaProgram
{
    static void Main()
    {
       Zaman zaman1=new Zaman(5,59,60);
       Zaman zaman2=new Zaman(1,0,120);
       Zaman zaman3=zaman1+zaman2;

Console.WriteLine("{0}.{1}.{2}",zaman3.Saat,zaman3.Dakika,zaman3.Saniye);
    }
}
```

Bu program ekrana `7.2.0` yazacaktır. Gördüğünüz gibi kendi operatörlerimizi yazmak, daha doğrusu + operatörüne aşırı yükleme yapmak çok da zor değil. Operatör metotlarımızın adı `operator+` mantığında olmalı. Örneğin söz konusu operatörümüz - olsaydı operatör metodumuzun adı `operator-` olacaktı. Operatörümüzün alacağı operandları bu metoda parametre olarak gönderiyoruz. Operatör metodunun tuttuğu değer operatörün operandlarıyla birlikte tuttuğu değeri ifade eder. `return new Zaman(ToplamSaat,ToplamDakika,ToplamSaniye);` satırıyla metodun, dolayısıyla da operatörün verilen parametreler ölçüsünde Zaman türünden bir nesne tutmasını sağladık. Ancak şimdi Zaman türünden bir nesneyle int türünden bir nesneyi toplamamız mümkün değildir. Eğer Zaman türünden bir nesneyle int türünden bir nesne toplanmak istenirse int türünden nesneyi saniye olarak hesaba katabiliriz. Bunun içinse Zaman sınıfına şu operatör metodunu ekleyebiliriz:

```
public static Zaman operator+(Zaman a, int b)
{
  int ToplamSaniye=a.Saniye+b;
  return new Zaman(a.Saat,a.Dakika,ToplamSaniye);
}
```

Gördüğünüz gibi bu kadar basit. Ayrıca şunu hatırlatmak isterim: `return new Zaman(a.Saat,a.Dakika,ToplamSaniye)` satırıyla metodun Zaman türünden verilen parametrelere göre yeni bir nesne oluşturulup bu nesneyi tutmasını sağladık. Dolayısıyla Zaman sınıfının yapıcı metodu çalıştırıldı. Dolayısıyla da bizim tekrar Saniye, Dakika ve Saat arasında aktarım yapmamıza gerek kalmadı. Operatör metotlarıyla ilgili bilmeniz gerekenler:

- Operatör metotlarının geri dönüş tipinin illaki söz konusu sınıf tipinden olmasına gerek yoktur. Örneğin Zaman sınıfındaki ilk toplama operatör metodunu şöyle değiştirebiliriz.

```
public static string operator+(Zaman a, Zaman b)
{
    int ToplamSaniye=a.Saniye+b.Saniye;
    int ToplamDakika=a.Dakika+b.Dakika;
    int ToplamSaat=a.Saat+b.Saat;
    Zaman nesne=new Zaman(ToplamSaat,ToplamDakika,ToplamSaniye);
    return nesne.Saat+"."+nesne.Dakika+"."+nesne.Saniye;
}
```

Burada `Zaman nesne=new Zaman(ToplamSaat,ToplamDakika,ToplamSaniye);` satırını ekleyerek Saat, Dakika, Saniye aktarımlarının yapılmasını sağladık. Bu metoda göre Zaman tipinden a ve b nesneleriyle yapılan a+b ifadesinin sonucu string tipinde olur.

- Operatör metotları static ve public olmalıdır.
- Dört işlem operatörleri (+, -, *, /) herhangi bir koşul olmaksızın aşırı yüklenebilirler.

İlişkisel operatörlerin aşırı yüklenmesi

Şimdi Zaman sınıfına aşağıdaki operatör metotlarını ekleyelim.

```
public static bool operator==(Zaman a, Zaman b)
{
    if(a.Saniye==b.Saniye&&a.Dakika==b.Dakika&&a.Saat==b.Saat)
        return true;
    else
       return false;
}
public static bool operator!=(Zaman a, Zaman b)
{
    return !(a==b);
}
```

İlişkisel operatörlerdeki ana kural ilişkisel operatörlerin birbirlerinin zıtlarının sınıf içinde ve aynı türde olmasının zorunlu olmasıdır. Yani biz burada yukarıdaki metotların yalnızca bir tanesini yazıp bırakamazdık ya da birininin geri dönüş tipini bool, birinin int yapamazdık. Ayrıca `return !(a==b);` satırı son derece mümkündür. Burada daha önce yazılmış bir metodu kullandık. Her iki metot da static olduğu için herhangi bir nesne tanımlamaya gerek yoktur. İlişkisel operatörlerin geri dönüş tipi bütün türlerde olabilmesine rağmen bool olması tavsiye edilir. Diğer ilişkisel operatörler olan <, >, <= ve >= operatörlerini de çift hâlinde olmak şartıyla tıpkı örneğimizdeki gibi aşırı yükleyebiliriz.

true ve false operatörlerinin aşırı yüklenmesi

Şimdiye kadar `true` ve `false`u bool türünden sabitler olarak gördük. Ancak şimdi bu true ve falseu operatörmüş gibi aşırı yükleyebileceğiz. Şimdi şu metotları Zaman sınıfına ekleyin:

```
public static bool operator true(Zaman a)
{
    if(a.Saat>12)
        return true;
    else
       return false;
}
public static bool operator false(Zaman a)
{
   if(a.Saat<=12)
       return true;
   else
       return false;
}
```

Bu metotları asıl programımız içinde şöyle kullanabiliriz:

```
Zaman zaman1=new Zaman(5,59,60);
Zaman zaman2=new Zaman(2,35,40);
if(zaman1)
   Console.WriteLine("Öğleden sonra");
else
   Console.WriteLine("Öğleden önce");
if(zaman2)
   Console.WriteLine("Öğleden sonra");
else
   Console.WriteLine("Öğleden önce");
```

Gördüğünüz gibi derleyici normalde `true` ya da `false` olması gereken bir yerde Zaman türünden bir nesne gördüğü zaman Zaman sınıfındaki `true` ve `false` metotlarını çalıştırıyor ve söz konusu Zaman nesnenisinin true ya da false değerlerden birisini tutmasını sağlıyor. Eğer böyle bir yapı oluşturmak istiyorsak hem true hem de false metotlarının sınıfımız içinde bulunması gerekir. Ayrıca true ve false operatör metotlarının geri dönüş tipi mutlaka bool türünden olmalıdır.

Mantıksal operatörlerin aşırı yüklenmesi

Örnek (sınıfımıza eklenecek):

```
public static bool operator|(Zaman a,Zaman b)
{
   if(a.Saat>12||b.Saat>12)
      return true;
   else
      return false;
}
```

& ve |, ! operatörlerinin aşırı yüklenebilmesi için herhangi bir şart yoktur. (! operatörünün tek operand aldığını unutmayın.) Ancak && ve || operatörlerinin aşırı yüklenebilmesi için söz konusu sınıf için şu şartların sağlanması gerekir:

- & ve | operatörleri aşırı yüklenmiş olmalıdır.
- true ve false operatörleri aşırı yüklenmiş olmalıdır.
- operator& ve operator| operatörlerinin parametreleri ilgili sınıf türünden olmalıdır.
- operator& ve operator| operatörlerinin geri dönüş tipi ilgili sınıf türünden olmalıdır.
- Yukarıdaki şartlar sağlandığı takdirde bizim ayrıca && ve || operatörlerini aşırı yüklememize gerek kalmaz. Yukarıdaki şartlar sağlandığı takdirde sanki && ve || operatörleri aşırı yüklenmiş gibi kodumuzu yazabiliriz. Örnek:

```
using System;
class Sinif
{
   public int Sayi;
   public Sinif(int sayi)
   {
      Sayi=sayi;
   }
   public static bool operator true(Sinif a)
   {
      return true;
   }
   public static bool operator false(Sinif a)
   {
      return false;
   }
   public static Sinif operator&(Sinif a,Sinif b)
   {
      return new Sinif(20);
   }
   public static Sinif operator|(Sinif a,Sinif b)
   {
      return new Sinif(30);
   }
}
class AnaProgram
{
   static void Main()
   {
      Sinif a=new Sinif(50);
      Sinif b=new Sinif(10);
      Console.WriteLine((a||b).Sayi);
      Console.WriteLine((a&&b).Sayi);
   }
}
```

Yukarıdaki programda ekrana alt alta 50 ve 20 yazar. Peki neden 50 ve 20 yazdı? İşte mantığı:

true operatörü	false operatörü	a||b	a&&b
true	false	ilk operand	& operatör metodu
false	true	| operatör metodu	ilk operand
true	true	ilk operand	ilk operand
false	false	| operatör metodu	& operatör metodu

Tablodan da görebileceğiniz gibi örneğimizde true operatörü `true`, false operatörü `false` değer ürettiği için `a||b` ifadesi ilk operand olan a'yı, `a&&b` ifadesi ise & operatör metodunun geri dönüş değerini tutar.

Tür dönüşüm operatörünün aşırı yüklenmesi

Hatırlarsanız C#'ta iki tane tür dönüşüm mekanizması vardı. Bunlardan birincisi bilinçli, ötekisi de bilinçsiz tür dönüşümüydü. Şimdi bu iki olayın kendi sınıflarımız türünden nesneler için de gerçekleşmesini sağlayacağız. Bu iki olayın taslağı şu şekildedir:

```
public static implicit operator HedefTur(KaynakTurdekiNesne)
{
   return HedefTurdenVeri;
}
```

Bu bilinçsiz tür dönüşümü içindi. Şimdi de bilinçli tür dönüşümü metodunun taslağı:

```
public static explicit operator HedefTur(KaynakTurdekiNesne)
{
   return HedefTurdenVeri;
}
```

Şimdi bu taslakları örneklendirelim:

```
using System;
class Sinif
{
   public int Sayi;
   public Sinif(int sayi)
   {
      Sayi=sayi;
   }
   public static implicit operator int(Sinif a)
   {
      return a.Sayi;
   }
}
class AnaProgram
{
   static void Main()
   {
      Sinif a=new Sinif(50);
      int b=a;
      Console.WriteLine(b);
```

```
    }
}
```

Şimdi de bilinçli tür dönüşümünü örneklendirelim:

```
using System;
class Sinif
{
   public int Sayi;
   public Sinif(int sayi)
   {
      Sayi=sayi;
   }
   public static explicit operator int(Sinif a)
   {
      return a.Sayi;
   }
}
class AnaProgram
{
   static void Main()
   {
      Sinif a=new Sinif(50);
      int b=(int)a;
      Console.WriteLine(b);
   }
}
```

Gördüğünüz gibi bilinçli ve bilinçsiz tür dönüşümü operatör metotlarını yazarken tek değişen `implicit` ve `explicit` anahtar sözcükleri. Bir sınıf içinde parametre ve geri dönüş tipi aynı olan explicit ve implicit metotlar aynı anda bildirilemez. Eğer implicit metodu bildirmişsek explicit (bilinçli) tür dönüşümüne de izin vermiş oluruz. İlk örneğimizde (implicit) a nesnesini int türünden değişkenlerin kullanılabildiği her yerde kullanabiliriz. İkinci örneğimizde ise aynı şeyi tür dönüştürme operatörünü kullanarak yapabiliriz. implicit ve explicit metotlarla yalnızca Sinif türünden başka türlere ya da başka türlerden Sinif türüne dönüşüm yapabiliriz. Yani örneğin byte ile int arasındaki dönüşüm mekanizmasına müdahale edemeyiz.

BİLGİLENDİRME: Şimdi konunun başında gördüğümüz Zaman sınıfını düşünün. Artık implicit metot sayesinde Zaman türünden bir nesneye `a="12.45.34";` şeklinde değer atayabilirsiniz. Bunun için Zaman sınıfının implicit metodu içinde gerekli parçalamaları yapar ve sonucu bir Zaman nesnesinde tutarsınız. Gördüğünüz gibi adeta kendi sabitlerimizi yazabiliyoruz. Bunun gibi daha birçok şey hayal gücünüze kalmış.

NOT: Artık bu öğrendiğimiz implicit metot sayesinde Sinif türünden bir nesneyi `new` anahtar sözcüğünü kullanmadan direkt `Sinif a=23;` gibi bir şey yazarak oluşturabiliriz. Ancak bunun için tabii ki sınıfımız içinde int türünden Sinif türüne implicit dönüşüm metodunu oluşturmamız gerekir.

NOT: Bildiğiniz gibi C#'ta bazı türler arasında bilinçsiz tür dönüşümü olur. Örneğin C#'ta byte türünden inte bilinçsiz tür dönüşümü mümkündür. Ancak biz programımızı yazarken bunları kafamıza takmamıza gerek yok. Yani Sinif türünden int'e dönüşüm yapan metot oluşturmuşsak Sinif türünden byte'a dönüşüm yapan metot oluşturamayız diye bir kural yoktur. İstediğimiz gibi tür dönüşüm metotlarını yazabiliriz. Yeter ki bir türden bir türe dönüşüm yapan birden fazla metot oluşturmayalım ve dönüşümün taraflarından bir tanesi Sinif türünden olsun.

Operatör aşırı yüklemeyle ilgili son notlar

Atama operatörünü (=) aşırı yükleyemeyiz. Çünkü zaten gerek yoktur. Biz atama operatörünü aşırı yükleyelim veya yüklemeyelim zaten kullanabiliriz. İşlemli atama operatörlerinde ise şöyle bir kural vardır: Örneğin + operatörünü aşırı yüklemişsek += operatörünü de kullanabiliriz. Bu durum bütün işlemli atama operatörlerinde geçerlidir. Hiçbir operatörün öncelik sırasını değiştiremeyeceğimiz gibi temel veri türleri (string, int, vb.) arasındaki operatörlere de müdahale edemeyiz. Bu konuda işlenenler dışında hiçbir operatörü aşırı yükleyemeyiz. Örneğin dizilerin elemanlarına erişmek için kullanılan `[]` operatörünü, yeni bir nesneye bellekte yer açmak için kullanılan `new` operatörünü, `?:` operatörünü, vb. aşırı yükleyemeyiz.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

İndeksleyiciler

Daha önceden de bildiğiniz gibi tanımladığımız diziler System.Array sınıfı türünden bir nesnedir. Bu sayede tanımladığımız diziler üzerinden System.Array sınıfının üye elemanlarına erişebiliriz. Örneğin System.Array sınıfına ait olan `Length` özelliği dizimizin kaç elemandan oluştuğunu tutar. `Sort()` metodu ise dizimizin elemanlarını sıralar. Hatırlarsanız dizilerin elemanlarına [] operatörüyle erişiyorduk. Yani Array sınıfı türünden bir nesnenin sonuna [] koyuyorduk ve bunun belirli bir anlamı vardı. Bunu sağlayan System.Array sınıfındaki bir indeksleyeciden başka bir şey değildir. Artık biz de Sinif türünden yarattığımız a nesnesini `a[5]` şeklinde yazabileceğiz ve bunun belirli bir anlamı olacak. İndeksleyiciler tıpkı diziler gibi tek boyutlu indeksleyiciler ve çok boyutlu indeksleyiciler olmak üzere ikiye ayrılır.

Tek boyutlu indeksleyiciler

Örnek:

```
using System;
class Sinif
{
   public int Sayi;
   public int this[int indeks]
   {
      get
      {
         return Sayi;
      }
      set
```

```
        {
            Sayi=value;
        }
    }
}
class AnaProgram
{
    static void Main()
    {
        Sinif a=new Sinif();
        a[5]=30;
        Console.WriteLine(a[5]);
    }
}
```

Gördüğünüz gibi indeksleyiciler sahte özelliklere oldukça benziyor. Ancak ciddi farkları da var. İndeksleyici tanımlarken parametre normal parantez yerine köşeli parantezler arasına yazılıyor. Esas programda [ve] arasına girilen veri indeksleyiciye parametre olarak gönderiliyor. İndeksleyicilere özel bir ad verilmiyor, bunun yerine `this` anahtar sözcüğü kullanılıyor. Ayrıca şunu hatırlatmak isterim: `a[5]=30;` satırıyla `a[5]`'in değil a'nın Sayi özelliği değiştiriliyor. Yani esas programı şöyle değiştirsek:

```
class AnaProgram
{
    static void Main()
    {
        Sinif a=new Sinif();
        a[5]=30;
        a[1]=2;
        Console.WriteLine(a[5]);
        Console.WriteLine(a[1]);
    }
}
```

Bu programda ekrana alta alta iki kez 2 yazar. Çünkü hem `a[5]=30;` hem de `a[1]=2;` satırları a nesnesinin `Sayi` özelliğini değiştirir. Şimdi programımızı şöyle değiştirelim:

```
using System;
class Sinif
{
    public int Sayi;
    public int this[int indeks]
    {
        get
        {
            return Sayi;
        }
        set
        {
            if(indeks>0)
```

```
            Sayi+=value;
         else if(indeks<0)
            Sayi-=value;
         else
            Sayi=Sayi;
      }
   }
}
class AnaProgram
{
   static void Main()
   {
      Sinif a=new Sinif();
      a[5]=45;
      a[-10]=23;
      a[100]=87;
      a[-80]=100;
      Console.WriteLine(a[0]);
   }
}
```

Bu program ekrana 9 yazar. İndeksleyicinin set bloğunda indeksleyiciye hangi indeksle ulaşıldığı kontrol ediliyor. Eğer indeksleyici 0'dan büyükse esas programda o indeksleyiciye atanan değer a nesnesinin Sayi özelliğine ekleniyor. Eğer indeksleyici 0'dan küçükse esas programda o indeksleyiciye atanan değer a nesnesinin Sayi özelliğinden çıkarılıyor. Eğer indeksleyici 0'sa Sayi'nin değeri değiştirilmiyor.

NOT: İndeksleyiciler de aşırı yüklenebilir. Bunu ise parametre türünü farklı yaparak sağlarız.
NOT: İndeksleyicilerin parametre ve geri dönüş tipi herhangi bir tip olabilir. int olması şart değildir.

Çok boyutlu indeksleyiciler

Çok boyutlu indeksleyicilerin tek boyutlu indeksleyicilerden tek farkı indeksleyicinin aldığı parametre sayısıdır. Örnek:

```
using System;
class Sinif
{
   private int Sayi;
   public int this[int indeks1,int indeks2]
   {
      get
      {
         return indeks1+indeks2+Sayi;
      }
      set
      {
         Sayi=indeks1*indeks2+value;
      }
   }
}
```

```
class AnaProgram
{
   static void Main()
   {
      Sinif a=new Sinif();
      a[5,4]=45;
      Console.WriteLine(a[-6,12]);
   }
}
```

Bu programda `a[5,4]=45;` satırında 5*4+45 işlemi yapılır ve sonuç Sayi özelliğine atanır. `Console.WriteLine(a[-6,12]);` satırında ise -6+12+65 işlemi yapılır ve sonuç ekrana yazılır. (sonuç: 71)

NOT: Aslında indeksleyicilerin en yaygın kullanımı sınıfımız içindeki bir dizinin elemanlarına direkt nesne ve [] operatörünü kullanarak erişmektir. Bunu mantığınızı kullanarak yapabilirsiniz. Anlatma gereksinimi görmüyorum.
NOT: İndeksleyicilerin set ve get blokları sahte özelliklerdeki set ve get bloklarının tüm özelliklerini taşırlar. Örneğin set bloğunu koymayarak nesnenin indekslerine değer atanmasını engelleyebiliriz.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Yapılar

Sizce int, double, byte birer nedir? "Sınıftır!" dediğinizi duyar gibiyim. Yaklaştınız ancak tam olarak sınıf değil. Temel veri türleri (int, double, vb.) birer yapıdır. int türünden tanımladığımız bir a değişkeni/özelliği de bir yapı nesnesidir. Peki sınıf ile yapı arasında ne fark var? Aşağıdaki kodu inceleyiniz.

```
int a=5;
Sinif b=23;
```

Buraya kadar ikisi de birbirinin aynısı gibi. Bu koddan Sinif sınıfının içinde bir implicit metot olduğunu ve atanan değerin sınıf içindeki int türünden bir özelliğe atandığını çıkarabiliriz. Şimdi bu kodu şöyle geliştirelim:

```
int a=5;
Sinif b=23;
int c=a;
Sinif d=b;
a=10;
b=100;
Console.WriteLine("{0}-{1}-{2}-{3}",a,b.Sayi,c,d.Sayi);
```

Bu kodda `int c=a;` satırında a'nın tuttuğu değer c'ye atanır. `Sinif d=b;` satırında ise d nesnesi b nesnesinin bellekteki adresini bulur ve o adresteki değeri tutar. Gördüğünüz gibi sınıf nesneleri birbirine adres gösterme yoluyla atanırken yapı nesneleri birbirine değer kopyalama yoluyla atanır. Bu kodda b nesnesinde yapılan bir değişiklik d nesnesini, d nesnesinde yapılan bir değişiklik de b nesnesini etkileyecektir. Ancak c nesnesinde yapılan bir değişiklik a nesnesini, a nesnesinde yapılan bir değişiklik de c nesnesini etkilemeyecektir. Ancak tabii ki sınıf nesnelerinin ikisini de başka bir değer atayarak ya da `new` anahtar sözcüğü ile oluşturmuş, sonra atama işlemini yapmışsak bu sefer değer kopyalama yoluyla birbirlerine atanırlar.

Kendimiz de int, float gibi yapılar oluşturabiliriz. Yapı yaratmak sınıf yaratmaya oldukça benzer, yalnızca `class` anahtar sözcüğü yerine `struct` anahtar sözcüğü kullanılır. Örnek:

```
struct YapiAdi
{
   public int Ozellik1;
   public string Ozellik2;
}
```

Yapılarla ilgili bilmeniz gerekenler:

- Tıpkı sınıflar gibi yapılarda da `new` operatörüyle nesne yaratılabilir. `new` anahtar sözcüğünü kullandığımızda yapının varsayılan yapıcı metodu veya varsa kendi tanımladığımız yapıcı metot çalıştırılır.
- Tıpkı sınıflardaki gibi `YapiAdi nesne;` yazarak bir yapı nesnesini kullanılabilir hâle getiremeyiz. Ya değer atamamız ya da `new` anahtar sözcüğünü kullanmamız gerekir.
- Bir yapı türündeki nesne bir metoda parametre olarak gönderildiğinde metodun içinde nesnenin değiştirilmesi esas nesneyi değiştirmez. Ancak sınıf nesnelerinde değiştirir.
- Tıpkı sınıflardaki gibi yapılarda da birden fazla yapıcı metot yaratabiliriz. Ancak yapılarda parametre almayan bir yapıcı metot bildiremeyiz. Hatırlarsanız sınıflarda bildirebiliyorduk.
- Bir yapının yapıcı metodu bildirildiğinde yapıcı metodun içinde yapının bütün özelliklerine ilk değer verilmesi gerekir.
- Yapı nesnelerinin faaliyet alanı sonlandığında otomatik olarak bellekten silinirler. Hatırlarsanız sınıf nesnelerinin faaliyet alanı sonlandığında garbage collection (çöp toplama) mekanizması devreye giriyordu. Ancak garbage

collection mekanizmasının nesnenin faaliyet alanı sonlanır sonlanmaz devreye girme garantisi yoktu. Özetle sınıf nesnelerinin aksine yapı nesnelerinin tam olarak ne zaman bellekten silindiğini anlayabiliriz.

- Yapılarda yıkıcı metot yaratılması yasaklanmıştır.
- Sınıflarda olduğu gibi yapılarda da sahte özellik (get ve set bloklu özellik) ve indeksleyici kullanabiliriz.
- Yapı nesneleriyle ilgili işlemler, sınıf nesneleriyle ilgili işlemlerden daha hızlı gerçekleşir.
- Tıpkı sınıflardaki gibi normal metotlar da bir yapı içerisinde bildirilebilir. Ayrıca üye elemanlar static olabilir.
- Tıpkı sınıflardaki gibi bir yapı içerisindeki özellikler const ya da readonly olabilir.
- Yapılardaki özelliklere ilk değer verilemez.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Enum sabitleri

Enum sabitleri sayesinde bazı özel sözcüklerin bazı tam sayıları temsil etmesini sağlayabiliriz. En genel enum bildirimi şu şekilde yapılır:

```
enum AD : tur{SOZCUK1,SOZCUK2}
```

Bu bildirimde enum sabitinin adı `AD`dır. `tur` ise sözcüklerin hangi türden sabitleri temsil edeceğini belirtir. Parantez içindeki sözcükler ise herhangi bir sabiti temsil etmesini istediğimiz sözcüklerdir. Parantez içine en fazla türün kapasitesi kadar sözcük girilebilir. Örneğin `tur` yerine byte yazmışsak en fazla 255 tane sözcük girebiliriz.

```
enum AD{SOZCUK1,SOZCUK2}
```

Burada bir tür belirtmediğimiz için otomatik olarak tür int olur. Enum sabitlerinin türünü byte, sbyte, short, ushort, int, uint, long ve ulong türlerinden biri yapabiliriz. Şimdi enum sabitleriyle ilgili bir örnek yapalım. Örneğimizde bir metot kendisine ilk parametre olarak gönderilen dizinin elemanlarını ekrana yazsın. Metodun ikinci parametresinde de dizinin elemanlarının ekrana nasıl yazılacağı ayarlansın. Eğer ikinci parametre `YANINA` olarak girilirse dizinin elemanları ekrana yan yana yazılsın. Eğer ikinci parametre `ALTINA` olarak girilirse dizinin elemanları ekrana alt alta yazılsın:

```
using System;
enum BICIM : byte{YANINA,ALTINA}
class Program
{
   static void DiziYaz(Array dizi,BICIM b)
   {
      if(b==BICIM.YANINA)
      {
```

```
            foreach(object o in dizi)
               Console.Write(o.ToString()+" ");
            Console.WriteLine();
         }
         else
            foreach(object o in dizi)
               Console.WriteLine(o);
      }
      static void Main()
      {
         int[] a=new int[10];
         DiziYaz(a,BICIM.YANINA);
         DiziYaz(a,BICIM.ALTINA);
      }
}
```

Şimdi başka bir örnek:

```
using System;
enum not:byte{basarisiz,gecmez,gecer,orta,iyi,pekiyi}
class Program
{
   static void Main()
   {
      Console.Write("Lütfen notunuzu giriniz: ");
      not a=(not)Convert.ToByte(Console.ReadLine());
      Console.WriteLine(a);
   }
}
```

Bu program ekrana kullanıcı 0 girerse `basarisiz`, 1 girerse `gecmez`, 2 girerse `gecer`, 3 girerse `orta`, 4 girerse `iyi` ve 5 girerse `pekiyi` yazmaktadır. Eğer kullanıcı 5'ten büyük bir sayı girerse sayıyı yazacaktır. Enum sözcüklerinin verilen türe göre temsil ettikleri bir tam sayı vardır. Biz enum sabitini tanımlarken herhangi bir ek düzenleme yapmadığımız için ilk sözcük 0'ı temsil eder. Diğer sözcükler de birer artarak kendi sayılarını temsil eder. Yani aşağıdaki tablo söz konusudur:

Sözcük	Temsil ettiği sayı
basarisiz	0
gecmez	1
gecer	2
orta	3
iyi	4
pekiyi	5

Ayrıca gördüğünüz gibi programımızda byte türünü not türüne (enum sabitine) bilinçli olarak dönüştürdük. Bilinçsiz olarak dönüştüremezdik. Ayrıca direkt olarak stringten nota dönüşüm yapamazdık. Yalnızca enum sınıfı ile string ve object dışındaki temel veri türleri arasında dönüşüm yapabiliriz. Şimdi başka bir program:

```
using System;
enum not:byte{basarisiz,basarili}
class Program
{
   static void Main()
   {
      not a=not.basarili;
      if(a==(not)1)
         Console.Write("Başarılısınız");
      else
         Console.Write("Başarısızsınız");
   }
}
```

Burada da int türü not türüne bilinçli olarak dönüştürüldü. Başka bir örnek:

```
using System;
enum not:byte{basarisiz,basarili}
class Program
{
   static void Main()
   {
      Console.Write(not.basarisiz);
   }
}
```

Bu programda ekrana `basarisiz` yazılır. Yani enum sözcükleri direkt kullanılmak istendiğinde yalnızca sözcük kullanılıyor. Başka bir örnek:

```
using System;
enum not:byte{basarisiz,basarili}
class Program
{
   static void Main()
   {
      not a=not.basarisiz;
      byte b=(byte)a;
      Console.Write(b);
   }
}
```

Burada da not türü byte'a bilinçli olarak dönüştürüldü.

Enum sıra numaralarını değiştirme

`enum not:byte{basarisiz,basarili}` satırında sözcükler 0'dan başlayarak birer birer artarak bir tam sayıyı temsil eder. Ancak hangi sözcüğün hangi tam sayıyı temsil ettiğini kendimiz belirtebiliriz. Örnekler:

```
enum not:byte{basarisiz=6,basarili=10}
```

Bu örnekte basarisiz 6'yı, basarili 10'u temsil eder. Bir tam sayı birden fazla sözcüğe verilebilir.

```
enum not:byte{basarisiz=6,basarili}
```

Bu örnekte basarisiz 6'yı, basarili 7'yi temsil eder.

```
enum not:byte{basarisiz,gecmez=5,gecer,orta,iyi=2,pekiyi}
```

Bu örnekte ise şu temsiller söz konusudur:

Sözcük	Temsil ettiği sayı
basarisiz	0
gecmez	5
gecer	6
orta	7
iyi	2
pekiyi	3

```
enum not:byte{basarisiz=-21,basarili}
```

Bu örnekte `basarisiz` -21'i `basarili` -20'yi temsil eder.

System.Enum sınıfı

System.Enum sınıfı enum sabitleriyle kullanılabilecek çeşitli üye elemanlara sahiptir. Burada yalnızca `GetNames()` metodu tanıtılacaktır. `GetNames()` metodu bir enum sabitinin tüm sözcüklerini string türündeki bir dizi olarak tutar. Örnek:

```
using System;
enum Gunler:byte
{
   PAZARTESI,
   SALI,
   CARSAMBA,
   PERSEMBE,
   CUMA,
   CUMARTESI,
   PAZAR
}
class Sinif
{
   static void Main()
   {
      string[] a=Gunler.GetNames(typeof(Gunler));
      Console.WriteLine(a[0]);
```

```
        //veya
        Console.WriteLine(Gunler.GetNames(typeof(Gunler))[3]);
    }
}
```

Bu örnek ekrana alt alta `PAZARTESI` ve `PERSEMBE` yazar.

NOT: Enum sabitine isim verirken ve enum sözcüklerini yazarken değişken adlandırma kurallarının tamamına uymalıyız.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

İsim alanları

İsim alanları en temel anlamıyla sınıfları organize etmeye (gruplandırmaya) yarayan bir yapıdır.

.Net Framework kütüphanesindeki hazır isim alanlarına erişme

Bildiğiniz üzere şimdiye kadar .Net Framework kütüphanesindeki hazır isim alanlarına erişmek için `using` anahtar sözcüğünü kullandık. Bu anahtar sözcükle ilgili bilmeniz gereken bazı ufak detaylar:

- `using` anahtar sözcüğüyle bir isim alanını varsayılan isim alanımız hâline getiririz. Yani o isim alanındaki sınıflara, yapılara, vb. direkt erişebiliriz. Eğer `using` anahtar sözcüğünü kullanmamış olsaydık söz konusu sınıfı ya da yapıyı kullanmadan önce söz konusu sınıfın ya da yapının bağlı olduğu isim alanını da . operatörüyle eklemek zorunda kalacaktık. Örneğin programımızın başında `using` anahtar sözcüğü ile `System` isim alanını belirtmezsek bu isim alanına bağlı olan `Console` sınıfının içindeki `WriteLine` metodunu `System.Console.WriteLine()` olarak kullanmak zorunda kalacaktık.
- .Net Framework kütüphanesinin en tepesinde System isim alanı ve bunun altında çeşitli görevler için özelleşmiş farklı isim alanları bulunur. Örneğin System.Data isim alanında veri tabanı işlemleri için özelleşmiş çeşitli sınıflara ulaşabiliriz. Bu isim alanının altında yine çeşitli isim alanları da vardır.
- `using` anahtar sözcüğüyle bir isim alanına erişim hakkı elde etmemiz onun altındaki veya üstündeki isim alanlarına da erişim hakkı elde ettiğimiz anlamına gelmez. Her bir isim alanını ayrı ayrı bildirmeliyiz.
- Temel veri türleri (int, double, vs.) aslında .Net Framework kütüphanesindeki birer yapıdan başka bir şey değildir. Bu yapılar System isim alanının altında bulunur. Tür dönüşümü konusunda "değişken türlerinin CTS karşılıkları" adı altında bu yapıları tablolamıştık. Yani aslında int türünden bir değişken tanımlarken System.Int32 yapısından bir nesne tanımlamış oluyoruz. Ancak `int a=34;` gibi bir komutla bir değişken tanımlarken `using` anahtar sözcüğüyle System isim alanını eklememize gerek yoktur. Çünkü C# otomatik olarak temel veri türlerini bizim erişimimize açık tutar. Yani biz `using` anahtar sözcüğüyle System isim alanına erişim hakkı elde etmemiş olsak bile bu isim alanına bağlı olan Int32 yapısını kullanabiliriz. Bu durum yalnızca temel veri türlerinde geçerlidir.

Aşağıdaki programı inceleyiniz:

```
class Sinif
{
   static void Main()
   {
      int a=5; //geçerli tanımlama
      Int32 a=5; //geçersiz tanımlama
   }
}
```

Gördüğünüz gibi C# ikisinde de aynı yapıyı (System.Int32) kullanmamıza rağmen ikincisine izin vermedi. Programın başına `using System;` satırını ekleseydik ikincisine de izin verecekti. Yani C#, değişkenleri C#'a özgü şekilde oluşturmuşsak System isim alanını eklememiz zorunlu değildir. Yapıların gerçek hâlleriyle değişken oluşturmak ise System isim alanının eklenmesini gerektirir.

- Çoğu zaman programımıza `using` anahtar sözcüğüyle birden fazla isim alanı eklememiz gerekir. Bu gibi durumlarda her bir isim alanını `using` anahtar sözcüğüyle ayrı ayrı belirtmeliyiz. Virgül ile ortak bildirim yapamayız.

Kendi isim alanımızı yaratma

Fark ettiyseniz şu ana kadar yazdığımız sınıfları, yapıları vs. herhangi bir isim alanına yazmadık. Bu gibi durumlarda C# söz konusu sınıf ya da yapıları varsayılan isim alanımızda sayar. Yani biz bir isim alanı oluşturmamışsak bile C# hayalî bir isim alanı oluşturur ve bize bu isim alanına direkt erişim hakkı verir. Ancak istersek türlerimizi bir isim alanı içine koyabiliriz. Örnek bir isim alanı bildirimi

```
namespace IsimAlani
{
   ...
}
```

Bir isim alanının içinde yalnızca sınıflar, temsilciler, enum sabitleri, arayüzler ve yapılar direkt olarak bulunabilir. Bundan sonra tüm bunlara kısaca "tür" diyeceğim. Temsilciler ve arayüzleri ileride göreceğiz. Örnek bir program:

```
using System;
namespace BirinciIsimAlani
{
   class Deneme
   {
   }
}
class AnaSinif
{
   static void Main()
   {
      BirinciIsimAlani.Deneme d=new BirinciIsimAlani.Deneme();
   }
}
```

Gördüğünüz gibi bulunduğumuz isim alanından farklı bir isim alanındaki türle ilgili işlemleri ilgili isim alanını da belirterek yapıyoruz.

İsim alanları diğer bloklarda olduğu gibi bloğun kapanmasıyla sonlanmaz. Eğer farklı dosyalardaki farklı türleri aynı isim alanına koyarsak bunlar aynı isim alanında sayılır. Örneğin `Deneme1.cs` dosyası şöyle olsun:

```
using System;
namespace Deneme
{
   class Sinif1
   {
      public Sinif1()
      {
         Console.WriteLine("Sinif1 türünden nesne yaratıldı.");
      }
   }
}
```

`Deneme2.cs` dosyası şöyle olsun:

```
using System;
namespace Deneme
{
   class Sinif2
   {
      public Sinif2()
      {
         Console.WriteLine("Sinif2 türünden nesne yaratıldı.");
      }
   }
}
```

Ve son olarak `Deneme3.cs` dosyamız şöyle olsun:

```
using System;
class Sinif3
{
   static void Main()
   {
      Console.WriteLine("Burası ana program");
      Deneme.Sinif1 a=new Deneme.Sinif1();
      Deneme.Sinif2 b=new Deneme.Sinif2();
   }
}
```

Şimdi bu üç cs dosyasını aynı klasöre koyalım. DOS'un `cd` komutuyla o klasöre geçelim ve `csc Deneme1.cs Deneme2.cs Deneme3.cs` komutunu verelim. Bu komutla C# derleyicisinin bu üç dosyayı tek dosya gibi düşünmesini sağladık. Bunu sınıflar için de yapabiliriz. Örneğin bir şans oyunu programının kullanıcı arayüzüyle ilgili üye elemanların bulunduğu sınıfı ayrı bir dosyaya, arka planla ilgili üye elemanların bulunduğu sınıfı başka bir dosyaya ve son olarak içinde Main metodunun olduğu ve temel komutları içeren son üye elemanların bulunduğu sınıfı da başka bir dosyaya koyarak komut isteminde aynı yöntemi kullanarak derleyebiliriz. Bu tür bir yöntem programımızın karmaşıklığını azaltacaktır. Yine bu tür bir yöntemde yalnızca bir tane Main metodu olmalıdır. Programımız, Main metodunun içinde bulunduğu dosya adında oluşacaktır. Yukarıdaki programa tekrar dönersek,

program ekrana şunları yazar:

```
Burası ana program.
Sinif1 türünden nesne yaratıldı.
Sinif2 türünden nesne yaratıldı.
```

C# bu tür bir durumda Sinif1 ve Sinif2 sınıflarını aynı isim alanında sayar. Şimdi başka bir örnek yapalım:

```
using System;
using Deneme;
namespace Deneme
{
   class Sinif1
   {
      public Sinif1()
      {
         Console.WriteLine("Sinif1 türünden nesne yaratıldı.");
      }
   }
}
class Sinif2
{
   static void Main()
   {
      Sinif1 a=new Sinif1();
   }
}
```

Gördüğünüz gibi programımızın başında `using Deneme;` satırını da kullandığımız için artık bu isim alanındaki türlerle ilgili işlemleri direkt yapabiliyoruz. `using` deyimleri programımızda hiçbir şey yapmadan önce yazılmalıdır. Yani programlarımızın daima ilk komutları `using` deyimleri olmalıdır. Aynı `using` deyiminin birden fazla kez yazılması durumunda program hata vermez, ancak gereksizdir. Örneğin aşağıdaki program hata vermez.

```
using System;
using System;
class Sinif
{
   static void Main()
   {
      Console.WriteLine("Deneme");
   }
}
```

`using` anahtar sözcüğünün bir başka kullanımı da istediğimiz bir bloğun sonunda nesnelerin `Dispose()` metodunu çağırmaktır. Örnek:

```
using System;
class Deneme:IDisposable //Arayüzleri ileride göreceğiz, o yüzden bu
kısma kafa yormanıza gerek yok.
{
```

```
    public void Dispose()
    {
        Console.WriteLine("Dispose() metodu çağrıldı.");
    }
}
class AnaSinif
{
    static void Main()
    {
        Deneme d=new Deneme();
        using(d)
        {
            Console.WriteLine("using bloğu");
        }//d.Dispose() metodu burada çağrılır.
        Console.WriteLine("using bloğu dışı");
    }
}
```

Bu program ekrana sırayla şunları yazar.

```
using bloğu
Dispose() metodu çağrıldı.
using bloğu dışı
```

Gördüğünüz gibi using bloğunun hemen sonunda Dispose() metodu çağrıldı. using bloğunu aşağıdaki şekilde de yazabilirdik:

```
using(Deneme d1=new Deneme(), d2=new Deneme())
{
    Console.WriteLine("using bloğu");
}//d1.Dispose() ve d2.Dispose() metodu burada çağrılır.
```

Bu örnekte d1 ve d2 nesneleri için ayrı ayrı Dispose() metodu çağrılacaktır. Bu using bloğunu aşağıdaki örnekteki gibi yazamazdık:

```
using(Deneme d1=new Deneme,Deneme d2=new Deneme())
{
    Console.WriteLine("using bloğu");
}
```

Yani nesneler aynı türden olmalıdır ve ortak tanımlama yapılmalıdır.

using ile takma isim (alias) verme

Diyelim ki iki tane isim alanında aynı adlı türler var ve biz bu isim alanlarını `using` anahtar sözcüğü ile eklemişiz. Tahmin edersiniz ki bu durumda bu türlerden biriyle işlem yapmak istediğimizde hata oluşur. Çünkü derleyici hangi isim alanındaki türü kullanmak istediğimizi bilemez. İşte bu gibi durumlarda takma isim (alias) kullanılır. Örnek:

```
using System;
using IsimAlani1;
using IsimAlani2;
//Eğer aşağıdaki iki satırı yazmasaydık programımız hata verirdi.
using Sinif1=IsimAlani1.Sinif;
using Sinif2=IsimAlani2.Sinif;
//Bu iki satırla derleyicinin IsimAlani1.Sinif sınıfını Sinif1,
IsimAlani2.Sinif sınıfını Sinif2 olarak tanımasını sağladık.
namespace IsimAlani1
{
   class Sinif
   {
      public Sinif()
      {
         Console.WriteLine("Burası IsimAlani1");
      }
   }
}
namespace IsimAlani2
{
   class Sinif
   {
      public Sinif()
      {
         Console.WriteLine("Burası IsimAlani2");
      }
   }
}
class AnaSinif
{
   static void Main()
   {
      Sinif1 a=new Sinif1();
      Sinif2 b=new Sinif2();
      //Gördüğünüz gibi nesne yaratırken programın başında
belirttiğimiz takma isimleri kullandık.
   }
}
```

NOT: Böyle bir programda asıl isim alanlarının `using` anahtar sözcüğüyle ayrı olarak belirtilmesi şart değildir. Yani programımızda `using IsimAlani1;` ve `using IsimAlani2;` satırları bulunmasa da olurdu.

NOT: Takma ad kullanımı .Net Framework kütüphanesindeki sınıflarda da geçerlidir. Örnek:

```
using K=System.Console;
class AnaSinif
{
   static void Main()
   {
      K.WriteLine("Deneme");
   }
}
```

Elbette ki programımızda Console dışında System isim alanına bağlı bir sınıf kullanacaksak `using System;` satırını programın başına eklemeliyiz. Ancak bu programda gerek yok. Başka bir örnek:

```
using System;
using S=Sinif;
class Sinif
{
   public Sinif()
   {
      Console.WriteLine("Deneme");
   }
}
class Ana
{
   static void Main()
   {
      S a=new S();
   }
}
```

Burada Sinif sınıfı varsayılan hayalî isim alanımızda olduğu için isim alanını belirtmedik.

```
using System;
using K=Console;
class Ana
{
   static void Main()
   {
      K.WriteLine("Deneme");
   }
}
```

Burada System isim alanı varsayılan isim alanımız olmasına rağmen varsayılan hayalî isim alanımız olmadığı için program hata verir.

İç içe geçmiş isim alanları

İsim alanları iç içe geçebilir. Örneğin .Net Framework kütüphanesinde System isim alanı altında veri tabanı işlemleri için özelleşmiş olan System.Data isim alanı bulunur. Kendimiz de iki türlü iç içe geçmiş isim alanları oluşturabiliriz. Birincisi klasik üst isim alanının blokları içinde yeni bir isim alanı tanımlayarak gerçekleşir. İkincisi de aşağıdaki gibi:

```
namespace UstIsimAlani.AltIsimAlani
{
    ...türler...
}
```

Burada UstIsımAlani isim alanına bağlı AltIsimAlani adında bir isim alanı oluşturduk.

Haricî takma isimler

Daha önce, iki isim alanına bağlı aynı isimli türler bulunduğunda ve bu isim alanlarını `using` anahtar sözcüğüyle programımıza eklediğimizde aynı adlı türlerle bir işlem yapmak gerektiğinde hata çıktığını ve bunu önlemek için takma isimler kullandığımızı söylemiştik. Peki diyelim ki iki tane DLL dosyamız var ve bu dosyalarda 100'er tane aynı isimli sınıf var. Farklı olan sadece sınıfların içeriği. Her bir sınıfa ayrı ayrı takma isim mi koyacağız? Tabii ki hayır. Bunun için haricî takma isimleri kullanacağız. Hatta biz bu örneğimizi biraz daha abartalım ve isim alanları da aynı adlı olsun. Artık örneğimize başlayabiliriz. Şimdi dosya1.cs dosyası oluşturun ve içine şu kodları yazın:

```
namespace IsimAlani
{
   public class bir
    {
    }
   public class iki
    {
    }
}
```

Şimdi dosya2.cs dosyası oluşturun ve içine şu kodları yazın:

```
namespace IsimAlani
{
   public class bir
    {
    }
   public class iki
    {
    }
}
```

Bu iki dosyanın içeriği tamamen aynı. Ancak siz sınıfların içeriği farklıymış gibi düşünün. Şimdi her iki dosyayı da kütüphane dosyası (DLL) hâline getirmek için komut satırından

```
csc /t:library dosya1.cs
csc /t:library dosya2.cs
```

komutlarını ayrı ayrı verin. Artık elimizde dosya1.dll ve dosya2.dll adlı iki sınıf kütüphanesi var. Bunlar haricî kütüphanelerdir. Çoğu durumda bütün kütüphaneleri bir exe dosyasına gömmek pek kullanışlı olmayabilir. Çünkü aynı kütüphanelerden farklı exe dosyalarının da faydalanmasını isteyebiliriz. Bu yüzden DLL dosyaları oluştururuz. Ayrıca kodumuzda sınıfları `public` olarak belirttik. Bu sayede bu DLL'lerdeki sınıflara dışarıdan erişimi mümkün kıldık. Konumuza dönecek olursak yukarıdaki örneğimizde iki farklı DLL dosyası tamamen aynı isim alanı ve türleri içeriyor. Şimdi esas programımızı oluşturalım (ismi program.cs olsun):

```
using IsimAlani1;
using IsimAlani2;
class Program
{
   static void Main()
   {
      bir a=new bir();
   }
}
```

Şimdi bu ana programımızı derlememiz gerekiyor. Ancak dikkat etmemiz gereken bir nokta var. Bu ana programın önceki hazırladığımız iki DLL ile ilişkilendirilmesi gerekiyor. Bunun için komut satırında şu komutu veriyoruz:

```
csc /r:dosya1.dll /r:dosya2.dll program.cs
```

Henüz bu program derlenmeyecektir. Çünkü derleyici `bir a=new bir();` satırında hangi DLL'deki `bir` sınıfını kullanacağını bilmemektedir. Programımızın hata vermemesi için program.cs dosyamızı yeni baştan şöyle yazalım:

```
extern alias Dosya1;
extern alias Dosya2;
class Ana
{
   static void Main()
   {
   }
}
```

Şimdi komut satırından derleme işlemini yeni takma isimlere göre şöyle yapmamız gerekiyor.

```
csc /r:Dosya1=dosya1.dll /r:Dosya2=dosya2.dll program.cs
```

Bu derleme işlemi sorunsuz gerçekleşecektir. Burada dosya1.dll dosyasına Dosya1 takma adını, dosya2.dll dosyasına da Dosya2 takma adını verdik. Pratikte pek faydalı olmasa da bir kütüphaneye birden fazla takma ad verebilirsiniz. Bunun için takma adları virgülle ayırırsınız. Şimdi esas programımızdan DLL'lere ulaşmaya sıra geldi.

:: operatörü

:: (iki tane iki nokta yan yana) operatörü ile takma isim verilmiş DLL kütüphanlerindeki tür ve isim alanlarına erişiriz. Şimdi örneklendirelim. Hatırlarsanız esas pprogramımızın Main bloğunda hiçbir kod yoktu. Şimdi program.cs dosyasının Main bloğuna şu kodları ekleyin:

```
Dosya1::IsimAlani.Sinif a=new Dosya1::IsimAlani.Sinif();
Dosya2::IsimAlani.Sinif b=new Dosya1::IsimAlani.Sinif();
```

Aynı şekilde bu tür uzun bir nesne yaratımı yapmak istemiyorsak programın başına

```
using Dosya1::IsimAlani;
using Dosya2::IsimAlani;
```

Ancak tabii ki yukarıdaki gibi bir using bildirimi aynı adlı isim alanlarında farklı adlı türler varsa mantıklıdır. Örneğimizde pek de mantıklı değildir.

global haricî takma ismi

.Net Framework kütüphanesi, kendi oluşturduğumuz kütüphaneler ve takma ad verilmemiş DLL kütüphanelerimiz otomatik olarak `global` takma adını alır. Yani programlar aşağıdaki gibi de yazılabilir.

```
using System;
namespace ia
{
   class Yardimci
   {
      public Yardimci()
      {
         Console.WriteLine("Yardımcı");
      }
   }
}
namespace ana
{
   class Sinif
   {
      static void Main()
      {
         global::ia.Yardimci a=new global::ia.Yardimci();
      }
   }
}
```

Başka bir örnek:

```
class prg
{
   static void Main()
   {
      global::System.Console.WriteLine("Deneme");
   }
```

```
}
```

Ancak az önce işlediğimiz örneği

```
global::IsimAlani.Sinif a=new global::IsimAlani.Sinif();
global::IsimAlani.Sinif b=new global::IsimAlani.Sinif();
```

şeklinde değiştiremezdik. Pek bir işimize yaramasa da bilmeniz de fayda var.

NOT: extern ifadeleri using ifadelerinden önce yazılmalıdır.
NOT: DLL dosyaları direkt türleri içerebileceği gibi isim alanlarını ve isim alanının altında da türleri içerebilir.
NOT: Eğer türler farklı bir DLL dosyasındaysa türleri `public` olarak belirtmek gerekir. Ancak aynı dosyada farklı bir isim alanındaysa, ya da dosyaları birlikte derleme söz konusuysa `public` anahtar sözcüğünü kullanmaya gerek yoktur. Şimdiye kadar öğrendiğimiz bütün türler `public` anahtar sözcüğüyle belirtilebilir.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

System isim alanı

Bu kısımda .Net Framework sınıf kütüphanesinin en tepesinde bulunan System isim alanındaki önemli türleri yakından inceleyeceğiz. Hatırlarsanız bu isim alanındaki rastgele sayı üretmeye yarayan Random sınıfını, matematiksel işlemler yapmamızı sağlayan Math sınıfını ve türler arasında dönüşüm sağlayan Convert sınıfını daha önce incelemiştik. Şimdi ise C#'ta tarih-saat işlemleriyle ilgili üye elemanlar barındıran DateTime ve TimeSpan yapılarını, çöp toplama mekanizmasıyla ilgili üye elemanlar barındıran GC sınıfını, verilerin baytlarıyla ilgili işlemler yapmamıza yarayan BitArray ve Buffer sınıflarını, dizilerle ilgili işlemler yapmamızı sağlayan Array sınıfını ve C#'taki temel tür yapılarını ayrıntılarıyla işleyeceğiz.

Temel tür yapıları

Daha önce söylemiştim, ancak yine de hatırlatmak istiyorum. C#'taki temel veri türleri aslında System isim alanındaki bazı yapılardan başka bir şey değildir. int, float, vb. System isim alanındaki Int32, Single, vb. yapılarına verilen takma isimden başka bir şey değildir. Yani C#'ın programlarımızın başına `using int=System.Int32;` satırını gizlice eklediğini düşünebilirsiniz. Aşağıdaki tabloyu inceleyiniz:

C#'taki takma adı	System isim alanındaki yapı
bool	Boolean
byte	Byte
sbyte	Sbyte
short	Int16
ushort	UInt16
int	Int32
uint	UInt32
long	Int64
ulong	UInt64
float	Single
double	Double
decimal	Decimal
char	Char

Tahmin edersiniz ki bu yapıların üye elemanları da vardır. Önce özellikleri inceleyelim:

MaxValue İlgili veri türünün tutabileceği maksimum değeri ilgili veri türünden tutar. string, object ve bool türü haricindekilerle kullanılabilir. Örnek:

```
Console.WriteLine(UInt64.MaxValue);
```

MinValue İlgili veri türünün tutabileceği minimum değeri ilgili veri türünden tutar. string, object ve bool türü haricindekilerle kullanılabilir. Örnek:

```
Console.WriteLine(UInt64.MinValue);
```

Epsilon Yalnızca Single ve Double ile kullanılabilir. En küçük pozitif değeri ilgili veri türünden tutar. Örnek:

```
Console.WriteLine(Single.Epsilon);
```

NaN Herhangi bir sayıyı temsil etmeyen değer (ilgili türde). Yalnızca Single ve Double ile kullanılabilir. Örnek:

```
Console.WriteLine(Single.NaN);
```

NegativeInfinity Negatif sonsuz (ilgili türde). Yalnızca Single ve Double ile kullanılabilir. Örnek:

```
Console.WriteLine(Single.NegativeInfinity);
```

PositiveInfinity Pozitif sonsuz (ilgili türde). Yalnızca Single ve Double ile kullanılabilir. Örnek:

```
Console.WriteLine(Single.PositiveInfinity);
```

Şimdi de metotlar:

Parse() String türünden veriyi ilgili türe çevirir. String ve object hariç bütün türlerle kullanılabilir. Aslında `Parse()` metodunun yaptığı iş stringin tırnaklarını kaldırmaktır. Stringin tırnakları kaldırılmış hâli hedef türle uyumluysa dönüşüm gerçekleşir. Örneğin string türündeki veri `"true"` ise Boolean türüne dönüşebilir. Ancak Int32 türüne dönüşmez. Benzer şekilde stringin içindeki karakter sayısı bir taneyse char'a dönüşebilir. Aksi bir durumda dönüşmez. Eğer dönüşüm gerçekleşmezse çalışma zamanı hatası alırsınız. Örnek:

```
int a=Int32.Parse("12");
```

CompareTo() Bu metotla, metodu çağıran veriyle parametre kıyaslanır. Metodu çağıran büyükse pozitif, parametre büyükse negatif, eşitlerse 0 değeri döndürür. Geri dönüş tipi inttir. Bütün türlerle kullanılabilir, ancak metodu çağıran object olamaz. Parametredeki verinin türü metodu çağırandan küçük ya da eşit kapasiteli olmalıdır. Bool türüyle bool türü kıyaslanabilir. true 1, false 0'mış gibi ele alınır. String türündeyse iki string aynıysa 0, diğer durumlarda 1 değeri geri döndürür. Örnek:

```
Console.WriteLine(12.CompareTo(100));
```

Equals() Bu metotla, metodu çağıran veriyle parametre kıyaslanır. Eşitlerse true, eşit değillerse false değeri geri döndürür. Bütün türlerle kullanılabilir. Hatasız sonuçlar için parametrenin tipi metodu çağırandan küçük ya da eşit kapasiteli olmalıdır. Uyumsuz türler kıyaslanırsa hata vermez, false değeri geri döndürür. Örnek:

```
Console.WriteLine(12.Equals(100));
```

ToString() Verilerin stringe dönüşmüş hâllerini tutar. Parse'ın tersidir diyebiliriz. Bütün türlerle kullanılabilir. Örnek:

```
string a=12.ToString();
```

IsInfinity() Parametredeki Single ya da Double türündeki veri sonsuz ise true, değilse false değeri üretir. Örnek:

```
Console.WriteLine(Single.IsInfinity(12));
```

IsNaN() Parametredeki Single ya da Double türündeki veri NaN (sayı olmayan) ise true, değilse false değeri üretir. Örnek:

```
Console.WriteLine(Single.IsNaN(12));
```

IsPositiveInfinity() Parametredeki Single ya da Double türündeki veri pozitif sonsuzsa true, değilse false değeri döndürür. Örnek:

```
Console.WriteLine(Single.IsPositiveInfinity(12));
```

IsNegativeInfinity() Parametredeki Single ya da Double türündeki veri negatif sonsuzsa true, değilse false değeri döndürür. Örnek:

```
Console.WriteLine(Single.IsNegativeInfinity(12));
```

GetNumericValue() Yalnızca Char yapısında bulunur. Char tipindeki parametre numerikse o sayıyı döndürür, numerik değilse negatif sayı döndürür. Geri dönüş tipi double'dır. Örnek:

```
Console.WriteLine(Char.GetNumericValue('5'));
```

Şimdi bir metot grubu inceleyelim. Aşağıdaki metotların tamamı yalnızca Char yapısında bulunur, tamamı static'tir ve tamamının iki farklı prototipi vardır. Birinci prototip:

```
Metot(char)
```

Bu metot prototipinde char türünden veri metoda sokulur.

```
Metot(string,int)
```

Bu metot prototipinde string içindeki int'te belirtilen indeks numarasındaki karakter metoda sokulur.

Bu metotların tamamının geri dönüş tipi bool'dur, belirtilen koşul sağlanmışsa true, sağlanmamışsa false değeri döndürür. Şimdi bu metotlar:

Metot	Koşul
IsControl	Karakter kontrol karakteri ise
IsDigit	Karakter bir rakam ise
IsLetter	Karakter bir harf ise
IsLetterOrDigit	Karakter bir harf ya da rakam ise
IsLower	Karakter küçük harf ise
IsNumber	Karakter rakam ise
IsPunctuation	Karakter noktalama işareti ise
IsSeperator	Karakter boşluk gibi ayırıcı ise
IsSurroGate	Karakter Unicode yedek karakteri ise
IsSymbol	Karakter sembol ise
IsUpper	Karakter büyük harf ise
IsWhiteSpace	Karakter tab ya da boşluk karakteri ise

Şimdi bunlara iki örnek verelim:

```
Console.WriteLine(Char.IsUpper('f'));
```

veya

```
Console.WriteLine(Char.IsUpper("deneme",3));
```

Şimdi iki metot inceleyelim. Bu metotların ikisi de yalnızca Char yapısında bulunur. Parametre olarak Char türünden veri alırlar. İkisi de statictir.

ToLower Parametresinden aldığı char'ı küçük harfe dönüştürüp char olarak tutar.
ToUpper Parametresinden aldığı char'ı büyük harfe dönüştürüp char olarak tutar.
Bu iki metot, parametre mantıksal olarak dönüştürülemeyecekse karakteri olduğu gibi tutar. Örneğin ToLower metodu parametre olarak 'h' veya '%' almışsa karakterde herhangi bir değişim yapmadan tutar.

Şimdi iki özellik inceleyelim. Bu iki özellik yalnızca Boolean yapısında bulunur ve False ve True stringlerini üretir. Örnekler:

```
string a=Boolean.FalseString;
string b=Boolean.TrueString;
```

Decimal türü

C# Decimal isminde diğer dillerden farklı bir veri türü barındırır. Değişkenler konusunu hatırlarsanız Decimal türü 16 bayt ile en fazla boyuta sahip olan türdü. Ancak işin enteresan tarafı kapasitesi çok da fazla değildi. Kapasitesi 8 byte olan double, decimaldan kat be kat fazla veri depolayabiliyordu. İşte şimdi decimal türünün sırrını göreceğiz.

Decimal türü System isim alanındaki Decimal yapısıyla temsil edilir. Decimal veri türünün bellekte saklanması diğer türlere nazaran biraz karmaşıktır. Bu yüzden bilgisayarların decimal türüyle işlem yapması biraz daha zordur. Decimal türünden bir nesne çok değişik yollarla tanımlanabilir. Örnekler:

```
Decimal a=new Decimal(20);
```

Gördüğünüz gibi Decimal yapısının bir yapıcı metodu varmış. Bu yapıcı metoda parametre olarak int, uint, long, ulong, float ve double türünden veri verilebilir. Başka bir tanımlama şekli

```
Decimal a=new Decimal(int alt, int orta, int ust, bool işaret, byte
ondalık);
```

buradaki `alt` 128 bitlik (yani 16 baytlık) sayının ilk 32 bitini, `orta` sonraki 32 bitini, `ust` sonraki 32 bitini temsil eder. işaret true olursa sayı negatif, false olursa pozitif olur. Son olarak `ondalık` da sayının ondalık kısmını ayırmak için kullanılan noktanın sağdan kaçıncı basamakta olduğunu, diğer bir değişle sayının ondalıklı kısmının kaç haneli olacağını belirtir. Şimdi bir örnek:

```
Decimal a=new Decimal(128755,0,0,false,2);
```

Bu koda göre a sayısı 1287.55'tir. Şimdi yalnızca Decimal yapısına özgü metotları inceleyelim:

ToByte(), **ToSbyte**, **ToInt16**, **ToUInt16**, **ToInt32**, **ToUInt32**, **ToInt64**, **ToUInt64**, **ToSingle()** ve **ToDouble()** metotlarının tamamı statictir, parametre olarak decimal türünden veri alırlar ve ilgili türdeki veriyi üretirler. Yani tür dönüşümü yaparlar. Bir örnek:

```
int a=Decimal.ToInt32(5m);
```

Şimdi decimal sayılar üzerinde çeşitli işlemler yapan metotlara geçelim:
d1 ve d2 decimal türünden iki veri olmak üzere

```
Decimal.Add(d1,d2); //d1 ve d2'nin toplamını decimal türünden tutar.
Decimal.Divide(d1,d2); //d1'in d2'ye bölümünü decimal türünden tutar.
Decimal.Multiply(d1,d2); //d1 ile d2'nin çarpımını decimal türünden
tutar.
Decimal.Subtract(d1,d2); //d1-d2'nin sonucunu decimal türünden tutar.
(çıkarma işlemi)
Decimal.Remainder(d1,d2); //d1'in d2'ye bölümünden kalanı decimal
türünden tutar. (mod alma işlemi)
Decimal.Floor(d1); //d1'den büyük olmayan en büyük tam sayıyı decimal
```

```
türünden tutar. (aşağı yuvarlama)
Decimal.GetBits(d1);
/*d1 için decimal sayı tanımlarken kullandığımız yapıcı işlevdeki beş
parametreyi int türündeki bir dizi olarak tutar.*/
Decimal.Negate(d1); //d1'in negatifini tutar.
Decimal.Round(d1,sayi);
/*sayi int türünde olmalıdır. Bu metot ile d1'in ondalık kısmındaki
hane sayısı sayi kadar kalır. Yani d1 12.53666 ve sayi
3 ise 12.537 tutulur. Son kaybedilen hane 5 ya da 5'ten büyükse son
kalan hane 1 artılır. sayi 0 olursa d1 tam sayıya
yuvarlanmış olur.*/
Decimal.Truncate(d1); //d1'in tam sayı kısmını tutar. Herhangi bir
yuvarlama yapılmaz.
```

Decimal sayıların normal sayılardan en önemli farklarından biri de şudur:

```
double a=2.00d;
decimal b=2.00m;
Console.WriteLine(a);
Console.WriteLine(b);
```

Bu program ekrana alt alta 2 ve 2,00 yazar. Yani normal sayılarda ondalık kısımdaki etkisiz sıfırlar sonuca yansımazken, decimal sayılarda yansır. Decimal sayılar daha çok ekonomi, bankacılık, para, muhasebe, vb. ile ilgili yazılım geliştirirken kullanılır.

System.Array sınıfı

Array sınıfındaki birçok metodu ve özelliği daha önce görmüştük. Burada yalnızca hatırlatma yapacağım ve birkaç yeni üye eleman göreceksiniz.

Özellikler

IsFixedSize Dizinin eleman sayısı sabitse true, değilse false değeri verir. Henüz eleman sayısı sürekli değişen dizileri görmediğimiz için sonucu daima true olacaktır. Örnek (`dizi` bir dizi olmak üzere):

```
Console.WriteLine(dizi.IsFixedSize);
```

IsReadOnly Dizideki elemanlar salt-okunur ise true, değilse false değeri verir. Henüz salt-okunur dizileri görmediğimiz için bunun sonucu da daima false olacaktır Örnek.

```
Console.WriteLine(dizi.IsReadOnly);
```

Length Dizideki eleman sayısını verir (int). Örnek:

```
Console.WriteLine(dizi.Length);
```

Rank Dizinin kaç boyutlu olduğunu verir (int). Örnek:

```
Console.WriteLine(dizi.Rank);
```

Metotlar

BinarySearch, **Clear**, **Copy**, **CopyTo**, **GetLength**, **GetValue**, **Reverse**, **SetValue**, **Sort** ve **CreateInstance** metotlarını diziler konusunda ayrıntılı olarak görmüştük. Burada henüz görmediğimiz bir metodu tanıtmak istiyorum. IndexOf metodu bir elemanın bir dizi içinde ilk görüldüğü indeksi verir. Örnek:

```
int[] dizi={1,6,9,23,5};
Console.WriteLine(Array.IndexOf(dizi,5));
```

Bu kod ekrana 4 yazar.

Tarih ve zaman işlemleri

Hatırlarsanız önceki konularımızın birinde Zaman sınıfı oluşturmuş ve iki zaman nesnesiyle çeşitli işlemler yapmıştık. Aslında şimdi göreceğimiz `DateTime` ve `TimeSpan` yapıları da benzer işleve sahip. DateTime yapısı belirli bir zamanı tutmaya yararken, TimeSpan ise zamanın miktarıyla ilgilenir. Şimdi bir örnek yapalım:

```
Console.WriteLine("En küçük: "+DateTime.MinValue);
Console.WriteLine("En büyük: "+DateTime.MaxValue);
```

Bu kod ekrana

```
En küçük: 01.01.0001 00:00:00
En büyük: 31.12.9999 23:59:59
```

çıktısını verecektir. Gördüğünüz gibi DateTime yapısına ait olan MinValue ve MaxValue özellikleri -tıpkı temel veri türlerinde olduğu gibi- DateTime yapısının tutabileceği en büyük ve en küçük verileri veriyor. Şimdi başka önemli özelliklere bakalım:
Now şimdiki saati, **Today** bugünü DateTime türünden verir. Örnek:

```
Console.WriteLine(DateTime.Now);
Console.WriteLine(DateTime.Today);
```

Bu kodun ekran çıktısı şuna benzer:

```
25.12.2008 16:52:25
25.12.2008 00:00:00
```

Aynı işlemler DateTime yapısı türünden nesne oluşturularak da yapılabilir. Örnek:

```
DateTime Tarih=new DateTime();
DateTime Zaman=new DateTime();
Tarih=DateTime.Today;
Zaman=DateTime.Now;
Console.WriteLine(Tarih);
Console.WriteLine(Zaman);
```

Şimdi de DateTime yapısına ait diğer güzel özellikleri inceleyelim, bunlar static değildir, yani DateTime türündeki nesnelerle erişilir
Date DateTime nesnesine ilişkin saat dışındaki bilgiyi verir. (DateTime)
Month DateTime nesnesinin ay bilgisini verir. (int)
Day DateTime nesnesinin gün bilgisini verir. (int)
Year DateTime nesnesinin yıl bilgisini verir. (int)

DayOfWeek DateTime nesnesinin haftanın kaçıncı günü olduğunu verir. (DayOfWeek enum sabiti)
DayOfYear DateTime nesnesinin yılın kaçıncı gününe denk geldiğini verir. (int)
TimeOfDay Geçerli gün için 00:00:00'dan itibaren ne kadar zaman geçtiğini TimeSpan nesnesi olarak verir. (TimeSpan)
Hour DateTime nesnesinin saat bilgisini verir. (int)
Minute DateTime nesnesinin dakika bilgisini verir. (int)
Second DateTime nesnesinin saniye bilgisini verir. (int)
Milisecond DateTime nesnesinin milisaniye bilgisini verir. (1 saniye=1000 milisaniye)(int)
Ticks x saniyenin 10 milyonda biri olmak üzere, DateTime nesnesindeki tarih ile 1 Ocak 0001 00:00:00 arasındaki x sayısını verir. (long)

Bir DateTime nesnesini şu yöntemlerle oluşturabiliriz.

```
DateTime nesne=new DateTime(long tick_sayısı);
DateTime nesne=new DateTime(int yıl,int ay,int gün);
DateTime nesne=new DateTime(int yıl,int ay,int gün,int saat,int
dakika,int saniye);
DateTime nesne=new DateTime(int yıl,int ay,int gün,int saat,int
dakika,int saniye,int milisaniye);
```

Gördüğünüz gibi DateTime yapısının aşırı yüklenmiş birçok yapıcı metodu varmış. Şimdi bir örnek yapalım.

```
DateTime a=new DateTime(2008,12,25);
Console.WriteLine(a.Day);
```

Bu kod ekrana 25 yazar. Gelelim TimeSpan yapısına. TimeSpan yapısı belirli bir tarih tutmaktan ziyade zamanın miktarı ile ilgilenen bir yapıdır. İki DateTime nesnesini birbirinden çıkarırsak sonuçta bir TimeSpan nesnesi oluşur. Örnek:

```
DateTime Gun1=new DateTime(2008,5,5);
DateTime Gun2=new DateTime(2008,12,3);
TimeSpan fark=Gun2-Gun1;
Console.WriteLine(fark.Days);
```

Bu program ekrana 212 yazar. Evet gerçekten de iki tarih arasında 212 gün fark vardır. Ayrıca gördüğünüz gibi TimeSpan yapısının Days adında static olmayan bir özelliği varmış. Bunun gibi Hours, Minutes, Seconds ve Milliseconds özellikleri de vardır. Bu özelliklerin tamamı ilgili TimeSpan nesnesinin ilgili kısmını döndürür. Tamamının geri dönüş tipi inttir. Yine TimeSpan yapısının TotalDays, TotalHours, TotalMilliseconds, TotalMinutes ve TotalSeconds özellikleri ise ilgili TimeSpan nesnesini ilgili cins türünden döndürür. Yani cinsler arasında dönüşüm yapılır. Bunların geri dönüş tipi ise double'dır.

```
TimeSpan GunSayisi=new TimeSpan(45,0,0,0);
DateTime nesne=DateTime.Today+GunSayisi;
Console.WriteLine(nesne);
```

Bu kod bugünden 45 gün sonrasını ekrana yazar. Başka bir örnek

```
TimeSpan DakikaSayisi=new TimeSpan(0,0,23,0);
DateTime nesne=DateTime.Now+DakikaSayisi;
Console.WriteLine(nesne);
```

Bu kod ise şimdiden 23 dk. sonrasını ekrana yazar. <, >, >=, <=, ==, != operatörlerini iki DateTime nesnesini ya da iki TimeSpan nesnesini kıyaslamak için kullanabiliriz. Ayrıca DateTime yapısına ait olan AddDays() metodu

DateTime türünden bir nesneye gün ekletip tutmak için kullanılır. Örnek:

```
DateTime a=new DateTime(2008,1,1);
Console.WriteLine(a.AddDays(65));
```

Bu program ekrana `06.03.2008 00:00:00` yazar. Ancak burada tabii ki a nesnesinin değeri değişmez. AddDays() metodunun geri dönüş tipi DateTime'dır.

BitConverter sınıfı

Alt seviye programlama yaparken genellikle verilerin kendileriyle değil, baytlarıyla ilgileniriz. İşte BitConverter sınıfı bu işlemleri yaparken işimize yarayacak üye elemanları içerir. BitConverter sınıfında bir tane özellik vardır. Bu özellik `IsLittleEndian` özelliğidir. Sistemimizin verileri hangi mimariye göre sakladığını kontrol eder. Bilgisayarlar verileri iki çeşit mimariye göre saklarlar.

1. Düşük anlamlı byte ilk sırada
2. Yüksek anlamlı byte ilk sırada

Eğer düşük anlamlı byte ilk sıradaysa bu mimariye "Little Endian", yüksek anlamlı byte ilk sıradaysa bu mimariye de "Big Endian" denir. IsLittleEndian özelliği mimari "Little Endian" ise true, "Big Endian"sa false değeri üretmektedir. Intel ve AMD'nin bütün işlemcileri Little Endian mimarisine göre çalışmaktadır. Yani bu durumda ev bilgisayarlarının %90'ının bu mimariden olduğunu söyleyebiliriz. Little Endian sistemlerde verilerin belleğe önce düşük anlamlı baytları, sonra yüksek anlamlı baytları yazılır. Bellekten okuma yapılırken de aynı sıra gözetilir. Big Endian sistemlerde ise bu durumun tam tersi söz konusudur. Aşağıdaki kodla sisteminizin hangi mimariyle çalıştığını öğrenebilirsiniz.

```
if(BitConverter.IsLittleEndian)
   Console.WriteLine("Little Endian");
else
   Console.WriteLine("Big Endian");
```

Gördüğünüz gibi IsLittleEndian static bir metot. Gelelim BitConverter sınıfının en önemli metoduna. GetBytes() metodu bool, char, int, long, double, short, uint, ulong, ushort türünden verilerin bytelarını byte türünden bir dizi olarak tutar. Örnek:

```
int a=258; //00000000 00000000 00000001 00000010
byte[] dizi=BitConverter.GetBytes(a);
foreach(byte b in dizi)
{
   Console.WriteLine(b);
}
```

Benim bilgisayarımda bu program ekrana alt alta 2, 1, 0 ve 0 yazdı. Sizin bilgisayarınızda sonuç, bilgisayarınızın mimarisine göre benimkiyle aynı veya ters sırada olabilir. 10'un onluk sistemdeki karşılığı 2, 1'in ise 1'dir. Diğer iki 0'sa int türünün bellekte kapladığı diğer iki baytı temsil eder. int 32 bitlik (4 baytlık) bir veri tipidir. Eğer a değişkeninin türü short (16 bit-2 bayt) olsaydı ekrana sadece 2 ve 1 yazılacaktı. BitConverter sınıfına ait static bir metot grubu vardır. Bu metot grubu herhangi bir byte dizisini temel veri türlerine dönüştürür, yani GetBytes() metodunun tersini yaparlar. statictirler, iki parametre alırlar. Örnek olarak GetBytes() metodunun sonucunu kullanalım:

```
byte[] dizi={2,1,0,0};
Console.WriteLine(BitConverter.ToInt32(dizi,0));
```

Bu kod ekrana -mimariniz Little Endian ise- 258 yazar. Burada dizi dizisini 0. indeksten itibaren inte dönüştürüp ekrana yazdırdık. BitConverter sınıfının ToInt32'ye benzer ToBoolean(), ToChar(), ToInt16(), ToInt32(), ToInt64(), ToSingle(), ToDouble(), ToUInt16(), ToUInt32() ve ToString() metotları da vardır. Eğer dizi stringe dönüştürülürse bu örneğimizde ekrana `02-01-00-00` yazar. Bütün bu metotlar bilgisayarınızın mimarisine göre çalışırlar. Başka bir örnek verelim:

```
byte[] dizi={2,1};
Console.WriteLine(BitConverter.ToInt32(dizi,0));
```

Bu program çalışma zamanı hatası verir. Çünkü 32 bitlik (4 bayt) Int32 türüne eksik bir dizi verilmiştir. Bu dizi Int32 türüne dönüştürülemez. Başka bir örnek:

```
byte[] dizi={2,1};
Console.WriteLine(BitConverter.ToInt16(dizi,0));
```

Bu kod hata vermez. Çünkü Int16 türünün kapasitesi 16 bittir (2 bayt). Dolayısıyla da Int16 türüne iki elemanlı bir byte dizisi verilmesi yeterlidir. Başka bir örnek:

```
byte[] dizi={2,1,5,8};
Console.WriteLine(BitConverter.ToInt16(dizi,0));
```

Bu program hata vermez. Yalnızca ilk iki bayt (0. ve 1. indis) hesaplamaya girer ve sonuç Little Endian mimarisinde 258 çıkar.

Buffer sınıfı

Buffer sınıfıyla tür bilgisinden bağımsız işlemler yaparız. Buffer sınıfı verilerin değerlerine değil baytlarına bakar.

BlockCopy() metodu

Prototipi şöyledir:

```
static void BlockCopy(Array kaynak, int kaynak_indeks, Array Hedef, int
 Hedef_indeks, int adet)
```

Bu prototip `kaynak` dizisinin `kaynak_indeks` numaralı indeksinden itibaren `adet` kadar elemanını `Hedef` dizisine `Hedef_indeks` numaralı indeksten itibaren kopyalar. Ancak kopyalama türden bağımsız, bayt bayt yapılır. Örnek:

```
byte[] kaynak={1,2,3,1}; //00000001 , 00000010 , 00000011 , 00000001
short[] hedef=new short[4]; //0000000000000000 , 0000000000000000 ,
0000000000000000 , 0000000000000000
Buffer.BlockCopy(kaynak,0,hedef,0,4);
/*hedef dizisinin yeni hâli: 0000001000000001 , 0000000100000011 ,
0000000000000000 , 0000000000000000*/
foreach(short a in hedef)
  Console.WriteLine(a);
```

Bu program, mimarimiz Little Endian ise ekrana alt alta 513, 259, 0 ve 0 yazacaktır. Burada derleyici her elemanı bellekte 1 bayt yer kaplayan kaynak dizisinin elemanlarını her elemanı bellekte 2 bayt kaplayan hedef dizisine olduğu gibi kopyaladı. Tahmin edersiniz ki kaynak dizisinin iki elemanı hedef dizisinin bir elemanına tekabül eder ve iki kaynak elemanı birleşip bir hedef elemanı oluşturur. Benim sistemim Little Endian olduğu için derleyici burada karşılaştığı ilk baytı düşük anlamlı bayta, ikinci baytı da yüksek anlamlı bayta kopyaladı.

ByteLength() metodu

Kendisine parametre olarak verilen bir dizideki toplam bayt sayısını bulur. Örneğin kendisine parametre olarak gönderilen 3 elemanlı short türünden bir dizi sonucu 6 sayısı gönderir. Geri dönüş tipi inttir. Örnek:

```
short[] dizi=new short[4];
Console.WriteLine(Buffer.ByteLength(dizi));
```

GetByte() metodu

Prototipi şu şekildedir:

```
static byte GetByte(Array dizi,int a)
```

dizi dizisinin a. baytını verir. Örnek:

```
byte[] dizi={0,3,2,1,4};
Console.WriteLine(Buffer.GetByte(dizi,3));
```

Bu kod 1 değerini döndürür. Eğer dizinin tipi byte değil de short olsaydı işler değişirdi. Çünkü o zaman hem sıfırla beslenen baytlar da hesaba katılırdı ve hem de bilgisayarımızın mimarisi sonucu etkilerdi. Bu durumu örneklendirelim:

```
short[] dizi={0,3,2,1,4};
// 0000000000000000 0000000000000011 0000000000000010 0000000000000001
0000000000000100
Console.WriteLine(Buffer.GetByte(dizi,4));
```

Bu kod Little Endian mimarisinde ekrana 2 yazar. Mimari Big Endian olsaydı ekrana 0 yazacaktı. Çünkü Little Endian mimarisinde verilerin önce düşük anlamlı baytı okunur/yazılır. Big Endian mimarisinde ise tam tersi söz konusudur.

SetByte() metodu

Prototipi şu şekildedir:

```
static void SetByte(Array dizi,int a,byte deger)
```

a. baytı deger olarak değiştirir. Örnek:

```
byte[] dizi={2,1,4};
Buffer.SetByte(dizi,1,5);
Console.WriteLine(dizi[1]); //Ekrana 5 yazılır.
```

Yine eğer dizinin türü byte değil de int olsaydı işler değişirdi.

GC (Garbage Collection) sınıfı

Garbage Collection mekanizmasının normalde otomatik olarak yaptığı işleri bu sınıftaki üye elemanlar sayesinde manuel olarak da yapabiliriz. Şimdi üye elemanları inceleyelim:

```
GC.Collect();
```

bu metot Garbage Collection mekanizmasının devreye girmesini, dolayısıyla da eğer bellekte gereksiz nesne varsa silinmesini sağlar. Programın herhangi bir anında toplam tahsis edilmiş bellek miktarını byte cinsinden görmek içinse GC sınıfının GetTotalMemory(bool) metodu kullanılır. Eğer parametre true olarak girilirse bellek miktarı hesaplanmadan önce GC mekanizması başlatılarak bellekteki gereksiz nesneler silinir. Eğer parametre false olarak girilirse direkt bellek miktarı hesaplanır. Örnek:

```
Console.WriteLine(GC.GetTotalMemory(true));
```

Bu kitabın diğer sayfaları

- C# hakkında temel bilgiler
- İlk programımız
- Değişkenler
- Tür dönüşümü
- Yorum ekleme
- Operatörler
- Akış kontrol mekanizmaları
- Rastgele sayı üretme
- Diziler
- Metotlar
- Sınıflar
- Operatör aşırı yükleme
- İndeksleyiciler
- Yapılar
- Enum sabitleri
- İsim alanları
- System isim alanı
- Temel I/O işlemleri
- Temel string işlemleri
- Kalıtım
- Arayüzler
- Partial (kısmi) tipler
- İstisnai durum yakalama mekanizması
- Temsilciler
- Olaylar
- Önişlemci komutları
- Göstericiler
- Assembly kavramı
- Yansıma
- Nitelikler
- Örnekler
- Şablon tipler
- Koleksiyonlar
- yield
- Veri tabanı işlemleri
- XML işlemleri
- Form tabanlı uygulamalar
- Visual Studio.NET
- Çok kanallı uygulamalar
- Linux'ta C# kullanımı
- Kaynakça
- Giriş sayfası
- **Ayrıca Bakınız:**

  ASP.NET

O işlemleri

I/O İngilizce Input/Output'un kısaltmasıdır, Türkçeye girdi/çıktı olarak çevirebiliriz. Ancak I/O bilgisayar ve programlama dünyasında artık bir terim hüviyetine kavuştuğu için I/O olarak kullanmayı tercih ediyorum. Dosya kaydetme, ekrana ya da yazıcıya bilgi yazdırma, klavyeden bilgi girişi birer I/O işlemleridir. I/O işlemlerini System.IO isim alanının altındaki çeşitli sınıflarla yaparız. Şimdi isterseniz lafı fazla uzatmadan dosya ve klasör işlemleriyle ilgili sınıflara geçelim.

Directory sınıfı

Directory sınıfının hiçbir özelliği yoktur, System.IO altında bulunur, sadece static metotlar içerir.

DirectoryInfo CreateDirectory(string adres)

`adres` ile belirtilen adreste bir klasör oluşturur ve bu klasör bilgilerini bir DirectoryInfo nesnesi olarak tutar. Programımızın çalıştığı klasörde bir klasör oluşturmak için sadece klasörün adını yazmak yeterlidir. Örnekler:

```
Directory.CreateDirectory(@"C:\WINDOWS\deneme");
```

Bu kod C:\WINDOWS altında deneme isimli bir klasör oluşturur.

```
Directory.CreateDirectory("deneme");
```

Bu kod programın çalıştığı klasörde deneme isimli bir klasör oluşturur.

```
Directory.CreateDirectory(@"..\deneme");
```

Bu kod programın çalıştığı klasörün bir üst klasöründe deneme isimli bir klasör oluşturur.

```
Directory.CreateDirectory(@"..\..\deneme");
```

Bu kod programın çalıştığı klasörün iki üst klasöründe deneme isimli bir klasör oluşturur. `..` sayıları bu şekilde artırılabilir. Bu tür bir adres belirtme şekli bütün diğer metotlarda da geçerlidir. Ayrıca bu ve diğer bütün metotlarda da adres diye tarif ettiğimiz veriye dosya/klasörün adı da dâhildir.

void Delete(string adres)

Belirtilen adresteki boş klasörü silmek için kullanılır. Başka bir kullanımı daha vardır.

```
void Delete(string adres,bool a)
```

Bu metot ile eğer a true ise belirtilen adresteki klasör, içindeki bütün dosya ve klasörlerle birlikte silinir.

bool Exists(string adres)

Belirtilen adresteki klasörün olup olmadığını bool cinsinden tutar. Klasör varsa true, yoksa false döndürür.

string GetCurrentDirectory()

Çalışan programın hangi klasörde olduğunu verir. Örneğin Windows'taysak `C:\WINDOWS`'u tutar.

string[] GetDirectories(string adres)

Belirtilen adresteki bütün klasörleri adresleriyle birlikte bir string dizisi olarak tutar.

string GetDirectoryRoot(string adres)

Belirtilen adresteki klasörün kök dizin bilgisini verir. Örneğin adres `C:\Program Files\CONEXANT\CNXT_MODEM_HDAUDIO_SprtHD5m` ise `C:\` değerini döndürür.

string[] GetFiles(string adres)

Belirtilen adresteki dosyaları adresleriyle birlikte string dizisi olarak tutar. Bu ve benzer metotlarda liste İngilizce alfabetik sırasına göredir. GetFiles() metodunun bir prototipi daha vardır:

```
string[] GetFiles(string adres,string dosya)
```

Adresteki dosya(lar) adresleriyle birlikte string dizisi olarak tutulur. Dosya isminde joker karakterleri (*, ?) kullanılabilir.

string[] GetFileSystemEntries(string adres)

Belirtilen adresteki bütün dosya ve klasörleri adresleriyle birlikte bir string dizisi olarak tutar.

DateTime GetLastAccessTime(string adres)

Belirtilen adresteki dosya ya da klasöre en son ne zaman erişildiğini DateTime türünden tutar.

DateTime GetLastWriteTime(string adres)

Belirtilen adresteki dosya ya da klasörün en son ne zaman değiştirildiğini DateTime türünden tutar.

DateTime GetCreationTime(string adres)

Belirtilen adresteki dosya ya da klasörün ne zaman oluşturulduğunu DateTime türünden tutar.

string[] GetLogicalDrives()

Bilgisayardaki bütün sürücüleri bir string dizisi olarak tutar. Bu sürücülere her türlü sabit disk, CD-ROM sürücü, flash disk vb. dâhildir.

DirectoryInfo GetParent(string adres)

Belirtilen adresin bir üst klasörünü DirectoryInfo nesnesi olarak döndürür.

void Move(string kaynak_adres,string hedef_adres)

Dosya ve klasörleri bir konumdan başka bir konuma taşır. Örnekler:

```
Directory.Move(@"C:\Documents and Settings\Bekir Oflaz\Belgelerim\Örnek
 dosya.rtf",@"C:\Web\deneme.rtf");
```

Bu kod `C:\Documents and Settings\Bekir Oflaz\Belgelerim` altındaki `Örnek dosya.rtf` dosyasını `C:\Web` konumuna `deneme.rtf` adıyla taşır.

```
Directory.Move(@"C:\Web",@"C:\Documents and Settings\Bekir
Oflaz\Belgelerim\internet");
```

Bu kod sürücü klasöründeki Web klasörünü tüm içeriğiyle birlikte `C:\Documents and Settings\Bekir Oflaz\Belgelerim` klasörüne `internet` adıyla taşır. Eğer hedef klasörde aynı adlı dosya ya da klasör varsa çalışma zamanı hatası alırsınız.

void SetLastAccessTime(string adres, DateTime zaman)

Belirtilen adresteki dosya ya da klasörün en son erişim zamanını zaman olarak değiştirir.

void SetLastWriteTime(string adres, DateTime zaman)

Belirtilen adresteki dosya ya da klasörün en son değiştirilme zamanını zaman olarak değiştirir.

void SetCreationTime(string adres, DateTime zaman)

Belirtilen adresteki dosya ya da klasörün oluşturulma zamanını zaman olarak değiştirir.

void SetCurrentDirectory(string adres)

Programın çalıştığı klasörü belirtilen adres ile değiştirir.

File sınıfı

File sınıfındaki birçok metot Directory sınıfında da vardır, tek farkları aynı görevleri dosyalar için yerine getirmeleridir. Bu metotlar şunlardır: Exists(), Delete(), GetCreationTime(), GetLastAccessTime(), GetLastWriteTime(), Move(), SetCreationTime(), SetLastAccessTime(), SetLastWriteTime(). Şimdi File sınıfının Directory sınıfında olmayan metotlarına sıra geldi.

StreamWriter AppendText(string adres)

Adreste belirtilen dosya için daha sonra göreceğimiz bir StreamWriter nesnesi oluşturur.

void Copy(string kaynak,string hedef)

Kaynakta belirtilen dosya hedefe kopyalanır. Kopyalamada bir isim çakışması söz konusuysa, yani sizin kopyalama yapmak istediğiniz klasörde zaten aynı isimli bir dosya varsa çalışma zamanı hatası alırsınız. Bunu önlemek içinse;

```
void Copy(string kaynak,string hedef,bool a)
```

Burada a true olursa eğer hedef klasörde aynı adlı dosya varsa üstüne yazılır. a false yapılırsa iki parametreli hâlinden farkı kalmaz.

FileStream Create(string adres,int tampon)

Belirtilen adresteki dosya oluştururlur ve dosyaya ilişkin FileStream nesnesi döndürülür. tampon yazılmazsa yani sadece bir parametre yazılırsa varsayılan tampon miktarı kullanılır.

StreamWriter CreateText(string adres)

Belirtilen adreste üzerine yazmak için bir text dosyası oluşturulur ve dosyaya ilişkin StreamWriter nesnesi döndürülür.

FileAttributes GetAttributes(string adres)

Belirtilen adresteki dosya ya da klasörün FileAttributes enumu cinsinden özniteliği döndürülür. FileAttributes enum sabiti şu sözcükleri içerir: Archive, Compressed, Device, Directory, Encrypted, Hidden, Normal,

NotContentIndexed, Offline, ReadOnly, ReparsePoint, SparseFile, System, Temporary. Bir dosya ya da klasörün birden fazla özniteliği olabilir. Bu durumda öznitelikler virgülle ayrılır.

Open

Üç farklı aşırı yüklenmiş çeşidi vardır. Bunlar:

```
FileStream Open(string adres, FileMode a)
FileStream Open(string adres, FileMode a, FileAccess b)
FileStream Open(string adres, FileMode a, FileAccess b,FileShare c)
```

Open() metodu belirtilen adresteki dosyayı açar ve dosyaya ilişkin FileStream nesnesini döndürür. FileMode, FileAccess ve FileShare System.IO isim alanında bulunan enumlardır ve dosyanın ne şekilde açılacağını ve üzerinde ne şekilde işlem yapılacağını belirtirler.

FileMode enumunda bulunan sözcükler

Append Açılan dosyanın sonuna ekleme yapmak için kullanılır. Eğer dosya bulunmazsa oluşturulur.
Create Yeni dosya oluşturmak için kullanılır. Zaten dosya varsa üzerine yazılır.
CreateNew Yeni dosya oluşturmak için kullanılır, belirtilen dosya mevcutsa çalışma zamanı hatası verdirir.
Open Dosyayı açmak için kullanılır.
OpenOrCreate Belirtilen dosya varsa açılır, yoksa yenisi oluşturulur.
Truncate Belirtilen dosya açılır ve içi tamamen silinir.

FileAccess enumunda bulunan sözcükler

Read Dosya okumak için kullanılır.
ReadWrite Dosya okunmak ve yazılmak üzere açılır.
Write Dosya sadece yazılmak için açılır.

FileShare enumunda bulunan sözcükler

Inheritable Dosyanın child (yavru) prosesler tarafından türetilebilmesini sağlar.
None Dosyanın aynı anda başka prosesler tarafından açılmasını engeller.
Read Dosyanın aynı anda başka proseslerce de açılabilmesini sağlar.
ReadWrite Dosyanın aynı anda başka proseslerce de açılıp, okunup, yazılabilmesini sağlar.
Write Dosyaya aynı anda başka proseslerce yazılabilmesini sağlar.

FileStream OpenRead(string adres)

Belirtilen dosyayı yalnızca okumak için açar ve dosyaya ilişkin FileStream nesnesini döndürür.

StreamReader OpenText(string adres)

Belirtilen dosyayı yalnızca text modunda okumak için açar ve dosyaya ilişkin StreamReader nesnesini döndürür.

FileStream OpenWrite(string adres)

Belirtilen dosyayı yazma modunda açar ve dosyaya ilişkin FileStream nesnesini döndürür.

DirectoryInfo sınıfı

DirectoryInfo sınıfı Directory sınıfının aksine static olmayan metot ve özellikleri içerir. Önce özellikleri bir örnek üzerinde görelim:

```
using System;
using System.IO;
class DirectoryInfoSinifi
{
   static void Main()
   {
      string adres=@"C:\WINDOWS";
      DirectoryInfo d=new DirectoryInfo(adres);
      Console.WriteLine("Özellikler: "+d.Attributes);
      Console.WriteLine("Oluşturulma tarihi: "+d.CreationTime);
      Console.WriteLine("Var mı? "+d.Exists);
      Console.WriteLine("Uzantı: "+d.Extension);
      Console.WriteLine("Tam adres: "+d.FullName);
      Console.WriteLine("Son erişim zamanı: "+d.LastAccessTime);
      Console.WriteLine("Son değişiklik zamanı: "+d.LastWriteTime);
      Console.WriteLine("Klasör adı: "+d.Name);
      Console.WriteLine("Bir üst klasör: "+d.Parent);
      Console.WriteLine("Kök dizin: "+d.Root);
   }
}
```

Şimdi de DirectoryInfo sınıfının metotlarına geçelim. Bu metotların tamamı static değildir. Bu metotların çalışması için gereken adres bilgisi, kendisine ulaşılması için kullanılan DirectoryInfo nesnesindedir.

void Create()

Klasör oluşturur.

DirectoryInfo CreateSubdirectory(string adres)

Belirtilen adreste bir alt dizin oluşturur. Örneğin C:\deneme altında \deneme2\deneme3 dizini oluşturmak için şu kodları yazarız.

```
string adres=@"C:\deneme";
DirectoryInfo d=new DirectoryInfo(adres);
d.Create();
DirectoryInfo alt=d.CreateSubdirectory("deneme2");
alt.CreateSubdirectory("deneme3");
```

Gördüğünüz gibi CreateSubdirectory metodu kendisine ulaşılan nesne içinde parametredeki klasörü oluşturuyor ve oluşturduğu klasörü de DirectoryInfo nesnesi olarak döndürüyor.

Delete

İki farklı aşırı yüklenmiş versiyonu vardır.

```
void Delete()
void Delete(bool a)
```

Birincisinde klasör boşsa silinir, ikincisinde a true ise klasör, içindeki her şeyle silinir.

DirectoryInfo[] GetDirectories()

İlgili klasörde bulunan bütün dizinleri bir DirectoryInfo dizisinde tutar.

FileInfo[] GetFiles()

İlgili klasörde bulunan bütün dosyaları bir FileInfo dizisinde tutar.

FileSystemInfo[] GetFileSystemInfos()

İlgili klasördeki bütün dosya ve klasörler bir FileSystemInfo dizisinde tutulur.

void MoveTo(string hedef)

İlgili dizin, içindeki tüm dosya ve klasörlerle beraber hedefe taşınır.

void Refresh()

İlgili klasörün özelliklerini diskten tekrar yükler.

NOT: Bütün taşıma ve kopyalama işlemlerinde kaynak ve hedef aynı sürücüde olmalıdır.

FileInfo sınıfı

Önce özellikleri bir örnek üzerinde görelim. Kodu yazmadan önce kendiniz C:\WINDOWS klasöründe deneme.txt dosyasını oluşturun.

```
using System;
using System.IO;
class FileInfoSinifi
{
   static void Main()
   {
      string adres=@"C:\WINDOWS\deneme.txt";
      FileInfo d=new FileInfo(adres);
      Console.WriteLine("Öznitelikler: "+d.Attributes);
      Console.WriteLine("Oluşturulma tarihi: "+d.CreationTime);
      Console.WriteLine("Var mı? "+d.Exists);
      Console.WriteLine("Uzantı: "+d.Extension);
      Console.WriteLine("Tam adres: "+d.FullName);
      Console.WriteLine("Son erişim zamanı: "+d.LastAccessTime);
      Console.WriteLine("Son değişiklik zamanı: "+d.LastWriteTime);
      Console.WriteLine("Boyut: "+d.Length);
      Console.WriteLine("Klasör adı: "+d.Name);
      Console.WriteLine("Bulunduğu klasör: "+d.DirectoryName);
```

```
    }
}
```

Şimdi metotlara geçelim. FileInfo sınıfı File sınıfındaki AppendText(), Create(), CreateText(), Delete(), Open(), OpenRead(), OpenText() ve OpenWrite() metotlarını içerir. Bunları tekrar anlatmayacağım. File sınıfında olmayan metotlarsa;

CopyTo

İki aşırı yüklenmiş versiyonu vardır:

```
FileInfo CopyTo(string hedef)
FileInfo CopyTo(string hedef,bool a)
```

Birincisinde ilgili dosya hedefe kopyalanır. Hedefte aynı adlı dosya varsa çalışma zamanı hatası alırsınız. İkincisinde a true ise, hedefte aynı adlı dosya varsa üzerine yazılır.

void MoveTo(string hedef)

İlgili dosya hedefe taşınır.

void Refresh()

İlgili dosyanın bilgileri diskten tekrar alınır.

Path sınıfı

Path sınıfı çeşitli işlemler yapan static üye elemanlara sahiptir. Örnek program:

```
using System;
using System.IO;
class PathSinifi
{
   static void Main()
   {
      string adres=@"C:\dizin\deneme.txt";
      Console.WriteLine("Uzantı: "+Path.GetExtension(adres));
      string yeniAdres=Path.ChangeExtension(adres,"jpg");
      Console.WriteLine("Yeni uzantı: "+Path.GetExtension(yeniAdres));
      string adres2=@"C:\klasör";
      Console.WriteLine("Yeni adres: "+Path.Combine(adres,adres2));
      Console.WriteLine("Klasör: "+Path.GetDirectoryName(adres));
      Console.WriteLine("Dosya adı: "+Path.GetFileName(adres));
      Console.WriteLine("Uzantısız dosya adı:
"+Path.GetFileNameWithoutExtension(adres));
      Console.WriteLine("Tam adres: "+Path.GetFullPath(adres));
      Console.WriteLine("Kök dizin: "+Path.GetPathRoot(adres));
      Console.WriteLine("Geçici dosya adı: "+Path.GetTempFileName());
      Console.WriteLine("Geçici dosya dizini: "+Path.GetTempPath());
      Console.WriteLine("Dosya uzantısı var mı?
"+Path.HasExtension(adres));
      Console.WriteLine("Alt dizin ayıracı:
```

```
"+Path.AltDirectorySeparatorChar);
      Console.WriteLine("Dizin ayıracı: "+Path.DirectorySeparatorChar);
      Console.Write("Geçersiz dosya adı karakterleri: ");
      char[] dizi=Path.GetInvalidFileNameChars();
      foreach(char b in dizi)
         Console.Write(b+" ");
      Console.Write("\nGeçersiz adres karakterleri: ");
      char[] dizi2=Path.GetInvalidPathChars();
      foreach(char b in dizi)
         Console.Write(b+" ");
      Console.WriteLine("\nAdres ayırıcı karakter:
"+Path.PathSeparator);
      Console.WriteLine("Kök dizin ayıracı:
"+Path.VolumeSeparatorChar);
   }
}
```

Dosya yazma ve okuma işlemleri

FileStream sınıfı

FileStream sınıfı ile diskteki bir dosya açılır. StreamReader ve StreamWriter sınıflarıyla üzerinde işlem yapılır. Dosyalar üzerinde metin tabanlı ve bayt tabanlı işler yapabiliriz. Bir FileStream nesnesi çok değişik yollarla oluşturulabilir. Örnekler:

```
string adres=@"C:\Program Files\deneme.txt";
FileStream FSnesnesi1=new FileStream(adres,FileMode.OpenOrCreate);
FileStream FSnesnesi2=new
FileStream(adres,FileMode.OpenOrCreate,FileAccess.Write);
FileStream FSnesnesi3=new
FileStream(adres,FileMode.OpenOrCreate,FileAccess.Write,FileShare.None);
FileInfo FInesnesi1=new FileInfo(adres);
FileStream FSnesnesi4=FInesnesi1.OpenRead();
FileInfo FInesnesi2=new FileInfo(adres);
FileStream FSnesnesi5=FInesnesi2.OpenWrite();
FileInfo FInesnesi3=new FileInfo(adres);
FileStream FSnesnesi6=FInesnesi3.Create();
FileInfo FInesnesi4=new FileInfo(adres);
FileStream FSnesnesi7=FInesnesi4.Open(FileMode.OpenOrCreate);
```

Bu nesne yaratımlarındaki FileMode, FileAccess ve FileShare enumlarını önceden görmüştük. Dosyayla ilgili işlemimiz bittiğinde FileStream sınıfının Close() metodu ile FileStream nesnesi tarafından tutulan kaynaklar boşaltılır ve dosyayı başka prosesler kullanabilir hâle gelir. Close() metodunun kullanılışı:

```
FSnesnesi1.Close();
```

FileStream sınıfı ile yazma ve okuma

FileStream sınıfının Read() ve ReadByte() metotları dosya akımından byte düzeyinde veri okumamızı sağlarlar. ReadByte() metodu akımdan okuma yapamadığı zaman geriye -1 değerini döndürür. ReadByte() metodunun prototipi:

```
int ReadByte()
```

Bu metot ile akımdan bir baytlık bilgi okunur ve akımdaki okuma pozisyonu bir artırılır ki tekrar okuma da aynı değer okunmasın. Okunan byte değeri inte dönüştürülüp int olarak tutulur. İkinci metodumuz ise:

```
int Read(byte[] dizi, int baslangic, int adet)
```

Bu metot ile adet kadar bayt akımdan okunur, okunan bu baytlar dizi dizisine baslangic indeksinden itibaren yerleştirilir. Geri dönüş değeri okunan byte sayısıdır. Şimdi komut satırı argümanı olarak adı alınan dosyanın içeriğini ekrana yazan bir program yazalım:

```
using System;
using System.IO;
class DosyaAkimi
{
   static void Main(string[] args)
   {
      string adres=args[0];
      FileStream fs=new FileStream(adres,FileMode.Open);
      int OkunanByte;
      while((OkunanByte=fs.ReadByte())!=-1)
         Console.Write((char)OkunanByte);
   }
}
```

Şimdi de bir dosyaya byte düzeyinde veri yazmak için kullanılan Write() ve WriteByte() metotlarını inceleyelim. Bir dosya akımına bir byte yazmak için

```
void WriteByte(byte veri)
```

metodu kullanılır. Eğer yazma işlemi başarısız olursa çalışma zamanı hatası oluşur. Dosya akımına bir bayt dizisi yazdırmak içinse

```
void Write(byte[] dizi,int baslangic,int adet)
```

metodu kullanılır. Bu metot ile byte dizisinin baslangic indeksinden itibaren adet kadar elemanı akıma yazılır. Akımın konum göstericisi yazılan byte kadar ötelenir.

Dosya akımına yazılan veriler dosya sistemindeki dosyaya hemen aktarılmaz. Dosya akımı tamponlama mekanizması ile çalıştığı için belirli bir miktar veri yazılana kadar dosya güncellenmez. Ancak FileStream sınıfının Flush() metodunu kullanarak istediğimiz anda tamponu boşaltıp dosyayı tampondaki bilgilerle güncelleyebiliriz. Şimdi bir örnek program yazalım. Programımız kullanıcının girdiği yazıları (Console.ReadLine() ile) bir txt dosyasına kaydetsin, dosyanın adı da komut satırı argümanı olarak verilsin.

```
using System;
using System.IO;
class deneme
{
```

```
   static void Main(string[] args)
   {
      string dosya=args[0];
      FileStream d=new
FileStream(dosya,FileMode.CreateNew,FileAccess.Write);
      Console.Write("Dosyanın içeriğini yazın: ");
      string icerik=Console.ReadLine();
      foreach(char a in icerik)
         d.WriteByte((byte)a);
      d.Flush();
   }
}
```

FileStream sınıfının önemli özellikleri:
bool CanRead Akımdan okuma yapılıp yapılamayacağı öğrenilir.
bool CanSeek Akımda konumlandırma yapılıp yapılamayacağı öğrenilir.
bool CanWrite Akıma yazma yapılıp yapılamayacağı öğrenilir.
long Position Akımdaki aktif konum bilgisi öğrenilir.
long Length Akımın bayt olarak büyüklüğü öğrenilir.

FileStream sınıfının önemli metotları:
void Lock(long pozisyon,long uzunluk) Bu metotla akımın pozisyondan itibaren uzunluk kadar alanı başka proseslerin erişimine kapatılır.
long Seek(long ofset,SeekOrigin a) Bu metotla akımın konumu SeekOrigin ile belirtilmiş konumdan ofset byte kadar ötelenir. SeekOrigin enumunun içerdiği sözcükler şunlardır:

- **Begin** Akımın başlangıç noktası
- **Current** Akımın o anda bulunduğu nokta
- **End** Akımın en son noktası

StreamReader sınıfı

StreamReader sınıfı FileStream sınıfının aksine baytlarla değil, metinlerle ilgilenir. Bir StreamReader nesnesinin oluşturulma yöntemleri:

```
string dosya=@"C:\deneme\ornek.txt";
FileStream fs=new FileStream(dosya,FileMode.Open);
StreamReader sr1=new StreamReader(fs);
StreamReader sr2=new StreamReader(dosya);
FileInfo fi=new FileInfo(dosya);
StreamReader sr3=new StreamReader(fi);
```

Diğer sınıflarda olduğu gibi StreamReader nesneleriyle işimiz bittiğinde Close() metodunu kullanarak kaynakların iade edilmesi tavsiye edilir. StreamReader sınıfının önemli metotları:

```
string ReadLine()
```

Akımdan bir satırlık veriyi okur ve string olarak tutar. Eğer veri okunamazsa null değeri tutar. Okunan satırın sonuna "\n" eklenmez.

```
string ReadToEnd()
```

Akımdaki verilerin tamamını string olarak tutar. Okuma işlemi aktif konumdan başlayacaktır. Eğer okuma yapılamazsa boşluk tutar.

```
int Read()
```

Akımdan bir karakterlik bilgi okunur ve int'e dönüştürülür. İşlem başarısız olursa -1 ile geri döner.

```
int Read(char[] dizi,int indeks,int adet)
```

Akımdan adet kadar karakteri dizi[indeks] elemanından itibaren diziye yerleştirir. Yerleştirilen karakter sayısını döndürür.

```
int Peek()
```

Akımdan bir karakterlik bilgi okunur ve bu karakterin inte dönüşmüş hâli ile geri dönülür. İşlem başarısız olursa -1 ile geri döner. En önemli nokta ise konum göstericisinin yerinin değiştirilmemesidir.

Şimdi bir text dosyasının nasıl satır satır okunabileceğini görmek için bir örnek yapalım. Şimdi bir txt dosyası oluşturun ve içine şunları yazın: (adı deneme.txt olsun)

```
Bu bir metin belgesidir.
Bu deneme amaçlı yazılmıştır.
--------------------
Vikikitap
```

Şimdi programımızı yazalım:

```
using System;
using System.IO;
class Deneme
{
   static void Main()
   {
      string dosya="deneme.txt";
      FileStream fs=new FileStream(dosya,FileMode.Open);
      StreamReader sr=new StreamReader(fs);
      string satir;
      while((satir=sr.ReadLine())!=null)
         Console.WriteLine(satir);
      fs.Close();
    }
}
```

StreamWriter sınıfı

StreamReader sınıfı ile dosyalardan text tabanlı veriler okunabiliyordu. StreamWriter sınıfı ise bunun tam tersini yapar. Yani StreamWriter sınıfı ile dosyalara text tabanlı veriler yazılır. Bir StreamWriter nesnesi şu yollarla oluşturulabilir:

```
string dosya=@"C:\dosya.txt";
FileStream fs=new FileStream(dosya,FileMode.Open);
StreamWriter sw1=new StreamWriter(fs);
StreamWriter sw2=new StreamWriter(dosya);
FileInfo fi=new FileInfo(dosya);
```

```
StreamWriter sw3=fi.CreateText();
```

StreamReader sınıfında olduğu gibi StreamWriter sınıfında da Close() metoduyla StreamWriter nesnesine ilişkin kaynaklar iade edilir. StreamWriter sınıfının en önemli metotları:

```
void Write(string str)
```

Bu metotla akıma str yazısı eklenir. Yazının sonuna herhangi bir sonlandırıcı karakter konmaz. Bu metot ile diğer bütün veri türlerinden veri eklemek mümkündür. Örneğin: Write(5), Write(true), Write('c')

```
void WriteLine(string str)
```

Write metoduyla aynı işi yapar. Tek fark eklenen yazının sonuna kendisi "\n" stringini koyar. Ayrıca Write metodundan farklı olarak WriteLine() metodunu parametresiz kullanabiliriz. Bu durumda akıma sadece "\n" eklenir.

```
void Flush()
```

Tampondaki bilgilerin boşaltılmasını ve dosyanın güncellenmesini sağlar.

Ayrıca StreamWriter sınıfının NewLine özelliği ile satır ayıracı olan karakterleri belirleyebiliriz. Varsayılan olarak bu karakter "\n" ve "\r"dir. StreamWriter sınıfının kullanımına bir örnek verelim. Örneğimiz kullanıcının -Console.ReadLine() ile- girdiği yazıları dosyaya kaydetsin. Dosyanın adı komut satırı argümanıyla belirlensin.

```
using System;
using System.IO;
class Deneme
{
   static void Main(string[] args)
   {
      string dosya=args[0];
      FileStream fs=new
FileStream(dosya,FileMode.Append,FileAccess.Write);
      StreamWriter sw=new StreamWriter(fs);
      Console.Write("Yazınızı girin: ");
      string yazi=Console.ReadLine();
      sw.Write(yazi);
      sw.Flush();
   }
}
```

BinaryWriter ve BinaryReader sınıfları

BinaryWriter ve BinaryReader sınıflarının yaptığı iş StreamReader ve StreamWriter sınıflarının yaptığı işin aynısıdır. Ancak BinaryWriter ve BinaryReader sınıflarıyla StreamReader ve StreamWriter sınıflarından farklı olarak her türde veriyi akıma yazdırabiliriz, yani verinin illaki string olma zorunluluğu yoktur. Bu sınıflarda sırasıyla Write(int), Write(char), Write(char[]), Write(byte) ve ReadByte(), ReadChar(), ReadUInt32, ReadString() vb. metotlar bulunmaktadır. Bu metotlar bütün temel veri türleri için bildirilmiştir. Şimdi bir örnek program yazalım. Programla önce çeşitli türlerde verileri bir dosyaya yazalım, sonra bu verileri tekrar okuyalım.

```
using System;
using System.IO;
class Deneme
{
```

```
    static void Main()
    {
        int i=5;
        decimal d=15.54M;
        char c='A';
        string dosya="deneme.txt";
        FileStream fs1=new FileStream(dosya,FileMode.OpenOrCreate);
        BinaryWriter bw=new BinaryWriter(fs1);
        bw.Write(i);
        bw.Write(d);
        bw.Write(c);
        bw.Close();
        FileStream fs2=new FileStream(dosya,FileMode.Open);
        BinaryReader br=new BinaryReader(fs2);
        Console.WriteLine(br.ReadInt32());
        Console.WriteLine(br.ReadDecimal());
        Console.WriteLine(br.ReadChar());
        br.Close();
     }
}
```

NOT: BinaryWriter sınıfıyla oluşturulan bir dosyayı Notepad ile okumaya kalkarsanız anlamsız simgelerle karşılaşırsınız. Çünkü BinaryWriter sınıfı dosyalara text yöntemiyle değil, binary yöntemle kayıt yapar. BinaryReader sınıfı da bu binary kayıtları okur.

Console I/O işlemleri

I/O işlemleri için gerekli olan sınıflardan System.IO isim alanında olmayan tek sınıf Console sınıfıdır. Şimdiye kadar Console sınıfını ekrana bir şeyler yazmak için ya da kullanıcıdan girdi almak için sıklıkla kullandık. Şimdi ise Console sınıfının daha birçok yönlerini göreceğiz.

Konsol I/O işlemleri için önceden tanımlanmış üç tane standart akım mevcuttur. Bu akımlar TextWriter türü olan Console.Out, Console.Error ve TextReader türünden olan Console.In'dir. Konsol ekranına yazdığımız yazılar aslında TextWriter sınıfının metotları ile olmaktadır. Console.WriteLine ise bizim için sadece bir aracılık görevi yapar. Örneğin aşağıdaki her iki satır da eş görevdedir. İkisi de ekrana `merhaba` yazar.

```
Console.Out.WriteLine("merhaba");
Console.WriteLine("merhaba");
```

Yani özetle Out, Console sınıfına bağlı bir özelliktir ve geri dönüş tipi TextWriter'dır ve bu veriyle TextWriter sınıfının static olmayan bir metodu olan WriteLine()'a erişiriz.

Konsol ekranına yazı yazdırmak için Console sınıfının WriteLine() ve Write() metotlarını kullanırız. Bu ikisi arasındaki tek fark WriteLine'ın yazının sonuna "\n" ekleyip, Write'ın eklememesidir.

Konsoldan bilgi almak için Read() ve ReadLine() metotlarını kullanırız. Eğer tamponda herhangi bir veri yoksa Read() metodu kullanıcıdan veri girişi ister ve girilen stringteki ilk karakteri int olarak tutar. Sonraki Read() metotları ise o stringteki diğer karakterleri tutar. Eğer veri okunamazsa -1 değeri tutar. Örnek:

```
Console.Write((char)Console.Read()+" "+(char)Console.Read()+"
"+(char)Console.Read());
```

Bu satırda önce kullanıcıdan veri girişi istenir. Diyelim ki kullanıcı `deneme` girmiş olsun. Bu durumda ekrana

```
d e n
```

yazılır. ReadLine() metodu Read() metodunun aksine ekrandan girdileri string olarak alır. Console.In özelliğini kullanarak da Read() ve ReadLine() metotlarını çağırabiliriz. Ayrıca Console.Out özelliğini kullanarak da Write() ve WriteLine() metotlarını kullanabiliriz. Console.In'den erişebileceğimiz yani TextReader sınıfının diğer metotları:

```
int Peek()
```

Bu metot ile standart girdi akımından bir karakter okunur ancak bu karakter tampondan silinmez. Örnek:

```
Console.Write((char)Console.In.Peek()+" "+(char)Console.In.Peek()+"
"+(char)Console.In.Peek());
```

Burada kullanıcının ekrana `deneme` yazdığını varsayarsak ekrana `d d d` yazılacaktır.

```
int ReadBlock(char[] dizi,int indeks,int adet)
```

Bu metot ile standart girdi akımından adet kadar karakter diziye indeks elemanından itibaren yerleştirilir.

```
string ReadToEnd()
```

Bu metot ile standart girdi akımındaki bütün veriler okunarak tampondan temizlenir. Okunan veriler string nesnesi olarak döndürülür.

Standart akımların yönlendirilmesi

C#'ta bütün I/O işlemleri akımlar (stream) üzerine kuruludur. Standart akımları başka akımlara yönlendirmek mümkündür. Örnek olarak komut satırında bir programı şöyle çalıştırırsak

```
programadi > deneme.txt
```

Bu programın, ekran çıktılarını deneme.txt dosyasına yazacağı anlamına gelir. Bu da standart çıktı olan konsolun başka akıma yönlendirilmesi anlamına gelir. Komut satırından

```
programadi < deneme.txt
```

Burada da Console.ReadLine() ya da Console.Read() yerine deneme.txt dosyasındaki metin kullanılır. C# programlarımız içinde akımları yönlendirmek içinse Console sınıfının şu metotları kullanılır.
`static void SetOut(StreamWriter sw)` veya `static void SetOut(TextWriter tw)`
`static void SetError(StreamWriter sw)` veya `static void SetError(TextWriter tw)`
`static void SetIn(StreamReader sr)` veya `static void SetIn(StreamReader sr)`
Şimdi Console.In akımını SetIn metodu ile bir dosya akımına yönlendirelim. Yani giriş klavyeden değil, dosyadan alınsın.

```
using System;
using System.IO;
class Deneme
{
   static void Main()
   {
      FileStream fs=new FileStream("deneme.txt",FileMode.Open);
      Console.SetIn(new StreamReader(fs));
      Console.WriteLine(Console.ReadLine());
      Console.WriteLine(Console.ReadLine());
   }
```

```
}
```

Programı derleyin, ancak çalıştırmadan önce programla aynı klasörde deneme.txt dosyası oluşturun ve içine iki satırlık yazı yazın. Aynı şekilde Console.WriteLine() metodu da dosyaya yönlendirilebilir. Tabii ki bunun için Console.SetOut metodu kullanılmalıdır. Programın içindeki hata mesajlarını ise SetError metodu ile bir dosyaya yazdırabiliriz. Örnek:

```
using System;
using System.IO;
class Deneme
{
   static void Main()
   {
      Console.WriteLine("Bu bir denemedir.");
      FileStream fs1 = new FileStream("deneme.txt", FileMode.Create);
      TextWriter tw = Console.Out;
      StreamWriter sw = new StreamWriter(fs1);
      Console.SetOut(sw);
      Console.WriteLine("Dosyaya yazma yapıyoruz.");
      Console.SetOut(tw);
      Console.WriteLine("Merhaba dünya");
      sw.Close();
   }
}
```

Burada önce ekrana `Bu bir denemedir.` yazılıyor. Sonra yeni bir FileStream nesnesi oluşturuluyor. Sonra hedefi konsol ekranı olan bir TextWriter nesnesi oluşturuluyor. Sonra ilişiği deneme.txt olan bir StreamWriter nesnesi oluşturuluyor. Sonra çıktı birimi deneme.txt olarak değiştiriliyor. Sonra ekrana -daha doğrusu dosyaya- `Dosyaya yazma yapıyoruz.` yazdırılıyor. Sonra çıktı birimi yeniden ekran yapılıyor. Sonra ekrana `Merhaba dünya` yazdırılıyor. En son da sw akımının kaynakları boşaltılıyor.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Temel string işlemleri

String sınıfı

Yeni string oluşturma

Yeni bir string nesnesi şu yollarla oluşturulabilir.

```
string a="deneme";
char[] dizi={'a','b','c','d'};
String b=new String(dizi);
String c=new String(dizi,1,2); //dizi[1]'den itibaren 2 eleman stringe
atandı.
String d=new String('x',50); //burada 50 tane x'in yan yana geldiği bir
 string oluşuyor. yani d="xxxx..."
```

String metotları

```
static string Concat(params Array stringler)
```

İstenildiği kadar, istenilen türde parametre alır. Aldığı parametreleri birleştirip string olarak tutar.

```
static int Compare(string a,string b)
```

a ve b kıyaslanır, a büyükse pozitif bir sayı, b büyükse negatif bir sayı, eşitlerse 0 döndürür. Büyüklük-küçüklük mantığı sözlükte önde/sonda gelme mantığının aynısıdır. Sonda gelen daha büyük sayılır. Türkçe alfabesine uygundur.

```
static int Compare(string a,string b,bool c)
```

Birinci metotla aynı işi yapar. Tek fark eğer c true ise kıyaslamada büyük/küçük harf ayrımı gözetilmez. Eğer c false ise birinci metottan farkı kalmaz.

```
static int Compare(string a,int indeks1,string b,int indeks2)
```

Birinci metotla aynı mantıkta çalışır. Tek fark kıyaslamada ilk elemanların a[indeks1] ve b[indeks2] sayılmasıdır. Yani stringlerdeki bu elemanlardan önceki elemanlar yok sayılır.

```
static int Compare(string a,int indeks1,string b,int indeks2,bool c)
```

c true yapılırsa az önceki metodun büyük/küçük ayrımı gözetmeyeni kullanılmış olur.

```
int CompareTo(string str)
```

Compare metodunun tek parametre almış ve static olmayan hâlidir. Metodu çağıran veriyle parametre kıyaslanır. Metodu çağırana a, parametreye b dersek `static int Compare(string a,string b)` metodunun yaptığı işin aynısını yapar.

```
int IndexOf(string a)
int IndexOf(char b)
```

Kendisini çağıran string içinde parametredeki veriyi arar. Bulursa bulunduğu indeksi tutar. Eğer aranan birden fazla karaktere sahipse ilk karakterin indeksini tutar. Eğer arananı bulamazsa -1 değeri döndürür. Eğer stringin içinde arananandan birden fazla varsa ilk bulunanın indeksini döndürür.

```
int LastIndexOf(string a)
int LastIndexOf(char b)
```

IndexOf metoduyla aynı işi yapar. Tek fark arananın son bulunduğu yerin indeksini vermesidir. Örneğin aranan 'n' karakteri ise ve string "benim çantam var" ise 2 değil, 8 döndürür.

```
int IndexOfAny(char[] a)
int LastIndexOfAny(char[] b)
```

Birincisi a dizisinin herhangi bir elemanının ilk bulunduğu indeks ile geri döner. İkincisi ise b dizisinin herhangi bir elemanının son bulunduğu indeks ile geri döner. char dizisindeki elemanların hiçbiri bulunamazsa -1 ile geri dönülür.

```
bool StartsWith(string a)
bool EndsWith(string b)
```

Birincisi metodu çağıran string a ile başlıyorsa true, diğer durumlarda false değeri üretir. İkincisi metodu çağıran string b ile bitiyorsa true, diğer durumlarda false değeri üretir.

```
string Trim()
```

metodu çağıran stringin başındaki ve sonudaki boşluklar silinir.

```
string Trim(params char[] dizi)
```

metodu çağıran stringin başındaki ve sonundaki dizi dizisinde bulunan karakterler silinir. Örneğin string `ebebesdefbebe` ise ve dizi 'e' ve 'b' elemanlarından oluşuyorsa `sdef` değeri döndürülür.

```
string PadRight(int toplam)
string PadRight(int uzunluk,char c)
```

Birincisinde metodu çağıran stringin uzunluğu toplam olana kadar sağına boşluk eklenir. İkinci metotta ise aynı işlem boşluk ile değil c karakteri ile yapılır. Örneğin c karakterini '.' yaparak bir kitabın içindekiler bölümünü hazırlayabiliriz. Aynı işlemi stringin soluna yapmak için

```
string PadLeft(int toplam)
string PadLeft(int uzunluk,char c)
```

metotları kullanılır. Bu da sağa yaslı yazılar yazmak için uygundur.

```
string[] Split(params char[] ayirici)
string[] Split(params char[] ayirici,int toplam)
```

Birinci metotta metodu çağıran string ayirici dizisindeki karakter(ler)e göre parçalara ayrılır ve sonuç bir string dizisinde tutulur. İkincisinde ise bu işlem en fazla toplam kez tekrarlanır. Örnek:

```
string a="Ahmet,Mehmet,Osman,Ayşe";
string[] b=a.Split(',');
Console.WriteLine(b[0]);
```

Join() metodu ise Split() metodunun tam tersi şekilde işler.

```
static string Join(string ayirici,string[] yazilar)
static string Join(string ayirici,string[] yazilar,int baslangic,int
toplam)
```

Birincisinde yazilar dizisinin elemanlarının arasına ayirici eklenerek tek bir string hâline getirilir. İkincisinde ise yazilar[baslangic]'ten itibaren toplam kadar eleman ayirici ile birleştirilip tek bir string olarak tutulur.

```
string ToUpper()
```

Kendisini çağıran stringin harflerini büyük yapar ve tutar.

```
string ToLower()
```

Kendisini çağıran stringin harflerini küçük yapar ve tutar.

```
string Remove(int indeks,int adet)
```

Yazıdan indeks nolu karakterden itibaren adet kadar karakteri yazıdan siler ve oluşan bu yeni stringi tutar.

```
string Insert(int indeks,string str)
```

Yazının indeks. elemanından sonrasına str stringini yerleştirir ve tutar.

```
string Replace(char c1,char c2)
```

Yazıda geçen bütün c1 karakterlerini c2 olarak değiştirir ve tutar.

```
string Replace(string s1,string s2)
```

Yazıda geçen bütün s1 yazılarını s2 olarak değiştirir ve tutar.

```
string Substring(int indeks)
```

Metodu çağıran stringin indeks. elemanından sonraki yazıyı tutar.

```
string Substring(int indeks,int toplam)
```

Metodu çağıran yazının indeks. elemanından sonraki toplam karakterlik yazıyı tutar.

Yazıları biçimlendirme

Şimdi isterseniz metotlardan başımızı kaldıralım ve yazıları biçimlendirmeye başlayalım. Biçimlendirme yazıların ekrana veya başka bir akıma istenilene formatta yazılması olayıdır. Ancak öncelikle yazılar üzerinde biçimlendirme yapabilmek için ilgili metodun biçimlendirmeyi destekliyor olması gerekir. Console.WriteLine(), String.Format() ve ToString() metotları biçimlendirmeyi destekler. İlk olarak Console.WriteLine() metodunu kullanarak yazıların nasıl biçimlendirilebileceğine bakalım.

```
Console.WriteLine("{0} numaralı atlet {1}. oldu.",no,sira);
```

Burada ekrana yazılacak yazıdaki {0} ifadesi no verisini, {1} ifadesi de sira verisini ifade ediyor. Bu verileri istediğimiz kadar artırabiliriz. Metin biçimlendirme taslağı:

```
{degisken_no,genislik:format}
```

şeklindedir. Farkındaysanız biz yukarıda sadece `degisken_no`'yu kullandık. Diğerleri kullanılmadığı zaman varsayılan değerleri kullanılır. genislik yazının karakter cinsinden minimum genişliğidir. Eğer yazının büyüklüğü genislik değerinden küçükse kalan kısımlar boşlukla doldurulur. Eğer genislik pozitif bir sayıysa yazı sağa dayalı, negatif bir sayıysa sola dayalı olur. Örnek:

```
int a=54;
Console.WriteLine("{0,10} numara",a);
Console.WriteLine("{0,-10} numara",a);
```

Bu kodun konsol ekranındaki çıktısı şöyle olur.

```
        54 numara
54          numara
```

Dikkat ettiyseniz henüz `format`'ı kullanmadık. Formatı kullanmadığımız için şimdiye kadar varsayılan değerdeydi.

```
Console.WriteLine("{0:X}",155);
```

Burada 155 sayısı ekrana 16'lık sayı sisteminde yazılır. Yani ekrana 9B yazılır. Şimdi bu örneği biraz daha geliştirelim:

```
Console.WriteLine("{0:X5}",155);
```

Bu kodla ekrana 0009B yazılır. Yani X'in yanındaki sayı, esas sayının kaç haneli olacağını belirliyor. Buradaki 5 sayısına duyarlıklık değeri denir ve her format belirleyicisi için farklı anlama gelir. Format belirleyiciler, duyarlılık değerleri ve açıklamaları:

Format belirleyici	Format belirleyici açıklaması	Duyarlılık anlamı
C veya c	Para birimi	Ondalık basamakların sayısı
D veya d	Tam sayı	En az basamak sayısı, soldaki basamaklar 0'la beslenir
E veya e	Bilimsel notasyon	Ondalık basamak sayısı
F veya f	Gerçek sayılar (float)	Ondalık basamak sayısı
G veya g	E veya F biçimlerinden kısa olanı	Ondalık basamak sayısı
N veya n	Virgül kullanılarak gerçek sayılar yazılır	Ondalık basamak sayısı
P veya p	Yüzde	Ondalık basamak sayısı
R veya r	Stringe dönüşen türün tekrar eski türe dönüşmesini sağlar	Yok
X veya x	On altılık düzende yazar	En az basamak sayısı, soldaki basamaklar 0'la beslenir

Şimdi bunlarla ilgili bir örnek yapalım:

```
using System;
class Formatlama
{
   static void Main()
   {
      float f=568.87f;
      int a=105;
      Console.WriteLine("{0:C3}",a);
      Console.WriteLine("{0:D5}",a);
      Console.WriteLine("{0:E3}",f);
      Console.WriteLine("{0:F4}",f);
      Console.WriteLine("{0:G5}",a);
      Console.WriteLine("{0:N1}",f);
      Console.WriteLine("{0:P}",a);
      Console.WriteLine("{0:X5}",a);
   }
}
```

Bu programın ekran çıktısı şöyle olur:

```
105,000 TL
00105
5,689E+002
568,8700
105
568,9
%10.500,00
00069
```

NOT: Eğer sayıların normal ondalıklı kısmı duyarlılık değerinden uzunsa yuvarlama yapılır.

String.Format() ve ToString() metotlarıyla biçimlendirme

String sınıfının Format() metodu tıpkı WriteLine() metodu gibi çalışmaktadır. Tek fark WriteLine() biçimlendirdiği yazıyı ekrana yazarken Format() metodu yazıyı bir string olarak tutar. Örnek:

```
string a=String.Format("{0:C3}",50);
```

ToString() metodunda ise biçimlendirme komutu parametre olarak girilir. Onun dışında Format metodundan farkı yoktur. Örnek:

```
string a=50.ToString("0:C3");
```

Tarih saat biçimlendirme

Örnek program:

```
using System;
class TarihZaman
{
   static void Main()
   {
      DateTime dt=DateTime.Now;
      Console.WriteLine("d--> {0:d}",dt);
      Console.WriteLine("D--> {0:D}",dt);
      Console.WriteLine();
      Console.WriteLine("t--> {0:t}",dt);
      Console.WriteLine("T--> {0:T}",dt);
      Console.WriteLine();
      Console.WriteLine("f--> {0:f}",dt);
      Console.WriteLine("F--> {0:F}",dt);
      Console.WriteLine();
      Console.WriteLine("g--> {0:g}",dt);
      Console.WriteLine("G--> {0:G}",dt);
      Console.WriteLine();
      Console.WriteLine("m--> {0:m}",dt);
      Console.WriteLine("M--> {0:M}",dt);
      Console.WriteLine();
      Console.WriteLine("r--> {0:r}",dt);
      Console.WriteLine("R--> {0:R}",dt);
      Console.WriteLine();
```

```
      Console.WriteLine("s--> {0:s}",dt);
      Console.WriteLine();
      Console.WriteLine("u--> {0:u}",dt);
      Console.WriteLine("U--> {0:U}",dt);
      Console.WriteLine();
      Console.WriteLine("y--> {0:y}",dt);
      Console.WriteLine("Y--> {0:Y}",dt);
   }
}
```

Programın ekran çıktısı şuna benzer olmalıdır.

```
d--> 04.01.2009
D--> 04 Ocak 2009 Pazar

t--> 16:25
T--> 16:25:50

f--> 04 Ocak 2009 Pazar 16:25
F--> 04 Ocak 2009 Pazar 16:25:50

g--> 04.01.2009 16:25
G--> 04.01.2009 16:25:50

m--> 04 Ocak
M--> 04 Ocak

r--> Sun, 04 Jan 2009 16:25:50 GMT
R--> Sun, 04 Jan 2009 16:25:50 GMT

s--> 2009-01-04T16:25:50

u--> 2009-01-04 16:25:50Z
U--> 04 Ocak 2009 Pazar 14:25:50

y--> Ocak 2009
Y--> Ocak 2009
```

Özel biçimlendirme oluşturma

Standart biçimlendirme komutlarının yanında bazı özel karakterler yardımıyla kendi biçimlerimizi oluşturabiliriz. Bu özel biçimlendirici karakterler aşağıda verilmiştir:

- # → Rakam değerleri için kullanılır.
- , → Büyük sayılarda binlikleri ayırmak için kullanılır.
- . → Gerçek sayılarda ondalıklı kısımları ayırmak için kullanılır.
- 0 → Yazılacak karakterin başına ya da sonuna 0 ekler.
- % → Yüzde ifadelerini belirtmek için kullanılır.
- E0, e0, E+0, e+0, E-0, e-0 → Sayıları bilimsel notasyonda yazmak için kullanılır.

Örnek bir program:

```
using System;
class OzelBicimlendirme
{
   static void Main()
   {
      Console.WriteLine("{0:#,###}",12341445);
      Console.WriteLine("{0:#.##}",31.44425);
      Console.WriteLine("{0:#,###E+0}",13143212);
      Console.WriteLine("{0:#%}",0.25);
   }
}
```

Bu programın ekran çıktısı şu gibi olmalıdır:

```
12.341.445
31,44
1.314E+4
25%
```

Düzenli ifadeler

Düzenli ifadeler değişken sayıda karakterden oluşabilecek ancak belirli koşulları sağlayan ifadelerdir. Örneğin e-posta adreslerini düşünebiliriz. Dünyada milyonlarca e-posta adresi olmasına ve farklı sayıda karakterden oluşabilmesine rağmen hepsi `kullaniciadi@domainismi.domaintipi` düzenindedir. Örneğin iletisim@microsoft.com bu düzenli ifadeye uymaktadır.

C#'taki düzenli ifade işlemleri temel olarak System.Text.RegularExpressions isim alanındaki Regex sınıfı ile yapılmaktadır. Bir karakter dizisinin oluşturulan düzenli ifadeye uyup uymadığını yine bu isim alanındaki Match sınıfıyla anlarız. Düzenli ifadeler başlı başına bir kitap olabilecek bir konudur. Burada sadece ana hatları üzerinde durulacaktır.

Düzenli ifadelerin oluşturulması

- Bir ifadenin mutlaka istediğimiz karakterle başlamasını istiyorsak ^ karakterini kullanırız. Örneğin ^9 düzenli ifadesinin anlamı yazının mutlaka 9 karakteri ile başlaması demektir. "9Abc" yazısı bu düzene uyarken "f9345" bu düzene uymaz.
- Belirli karakter gruplarını içermesi istenen düzenli ifadeler için \ karakteri kullanılır:

  \D ifadesi ile yazının ilgili yerinde rakam olmayan tek bir karakterin bulunması gerektiği belirtilir.

  \d ifadesi ile yazının ilgili yerinde 0-9 arası tek bir karakterin bulunması gerektiği belirtilir.

  \W ifadesi ile yazının ilgili yerinde alfanumerik olmayan karakterin bulunması gerektiği belirtiliyor. Alfanumerik karakterler a-z, A-Z ve 0-9 aralıklarındaki karakterlerdir.

  \w ifadesi ile yazının ilgili yerinde bir alfanumerik karakterin bulunması gerektiği belirtiliyor.

  \S ifadesi ile yazının ilgili yerinde boşluk veya tab karakterleri haricinde bir karakterin olması gerektiği belirtiliyor.

  \s ifadesi ile yazının ilgili yerinde yalnızca boşluk veya tab karakterlerinden biri bulunacağı belirtiliyor.

Bu öğrendiğimiz bilgiler ışığında 5 ile başlayan, ikinci karakteri rakam olmayan, üçüncü karakteri ise boşluk olan bir düzenli ifade şöyle gösterilebilir:

```
^5\D\s
```

Tahmin edersiniz ki aynı zamanda burada düzenli ifademizin yalnızca 3 harfli olabileceği belirttik. Yukarıdaki `^5\D\s` ifadesine filtre denilmektedir.

- Belirtilen gruptaki karakterlerden bir ya da daha fazlasının olmasını istiyorsak + işaretini kullanırız. Örneğin,

```
\w+
```

filtresi ilgili yerde bir ya da daha fazla alfanumerik karakterin olabileceğini belirtiyor. "123a" bu filtreye uyarken "@asc" bu filreye uymaz. + yerine * kullansaydık çarpıdan sonraki karakterlerin olup olmayacağı serbest bırakılırdı.

- Birden fazla karakter grubundan bir ya da birkaçının ilgili yerde olacağını belirtmek için | (mantıksal veya) karakteri kullanılır. Örnek:

```
m|n|s
```

ifadesinde ilgili yerde m, n ya da s karakterlerinden biri olmalıdır. Bu ifadeyi parantez içine alıp sonuna + koyarsak bu karakterlerden biri ya da birkaçının ilgili yerde olacağını belirtmiş oluruz:

```
(m|n|s)+
```

- Sabit sayıda karakterin olmasını istiyorsak {adet} şeklinde belirtiriz. Örnek:

```
\d{3}-\d{5}
```

filtresine "215-69857" uyarken "54-34567" uymaz.

- ? karakteri, hangi karakterin sonuna gelmişse o karakterden en az sıfır en fazla bir tane olacağı anlamına gelir. Örnek:

```
\d{3}B?A
```

Bu filtreye "548A" veya "875BA" uyarken "875BBA" uymaz.

- . (nokta) işareti ilgili yerde "\n" dışında bir karakterin bulunabileceğini belirtir. Örneğin

```
\d{3}.A
```

filtresine "123sA" ve "8766A" uyar.

- \b bir kelimenin belirtilen yazıyla sonlanması gerektiğini belirtir. Örnek:

```
\d{3}dır\b
```

filtresine "123dır" uyarken "123dırb" uymaz.

- \B ile bir kelimenin başında ya da sonunda bulunmaması gereken karakterler belirtilir. Örnek:

```
\d{3}dır\B
```

filtresine "123dır" veya "dır123" uymazken "123dır8" uyar.

- Köşeli parantezler kullanarak bir karakter aralığı belirtebiliriz. Örneğin ilgili yerde sadece büyük harflerin olmasını istiyorsak [A-Z] şeklinde, ilgili yerde sadece küçük harfler olmasını istiyorsak [a-z] şeklinde, ilgili yerde sadece rakamlar olmasını istiyorsak [0-9] şeklinde belirtebiliriz. Ayrıca sınırları istediğimiz şekilde değiştirebiliriz. Örneğin [R-Y] ile ilgili yerde yalnızca R ve Y arası büyük harfler olabileceğini belirtiriz.

Regex sınıfı

Regex sınıfı bir düzenli ifadeyi tutar. Bir Regex nesnesi şöyle oluşturulur:

```
Regex nesne=new Regex(string filtre);
```

Yani bu yapıcı metoda yukarıda oluşturduğumuz filtreleri parametre olarak veririz. Regex sınıfının Match metodu ise kendisine gönderilen bir yazının düzenli ifadeye uyup uymadığını kontrol eder ve uyan sonuçları Match sınıfı türünden bir nesne olarak tutar.

Match sınıfı

Match sınıfının NextMatch() metodu bir Match nesnesindeki bir sonraki düzenli ifadeyi döndürür. Yazının düzenli ifadeye uyup uymadığının kontrolü ise Match sınıfının Success özelliği ile yapılır. Eğer düzenli ifadeye uygun bir yapı varsa Success özelliğinin tuttuğu değer true olur.

MatchCollection sınıfı

MatchCollection sınıfı ile bir yazı içerisinde düzenli ifadeye uyan bütün Match nesneleri tutulur. Bir MatchCollection nesnesi şöyle oluşturulur:

```
MatchCollection mc=Regex.Matches(string yazi,string filtre)
```

Burada Regex sınıfının static Matches metodu kullanılmışır. Regex sınıfının Matches metodu iki parametre alır. İlk parametresi kontrol edilmek istenen yazı, ikincisi de filtredir. Bir MatchCollection nesnesi oluşturduktan sonra foreach döngüsü yardımıyla bu kolleksiyondaki bütün Match nesnelerine erişebiliriz. Match nesnesine eriştikten sonra düzenli ifadeye uyan karakter dizisinin orijinal yazıdaki yerini Index özelliğiyle ve yazının kendisini ToString() metoduyla elde edebiliriz. MatchCollection sınıfının Count özelliği ile düzenli ifadeye uyan alt karakter dizilerinin sayısını elde ederiz. Eğer Count özelliğinin tuttuğu değer 0 ise düzenli ifadeye uyan yazı bulunamadı demektir. Şimdi bu teorik bilgileri uygulamaya dökelim. Filtremiz şöyle olsun:

```
A\d{3}(a|o)+
```

Bu filtreyle düzenli ifademizin şöyle olduğunu söyleyebiliriz:

- İlk karakter A olacak.
- A'dan sonra üç tane rakam gelecek.
- Üç rakamdan sonra a ya da o karakterlerinden biri ya da birkaçı gelecek. Şimdi programımızı yazalım. Programımız kullanıcının girdiği yazıdaki filreye uyan kısımları kontrol etsin:

```
using System;
using System.Text.RegularExpressions;
class duzenli
{
   static void Main()
   {
      string filtre=@"A\d{3}(a|o)+";
      Console.Write("Yazı girin: ");
      string yazi=Console.ReadLine();
      MatchCollection mc=Regex.Matches(yazi,filtre);
      if(mc.Count==0)
      {
         Console.WriteLine("Yazıda filreye uyumlu kısım yok!");
         return;
```

```
        }
        foreach(Match bulunan in mc)
        {
            Console.WriteLine("Bulunan yer: "+bulunan.Index);
            Console.WriteLine("Bulunan yazı: "+bulunan.ToString());
        }
    }
}
```

Bu programda kullanıcının `A123aA657oA456oao` girdiğini varsayarsak ekran çıktısı şu şekilde olur.

```
Bulunan yer: 0
Bulunan yazı: A123a
Bulunan yer: 5
Bulunan yazı: A657o
Bulunan yer: 10
Bulunan yazı: A456oao
```

Programdan da anlayacağınız üzere Index ve ToString() üye elemanları Match sınıfına aittir ve static değildir. Bu programımızı MatchCollection sınıfıyla yapmıştık. Şimdi aynı programı Regex ve Match sınıflarıyla yapalım:

```
using System;
using System.Text.RegularExpressions;
class duzenli
{
    static void Main()
    {
        string filtre=@"A\d{3}(a|o)+";
        Console.Write("Yazı girin: ");
        string yazi=Console.ReadLine();
        Regex nesne=new Regex(filtre);
        Match a=nesne.Match(yazi);
        Console.WriteLine(a.Success);
        Console.WriteLine(a.ToString());
        Console.WriteLine(a.Index);
        Console.WriteLine(a.NextMatch());
    }
}
```

Bu programda kullanıcının `A123aA657oA456oao` girdiğini varsayarsak ekran çıktısı şöyle olur.

```
True
A123a
0
A657o
```

Düzenli ifadelerin içinden bölüm seçme

Bazen bir yazının bir düzenli ifadeye uyup uymadığının öğrenilmesi bizim için yeterli gelmez. Bazen bu uyan yazıların bazı kısımlarını ayrı ayrı görmek isteyebiliriz. Örnek bir program:

```
using System;
using System.Text.RegularExpressions;
class gruplama
{
   static void Main()
   {
      string filtre=@"asf(\d+)(\w+)";
      Console.Write("Yazı girin: ");
      string yazi=Console.ReadLine();
      Regex nesne=new Regex(filtre);
      Match a=nesne.Match(yazi);
      Console.WriteLine("Uyan yazı: "+a.ToString());
      Console.WriteLine("Birinci kısım: "+a.Groups[1].ToString());
      Console.WriteLine("İkinci kısım: "+a.Groups[2].ToString());
   }
}
```

Kullanıcının `asf31321edcve34` girdiğini varsayarsak bu program ekrana şunları yazar:

```
Uyan yazı: asf31321edcve34
Birinci kısım: 31321
İkinci kısım: edcve34
```

Gördüğünüz gibi filtreyi, istediğimiz kısmı parantez içine alarak parçalıyoruz, sonra da bu kısımlara Match sınıfına ait, static olmayan Groups özelliğine eklenen indeksleyici ile erişiyoruz, sonra da bu kısımları ToString() metodu yardımıyla ekrana yazdırıyoruz.

Regex sınıfının önemli metotları

Split() metodu

Split() metodu bir yazıyı belirli bir düzenli ifadeye göre parçalara ayırır ve bütün parçaları bir string dizisi olarak tutar. Örnek program:

```
using System;
using System.Text.RegularExpressions;
class split
{
   static void Main()
   {
      string filtre=",+";
      string yazi="21,44,,34,,,332,21";
      Regex nesne=new Regex(filtre);
      string[] parcalar=nesne.Split(yazi);
      foreach(string a in parcalar)
        Console.WriteLine(a);
   }
```

```
}
```

Bu program ekrana alt alta 21, 44, 34, 332 ve 21 (virgüller olmadan) yazar. Filtremiz bir ya da daha fazla yan yana virgüldür ve bu vürgüllere göre yazımız parçalanmıştır. Bunu String sınıfının Split() metoduyla yapamazdık.

Replace() metodu

Replace() metodu, bir yazının bir düzenli ifadeye uyan kısımlarını başka bir yazıyla değiştirmek için kullanılır. Örnek:

```
using System;
using System.Text.RegularExpressions;
class replace
{
   static void Main()
   {
      string filtre=@"\d+:\d+";
      string yazi="Saati belirtmek için : işareti kullanılır. Örnek:
12:35";
      Regex nesne=new Regex(filtre);
      string degistirilmis=nesne.Replace(yazi,"00:00");
      Console.WriteLine(degistirilmis);
   }
}
```

Bu program ekrana `Saati belirtmek için : işareti kullanılır. Örnek: 00:00` yazar. Bu şekilde stringdeki bütün saat bilgilerini değiştirebilirdik. Yine bu da String sınıfındaki Replace() metodunda olmayan bir özellik.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Kalıtım

Diyelim ki elinizde A diye bir sınıf var. Ancak B diye bir sınıf daha oluşturmanız gerekiyor. Ancak bu B sınıfının içinde A sınıfındaki özellik ve metotların da bulunması gerekiyor. İşte bu durumda B sınıfını A sınıfından türetmeniz gerekir. Türetme kalıtım yoluyla olduğu için A sınıfının bütün üye elemanları B sınıfına adeta kopyalanır. Daha sonra B sınıfının kendine özel üye elemanlarını yazabiliriz. .Net kütüphanesindeki birçok sınıf birbirlerinden türetilmiştir. Örneğin temel veri türleri dediğimiz byte, int, uint, short, float ve bezerlerinin tamamı object sınıfından türetilmiştir. Bu sayede normalde object sınıfında bulunan ToString() metodunu bu yapı türünden nesnelerde de kullanabilirsiniz. Sınıflar türetilebilmesine rağmen yapılar türetilemez. Ancak bir sınıfın türetilmişinden yapı oluşturulabilir. C#'ta türetme şöyle yapılır:

```
class A
{
    ...
}
class B:A
{
    ...
}
```

Burada B sınıfı A sınıfından türetilmiştir ve artık B sınıfı A sınıfındaki bütün üye elemanları da içerir.

protected erişim belirleyicisi

Şimdiye kadar public ve private erişim belirleyicilerini görmüştük. Şimdi ise yeni bir erişim belirleyicisi olan protected'ı göreceğiz. Normalde bir sınıfı türettiğimizde türetilmiş sınıfın içinden ana sınıfta private olarak belirtilmiş üye elemanlara erişemeyiz. Ancak bu, private üye elemanların türetilmediği anlamına gelmez. Örneğin:

```
using System;
class A
{
   static private int Ozellik;
}
class B:A
{
   static void Main()
   {
      Console.WriteLine(Ozellik);
   }
}
```

Burada B sınıfından Ozellik özelliğine erişilemez. Ancak halen Ozellik özelliği B sınıfında vardır (kopyalanmıştır). Eğer bu programda private yerine protected'ı kullansaydık Ozellik özelliğine erişebilirdik. Yani protected erişim belirleyicisi ile belirtilen üye elemanlar türetilmiş sınıfın içinden kullanılabilirler. Ancak halen diğer sınıfların erişimine kapalıdırlar. Eğer ortada bir türetme söz konusu değilse protected erişim belirleyicisinin private'ten farkı kalmaz.

NOT: Eğer türetilen bir metottan bir private üye elemana erişiliyorsa bu durumda bir engelleme gelmez. Bu da az önce söylediğimiz "Private erişim belirleyici türetilmeye engel değildir." tezini doğrular. Örnek:

```
using System;
class A
{
   static private int Ozellik;
   static public void metot()
   {
      Console.WriteLine(Ozellik);
   }
}
class B:A
{
   static void Main()
   {
      metot();
   }
}
```

Bu program herhangi bir hata vermez.

Yapıcı metotlar ve kalıtım

C#'ta yapıcı metotların türetimiyle ilgili şu kurallar geçerlidir:

1. C#'ta yapıcı metotlar fiziksel olarak türetilmez.
2. Yavru sınıf türünden bir nesne yaratıldığında önce ana sınıfın parametre almayan yapıcı metodu, ardından yavru sınıftaki imzaya uyan yapıcı metot çalıştırılır.
3. Yavru sınıf türünden nesne yaratımında daima yavru sınıfın imzaya uyan bir yapıcı metodu olması gerekir.
4. Yavru sınıf türünden nesne yaratımlarında, ana sınıfın parametre almayan yapıcı metodu yavru sınıfın üye elemanlarıyla işlem yapar.
5. Yavru sınıf türünden nesne yaratımında, yavru sınıftaki ilgili (imzası uygun) yapıcı metoda base takısı eklenmişse ana sınıfın parametre almayan yapıcı metodu çalıştırılmaz. (base takısını birazdan göreceğiz)

Şimdi bu kuralları örneklendirelim:

```
using System;
class ana
{
   public ana()
   {
      Console.WriteLine("ana sınıfının parametre almayan yapıcı
metodu");
   }
}
class yavru:ana
{
   public yavru(int a)
   {
      Console.WriteLine("yavru sınıfının parametre alan yapıcı metodu.
alınan parametre: "+a);
   }
```

```
}
class esas
{
   static void Main()
   {
      yavru y=new yavru(5);
   }
}
```

Bu programda ekrana alt alta `ana sınıfının parametre almayan yapıcı metodu` ve `yavru sınıfının parametre alan yapıcı metodu. alınan parametre: 5` yazacaktır. Bu ikinci kuralın örneklendirilmesiydi. Şimdi üçüncü kuralı örneklendirelim:

```
using System;
class ana
{
   public ana()
   {
      Console.WriteLine("ana sınıfının parametre almayan yapıcı
metodu");
   }
}
class yavru:ana{}
class esas
{
   static void Main()
   {
      yavru y=new yavru();
   }
}
```

Bu program hata vermez. Çünkü sınıflar konusunda öğrendiğimiz üzere bir sınıfta hiç yapıcı metot olmadığı durumlarda varsayılan yapıcı metot oluşturulur. Varsayılan yapıcı metot parametre almaz ve hiçbir şey yapmaz. Bu örnekte y nesnesini yaratırken parametre verseydik yukarıda bahsettiğimiz üçüncü kural ihlal edilmiş olacaktı ve dolayısıyla programımız hata verecekti. Şimdi dördüncü kuralı örneklendirelim:

```
using System;
class ana
{
   public int ozellik;
   public ana()
   {
      ozellik=5;
   }
}
class yavru:ana{}
class esas
{
   static void Main()
```

```
    {
        yavru y=new yavru();
        Console.WriteLine(y.ozellik);
    }
}
```

Bu örnekte ekrana 5 yazılacaktır. Aslına bakarsanız dördüncü kuralın tersinin olması imkansız. Çünkü zaten ortada ana tipinden bir nesne yok, o yüzden ana sınıfının parametre almayan yapıcı metodunun kendi sınıfının üye elemanlarıyla çalışması bu örnekte imkansız. Ancak yine de fiziksel olarak bir sınıfta olmayan bir yapıcı metodun o sınıfın üye elemanlarıyla çalışması ilginç. Şimdi başka bir örnek yapalım:

```
using System;
class ana
{
    public ana(int a){}
}
class yavru:ana{}
class esas
{
    static void Main()
    {
        yavru y=new yavru();
    }
}
```

Bu program hata verir. Çünkü yukarıda saydığımız ikinci kuralı ihlal etmektedir. Çünkü ana sınıfta parametre almayan bir yapıcı metot yoktur. Ana sınıfta bir yapıcı metot tanımlandığı için varsayılan yapıcı metot oluşturulmamıştır. Şimdi beşinci kuralı örneklendirelim:

```
using System;
class A
{
    public int Ozellik1;
    public int Ozellik2;
    public A()
    {
        Console.WriteLine("Deneme"),
    }
    public A(int ozellik1,int ozellik2)
    {
        Ozellik1=ozellik1;
        Ozellik2=ozellik2;
    }
}
class B:A
{
    public int Ozellik3;
    public int Ozellik4;
    public B(int ozellik3,int ozellik4,int ozellik1,int
```

```
ozellik2):base(ozellik1,ozellik2)
   {
      Ozellik3=ozellik3;
      Ozellik4=ozellik4;
   }
}
class esas
{
   static void Main()
   {
      B b=new B(3,4,1,2);
      Console.WriteLine(b.Ozellik1+" "+b.Ozellik2+" "+b.Ozellik3+"
"+b.Ozellik4);
   }
}
```

Bu program ekrana `1 2 3 4` yazar. Bu örnekte `base` anahtar sözcüğü ana sınıftaki yapıcı metodu temsil etmektedir. Örneğimizde yavru sınıfın yapıcı metodu 4 parametre almakta, bu aldığı parametrelerin ikisini kendi bloğunda kullanmakta kalan iki parametreyi de ana sınıfın imzası uygun yapıcı metoduna göndermektedir. Ana sınıfın imzası uygun yapıcı metodu çalıştığında yavru sınıfın üye elemanlarıyla işlem yapacaktır. Asıl konumuza gelecek olursak bu örnekte yavru sınıfın bir yapıcı metoduna eklenen base takısı ile ana sınıfın bir yapıcı metodunu çalıştırdığımız için yavru sınıftaki base takısı eklenmiş ilgili yapıcı metodu çalıştıracak şekilde yavru sınıf türünden bir nesne yaratıldığında ana sınıfın parametre almayan yapıcı metodu çalıştırılmayacaktır.

NOT: base anahtar sözcüğü bu şekilde yalnızca yapıcı metotlarla kullanılabilir. Yani base anahtar sözcüğünü yalnızca yavru sınıftaki yapıcı metoda ekleyebiliriz ve base anahtar sözcüğünün ana sınıfta var olan bir yapıcı metodu belirtmesi gerekir.

Çoklu türetmeler

Sınıflar tıpkı nine, anne, çocuk yapısında olduğu gibi ard arda türetilebilir. Yani örneğin B sınıfı A sınıfında türetilip C sınıfı da B sınıfından türetilebilir. Bu durumda C sınıfı türünden bir nesne yarattığımızda eğer C sınıfının ilgili yapıcı metoduna base takısını eklememişsek önce A, sonra B, sonra da C sınıfının yapıcı metotları çalıştırılır. Yani gidişat anadan yavruya doğrudur. Ayrıca tahmin edebileceğiniz gibi C sınıfı hem A'nın hem de B'nin bütün üye elemanlarına sahip olur. Örnek:

```
using System;
class A
{
   public A()
   {
      Console.WriteLine("A sınıfı");
   }
}
class B:A
{
   public B()
   {
      Console.WriteLine("B sınıfı");
   }
```

```
}
class C:B
{
   public C()
   {
      Console.WriteLine("C sınıfı");
   }
   static void Main()
   {
      C nesne=new C();
   }
}
```

Bu program ekrana alt alta `A sınıfı`, `B sınıfı` ve `C sınıfı` yazacaktır. Bu örnekte base anahtar sözcüğünün kullanımı ise şöyledir:

```
using System;
class A
{
   public int OzellikA;
   public A(int a)
   {
      OzellikA=a;
   }
}
class B:A
{
   public int OzellikB;
   public B(int b,int a):base(a)
   {
      OzellikB=b;
   }
}
class C:B
{
   public int OzellikC;
   public C(int c,int b,int a):base(b,a)
   {
      OzellikC=c;
   }
   static void Main()
   {
      C nesne=new C(12,56,23);
      Console.WriteLine(nesne.OzellikA+" "+nesne.OzellikB+"
"+nesne.OzellikC);
   }
}
```

Gördüğünüz gibi base anahtar sözcüğü kendisinin bir üstündeki sınıfın yapıcı metoduna parametre gönderiyor ancak tabii ki işlemler alt sınıftaki üye elemanlar için yapılıyor. Bu durumda -hangi sınıf türünden nesne yaratırsak yaratalım- hiçbir sınıfın parametre almayan yapıcı metodu çalıştırılmayacaktır (tabii ki nesne yaratırken gerekli parametreleri verdiğimiz müddetçe). Bu örneği şöyle değiştirirsek program hata verir:

```
using System;
class A
{
   public int OzellikA;
   public A(int a)
   {
      OzellikA=a;
   }
}
class B:A
{
   public int OzellikB;
   public B(int b)
   {
      OzellikB=b;
   }
}
class C:B
{
   public int OzellikC;
   public C(int c,int b):base(b)
   {
      OzellikC=c;
   }
   static void Main()
   {
      C nesne=new C(12,56);
      Console.WriteLine(nesne.OzellikA+" "+nesne.OzellikB+"
"+nesne.OzellikC);
   }
}
```

Gördüğünüz gibi bu örnekte B sınıfının yapıcı metodundaki base takısı kaldırılmış ve gerekli düzenlemeler yapılmış. Bu örnekte B sınıfının yapıcı metodu çalıştırıldığı için ve B sınıfının söz konusu yapıcı metodunda base takısı olmadığı için B sınıfına göre ana sınıfın (bu durumda A sınıfı oluyor) parametre almayan yapıcı metodu çalıştırılmaya çalışılmıştır. A sınıfının parametre almayan yapıcı metodu olmadığı için program hata vermiştir. Yani ana sınıfın parametre almayan yapıcı metodu, tam olarak yavru sınıf tipinden nesne yaratılmasa da, sadece yavru sınıfın base takısı almayan yapıcı metodunun çalıştırılması durumunda da çalışır.

İsim saklama

Muhtemelen şunu merak etmişsinizdir: yavru sınıfta ana sınıftakiyle aynı isimli bir üye eleman varsa ne olacak? İşte bu durumda ana sınıftaki üye eleman gizlenir ve normal yöntemlerle erişilemez. Buna isim saklama denir. Örnek:

```
using System;
class A
{
   public int a=10;
}
class B:A
{
   public int a=20;
   static void Main()
   {
      B nesne=new B();
      Console.WriteLine(nesne.a);
   }
}
```

Bu programda ekrana 20 yazılır yani değeri 20 olan a özelliği, değeri 10 olan a özelliğini gizlemiştir. Ancak derleyici bu programda bizim isim gizlemeyi bilinçsiz olarak yaptığımızı düşünür ve uyarı verir (hata vermez). Böyle bir durumda isim gizlemeyi açıkça belirtmeliyiz. Böylelikle hem programımızdaki muhtemel gözden kaçmaları hem de derleyicinin uyarı vermesini önleriz. İsim gizlemeyi açıkça belirtmek için new anahtar sözcüğünü kullanırız. Yukarıdaki programı şöyle yazarsak derleyici uyarı vermez:

```
using System;
class A
{
   public int a=10;
}
class B:A
{
   public new int a=20;
   static void Main()
   {
      B nesne=new B();
      Console.WriteLine(nesne.a);
   }
}
```

Peki gizlediğimiz üye elemana erişebilir miyiz? Cevabımız evet. Bunun için base anahtar sözcüğünü kullanırız. Örnek:

```
using System;
class A
{
   public int a=10;
}
class B:A
```

```
{
   public new int a=20;
   int Metot()
   {
      return base.a;
   }
   static void Main()
   {
      B nesne=new B();
      Console.WriteLine(nesne.Metot());
   }
}
```

Burada ekrana 10 yazılır. base anahtar sözcüğü static metotların içinde kullanılamaz.

Ana ve yavru sınıf nesneleri

C# tür güvenliğine maksimum derecede önem vermektedir. Bu yüzden, eğer özel tür dönüşüm operatörleri bildirilmemişse farklı sınıf türlerinden nesneler birbirlerine atanamaz. Ancak türeyen sınıflarda bu kural delinir. Ana sınıf türünden nesnelere yavru sınıf türünden nesneler atanabilir. Örneğin C#'taki temel veri türlerinin object sınıfından türetildiğini söylemiştik. Bu sayede bir object nesnesine her türden veri atanabilir. Örnek bir program:

```
using System;
class A
{
}
class B:A
{
}
class MainMetodu
{
   static void Main()
   {
      A nesne1=new A();
      B nesne2=new B();
      nesne1=nesne2;
   }
}
```

Bu programda gözden kaçırmamamız gereken nokta nesne1 üzerinden B sınıfının kendine özgü üye elemanlarına erişemeyeceğimizdir. Bu şekilde ana sınıf türünden bir nesnenin kullanılabildiği her yerde yavru sınıf türünden bir nesneyi de kullanabiliriz. Örneğin bu örneğimizi düşünecek olursak bir metodun parametresi A tipinden ise bu metoda parametre olarak B tipinden bir nesneyi de verebiliriz. Çünkü her B nesnesi A nesnesinin taşıdığı bütün üye elemanları taşır. Bunu bilinçsiz tür dönüşümüne benzetebiliriz.

object sınıfı

Şimdiye kadar bütün temel veri türlerinin object sınıfından türediğini söylemiştim. Asıl bombayı şimdi patlatıyorum. Bütün sınıflar gizlice object sınıfından türer. Bunu kanıtlamak için şu programı yazın:

```
using System;
class A
{
}
class Ana
{
   static void Main()
   {
      A nesne=new A();
      Console.WriteLine(nesne.ToString());
   }
}
```

ToString() metodu normalde object sınıfına aittir. Ancak bütün sınıf nesneleriyle kullanılabilir. Bu programda ekrana nesnenin türü olan A yazılır. Başka bir örnek:

```
using System;
class A
{
}
class Ana
{
   static void Main()
   {
      A nesne=new A();
      object a=nesne;
   }
}
```

Yine bu program da son derece hatasızdır.

Sanal metotlar

Sanal metotlar ana sınıf içinde bildirilmiş ve yavru sınıf içinde tekrar bildirilen metotlardır. Şimdilik bunun adı isim saklamadan başka bir şey değildir. Ancak bazı anahtar sözcükler ekleyerek bunun klasik bir isim saklama işleminden farklı olmasını sağlayacağız. Şimdi klasik şöyle bir program yazalım:

```
using System;
class A
{
   public void Metot()
   {
      Console.WriteLine("A sınıfı");
   }
}
class B:A
```

```
 {
    public void Metot()
    {
       Console.WriteLine("B sınıfı");
    }
    static void Main()
    {
       A nesneA=new A();
       B nesneB=new B();
       nesneA=nesneB;
       nesneA.Metot();
    }
 }
```

Bu program ekrana `A sınıfı` yazar. Çünkü nesneA nesnesi A sınıfı türündendir ve bu nesne üzerinden Metot() metoduna erişilirse A sınıfına ait Metot() metodu çalıştırılır. Şimdi böyle bir durumda B sınıfına ait Metot() metodunun çalıştırılmasını sağlayacağız.

```
using System;
class A
{
   virtual public void Metot()
   {
      Console.WriteLine("A sınıfı");
   }
}
class B:A
{
   override public void Metot()
   {
      Console.WriteLine("B sınıfı");
   }
   static void Main()
   {
      A nesneA=new A();
      B nesneB=new B();
      nesneA=nesneB;
      nesneA.Metot();
   }
}
```

Bu program ekrana `B sınıfı` yazacaktır. Dikkat ettiyseniz bu programın öncekinden tek farkı A sınıfına ait metodun başına virtual anahtar sözcüğünün getirilmesi ve B sınıfına ait metodun başına da override anahtar sözcüğünün getirilmesi. Ana sınıftaki virtual anahtar sözcüğüyle ana sınıfa ait metodun bir sanal metot olmasını, yavru sınıftaki override anahtar sözcüğüyle de ana sınıfa ait aynı adlı sanal metodun görmezden gelinmesini sağladık. Bu sayede NesneA nesnesinin gizli türüne ait metot çalıştırıldı. Benzer mantıkla düşünecek olursak `object a='r';` gibi bir ifadede a nesnesinin gizli türünün char olduğunu söyleyebiliriz. Bu konu hakkında bilmeniz gereken diğer şeyler:

- Farkındaysanız burada bir nesnenin farklı türden bir nesne gibi davranması söz konusudur. Programlama jargonunda bunun adı çok biçimliliktir (polimorfizm).
- Eğer ana sınıftaki metot virtual olarak bildirilmeseydi ve yavru sınıftaki metot override edilseydi programımız hata verip derlenmezdi.
- Eğer yavru sınıftaki metot override olarak bildirilmeseydi ana sınıftaki metot çalıştırılırdı (program hata vermezdi).
- Yavru sınıftaki override olarak belirtilen metotla ana sınıftaki virtual olarak bildirilen metot aynı isimli ve imzaları da aynı olmalıdır.
- static metotlar virtual olarak bildirilemez. Şimdi örneğimizi biraz geliştirelim:

```
using System;
class A
{
   virtual public void Metot()
   {
      Console.WriteLine("A sınıfı");
   }
}
class B:A
{
   override public void Metot()
   {
      Console.WriteLine("B sınıfı");
   }
}
class C:B
{
   public void Metot()
   {
      Console.WriteLine("C sınıfı");
   }
   static void Main()
   {
      A nesneA=new A();
      C nesneC=new C();
      nesneA=nesneC;
      nesneA.Metot();
   }
}
```

Bu program ekrana `B sınıfı` yazar. Çünkü C sınıfına ait metot override edilmediği için C'den önce gelen türeme zincirindeki son override edilen metot çalıştırılacaktır. Şimdi başka bir örnek yapalım:

```
using System;
class A
{
   virtual public void Metot()
   {
      Console.WriteLine("A sınıfı");
```

```
    }
}
class B:A
{
    override public void Metot()
    {
        Console.WriteLine("B sınıfı");
    }
}
class C:B
{
    override public void Metot()
    {
        Console.WriteLine("C sınıfı");
    }
}
class D:C
{
    public new void Metot()
    {
        Console.WriteLine("D sınıfı");
    }
}
class E:D
{
    public new void Metot()
    {
        Console.WriteLine("E sınıfı");
    }
}
class F
{
    static void Main()
    {
        A a=new A();
        E e=new E();
        a=e;
        a.Metot();
    }
}
```

Bu örnekte ekrana `C sınıfı` yazılacaktır. override anahtar sözcüğü zaten bilinçli bir isim gizleme yapıldığını belirtir. Bu yüzden override olarak belirtilen bir üye elemanı tekrar new olarak belirtmek hatalıdır. Ayrıca bir türeme zincirindeki override döngüsü sondan delinebilir ama ortadan delinemez. Çünkü ortadan delinmeye çalışıldığında normal bir üye elemanı override etmemiz gerekir ki bunun yasak olduğunu daha önce söylemiştik.

Özet sınıf ve üye elemanlar

Özet sınıflar

Bazen bir ana sınıfın tek başına bir işlevi olmayabilir. Ana sınıfın tek görevi yavru sınıflara ait bazı ortak üye elemanları barındırmak olabilir ve ana sınıf türünden nesne yaratılmasını engellemek isteyebiliriz. İşte bu gibi durumlarda ana sınıf özet sınıf olarak bildirilir. Özet sınıfları bildirmek için `abstract` anahtar sözcüğü kullanılır. Örnek:

```
using System;
abstract class A
{
   public string Metot()
   {
      return "Deneme";
   }
}
class B:A
{
   static void Main()
   {
      B nesne1=new B();
      Console.WriteLine(nesne1.Metot());
      //A nesne2=new A();
      //Yukarıdaki satır olsaydı program hata verirdi.
   }
}
```

Özet metotlar

Metotlar da, tıpkı sınıflar gibi özet olarak bildirilebilir. Özet metotlar her yavru sınıfta bulunması gereken, ancak içeriği bu yavru sınıf tarafından belirlenen metotlar oluşturmak için kullanılır. Bu sayede programcının bu üye elemanları unutarak es geçmesi engellenir. Örnek bir özet metot yaratımı:

```
abstract public void Metot();
```

Gördüğünüz gibi normal metot yaratım satırının başına bir de abstract anahtar sözcüğü ekleniyor. Metodun bloğu yok, dolayısıyla da satır ; işareti ile sonlandırılıyor. Özet metotlarla ilgili bilmeniz gerekenler:

- Özet metotlar yalnızca özet sınıfların içinde bildirilebilir.
- Ana sınıfın içinde bildirilen özet metodu mutlaka, o sınıftan türeyen yavru sınıflar içinde override etmeliyiz.
- Özet metotlar içsel olarak zaten sanal oldukları için tekrar virtual olarak bildirmek hatalıdır.
- Elbette ki özet bir sınıf özet olmayan metot içerebilir.
- Özet metotlar private olarak bildirilemez. Ya public ya da protected olmalıdırlar.
- static metotlar özet olarak bildirilemez. Aynı şekilde özet metot bildiriminde virtual ve override anahtar sözcükleri kullanılamaz.

Özet özellikler

Özet özellikler, özet metotların taşıdığı bütün özellikleri taşırlar. Bunlara ek olarak şunları sayabiliriz:

- Yalnızca sahte özellikler özet olarak bildirilebilir.
- Ana sınıftaki özet sahte özelliği override eden yavru sınıftaki özelliğin set-get durumu ana sınıftaki özet özellikle aynı olmalıdır. Yani eğer özet özellikte sadece get bloğu varsa bu özelliği override eden özelliğin de sadece get bloğu olmalıdır. Aynı durum set bloğu için de geçerlidir. Benzer şekilde özet özellikte hem set hem de get varsa override özellikte de bu ikisi olmalıdır. Örnek:

```
using System;
abstract class A
{
   abstract public int ozellik
   {
      set;
      get;
   }
}
class B:A
{
   override public int ozellik
   {
      get{return 100;}
      set{Console.WriteLine("Bu bir denemedir");}
   }
   static void Main()
   {
      B nesne=new B();
      Console.WriteLine(nesne.ozellik);
      nesne.ozellik=200;
   }
}
```

Gördüğünüz gibi özet sahte özelliğin set ve get blokları ayrı olarak yazılmıyor. set ve/veya get sözcüklerinin sonuna ; işareti koyuluyor.

NOT: abstract, virtual ve override üye elemanlar private olamaz.

NOT: Daha önce sanal metotları görmüştük. Sanal sahte özellikler de olabilir. Örnek:

```
using System;
class A
{
   virtual public int ozellik
   {
      set{}
      get{return 12;}
   }
}
class B:A
{
```

```
   override public int ozellik
   {
      get{return 100;}
      set{Console.WriteLine("Bu bir denemedir");}
   }
   static void Main()
   {
      B nesne=new B();
      A nesne2=new A();
      nesne2=nesne;
      Console.WriteLine(nesne2.ozellik);
      nesne2.ozellik=200;
   }
}
```

Bu program alt alta `100` ve `Bu bir denemedir` yazacaktır.

sealed anahtar sözcüğü

Bazen bir sınıftan türetilme yapılamamasını isteyebiliriz. İşte bu gibi durumlarda sealed anahtar sözcüğünü kullanabiliriz. Örnek:

```
 sealed class Sinif
 {
    ...
 }
```

Bir sealed sınıf yukarıdaki gibi bildirilir. Bu sınıf türetilemez.
NOT: abstract (özet) sınıflar sealed olarak işaretlenemez.
NOT: sealed sınıflara ek olarak static sınıflar ve yapılar da türetilmeyi desteklemez.

Kalıtımla ilgili son notlar

Genel kural olarak yavru sınıftaki ana sınıftan devralınan elemanlar, ana sınıftan devralınan elemanları; yavru sınıfta isim gizleme yoluyla yeniden yazılan veya ana sınıfta bulunmayıp yavru sınıfta yazılan elemanlar yavru sınıftaki isim gizleyen elemanları kullanma eğilimindedir. Bu durumları örneklendirelim:

```
using System;
class A
{
   public void Metot1()
   {
      Metot2();
   }
   public void Metot2()
   {
      Console.WriteLine("A sınıfı");
   }
}
class B:A
```

```
{
   public new void Metot2()
   {
      Console.WriteLine("B sınıfı");
   }
}
class Ana
{
   static void Main()
   {
      B b=new B();
      b.Metot1();
   }
}
```

Bu programda ekrana A sınıfı yazılır.

```
using System;
class A
{
   public void Metot1()
   {
      Metot2();
   }
   public void Metot2()
   {
      Console.WriteLine("A sınıfı");
   }
}
class B:A
{
   public new void Metot1()
   {
      Metot2();
   }
   public new void Metot2()
   {
      Console.WriteLine("B sınıfı");
   }
}
class Ana
{
   static void Main()
   {
      B b=new B();
      b.Metot1();
   }
}
```

Bu programda da ekrana `B sınıfı` yazılacaktır. İstersek bu öncelikleri `virtual` ve `override` anahtar sözcükleriyle değiştirebiliriz. Şimdi birinci örneği şöyle değiştirelim:

```
using System;
class A
{
   public void Metot1()
   {
      Metot2();
   }
   virtual public void Metot2()
   {
      Console.WriteLine("A sınıfı");
   }
}
class B:A
{
   override public void Metot2()
   {
      Console.WriteLine("B sınıfı");
   }
}
class Ana
{
   static void Main()
   {
      B b=new B();
      b.Metot1();
   }
}
```

Bu örnekte ekrana `B sınıfı` yazılır. İkinci örnekteki durumu çözmek için base anahtar sözcüğü kullanılabilir. Ancak onu örnekleme gereksinimi görmüyorum. Çünkü daha önce görmüştük.

Özet geçmek gerekirse ana sınıftan devralınan elemanların da isim gizleyen elemanları kullanmasını istiyorsak ana sınıftaki elemanı virtual, yavru sınıftaki elemanı override olarak belirtiriz. Eğer çoklu türetme söz konusuysa anne sınıfta override edilmiş elemanı torun sınıfta tekrar override ederek nine sınıftan devralınan bir elemanın torun sınıftaki isim gizleyen elemanları kullanmasını sağlayabiliriz. Eğer torun sınıftaki elemanı override etmezsek torun sınıftaki nine sınıftan devralınan elemanlar anne sınıfın elemanını kullanacaktır.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Arayüzler

İsterseniz şimdi geçen bölümün son konularını biraz hatırlayalım. Geçen bölümde sanal metot ve sanal sahte özellikleri görmüştük. Bu sayede bir nesnenin kendi türüyle değil, gizli türüyle işlem yapılmasını sağlamıştık. Geçen bölümde gördüğümüz diğer önemli şey ise özet metot ve özet sahte özelliklerdi. Özet metot ve özet sahte özellikler ana sınıf içinde bildiriliyordu, içerikleri yoktu ve bu sınıflardan türeyen sınıflar da bu metot ve sahte özellikleri içermek zorundaydı. Yani diğer bir deyişle ana sınıftaki özet üye elemanlar yavru sınıflara şunları diyordu: "Bizi içermek zorundasınız, ancak içeriğimizi kendiniz yazacaksınız". Yani özet üye elemanları kılavuza benzetebiliriz. Kılavuz bir işin nasıl yapılacağını ana hatlarıyla anlatır, ancak bu işi asıl yapan kılavuzu okuyan kişidir. Şimdi öyle bir sınıf düşünün ki sadece özet üye elemanlardan oluşsun. Yani bu sınıfın işlevsel olarak hiçbir faydası yok. Sadece kendinden türeyen sınıflara kılavuzluk yapıyor. İşte, arayüzlere sınıfların sadece özet üye eleman içerebilenleri diyebiliriz.

Arayüz bildirimi

- Arayüzler interface anahtar sözcüğüyle bildirilir.
- Bir arayüzde özellik, metot, indeksleyici, temsilci ve olay bildirilebilir. Temsilci ve olayları ileride göreceğiz.
- Arayüz isimleri geleneksel olarak I harfi ile başlar, ancak zorunlu değildir.
- Arayüz elemanlarını static olarak bildiremeyiz.
- Arayüz elemanları içsel olarak zaten publictir. Tekrar bir erişim belirleyici kullanmak hatalıdır.
- Arayüzler sahte olmayan özellik içeremez.
- Arayüzlerde yapıcı ve yıkıcı metot olmaz.

Şimdi iki metodu, bir sahte özelliği ve bir indeksleyicisi olan bir arayüz yazalım:

```
interface IArayuz
{
    int Metot1();
    int Metot2();
    int sahteozellik
    {
        set;
        get;
    }
    int this[int indeks]
    {
        get;
```

```
    }
}
```

Bu arayüzde parmak basmamız gereken önemli bir nokta var. Bu arayüzü kullanan sınıftaki indeksleyicinin sadece get bloğu olabilir veya hem get hem de set bloğu olabilir. Yani indesleyicilerdeki durum sahte özelliklerden biraz farklıdır. Arayüzü kullanan sınıftaki indeksleyici en az arayüzdeki set-get durumuna sahip olmalıdır.

Arayüzlerin uygulanması

Arayüzlerin uygulanması sınıf türetmeyle aynı şekilde yapılır. Örnek:

```
class A:IArayuz
{
   //IArayuz arayüzündeki bütün elemanları içermeli.
}
```

Bir sınıf birden fazla arayüzü kullanabilir. (Bu sınıf türetmede yoktu). Örnek:

```
class A:Arayuz1,Arayuz2
{
   //Hem Arayuz1 hem de Arayuz2 arayüzündeki bütün elemanları içermeli.
}
```

- Tabii ki arayüzü kullanan sınıfta sadece arayüzdeki elemanlar bulunur diye bir kural yoktur.
- Arayüzler de sınıf türetmeyle aynı şekilde birbirlerinden türetilebilir. Bu durumda yavru arayüz ana arayüzün taşıdığı bütün elemanları taşır.
- Sınıflardan farklı olarak arayüzleri birden fazla arayüzden türetebiliriz. Örnek:

```
interface Arayuz1
{
   int Metot1();
}
interface Arayuz2
{
   string Metot2();
}
interface Arayuz3:Arayuz1,Arayuz2
{
   double Metot4();
}
```

Burada Arayuz3'ü kullanan bir sınıf her üç metodu da içermelidir.

- Tıpkı sınıflardaki gibi new anahtar sözcüğünü kullanarak isim gizleme yapabiliriz. Örnek:

```
interface Arayuz1
{
   int Metot1();
}
interface Arayuz2:Arayuz1
{
   new int Metot1();
}
```

Burada aklınıza şöyle bir sorunun gelmesi doğaldır: Arayuz2'yi kullanan bir sınıfta Metot1()'in bildirimi yapıldığında geçerli bir bildirim olur mu? Bunun cevabı hayırdır. Arayuz2'nin Arayuz1'e ait metodu gizlemesi sonucu değiştirmez. Bu sorunu halletmek için sınıfımızı şöyle yazabiliriz.

```
class deneme:Arayuz2
{
   int Arayuz1.Metot1()
   {
      ...
   }
   int Arayuz2.Metot1()
   {
      ...
   }
}
```

Burada hem Arayuz1'in hem de Arayuz2'nin isteklerini yerine getirdik. Ancak şunu hatırlatmak isterim: eğer sınıfımız Arayuz2 arayüzünü kullanmasaydı programımız hata verirdi.

Arayüz nesneleri

Kulağa biraz itici gelse de bir arayüz türünden nesne oluşturulabilir. Arayüz nesneleriyle ilgili bilmeniz gereken en önemli şey şudur:
Tıpkı türetmedeki gibi bir arayüz nesnesine o arayüzü kullanan sınıf türünden nesne atanabilir. Bu durumda o arayüz nesnesinin gizli türü o sınıf olur ve o sınıfa ait üye elemana arayüz nesnesi üzerinden erişilebilir. Örnek:

```
using System;
interface arayuz
{
   int Metot();
}
class A:arayuz
{
   public int Metot()
   {return 0;}
   static void Main()
   {
      arayuz a;
      A s=new A();
      a=s;
      Console.WriteLine(a.Metot());
   }
}
```

Gördüğünüz gibi arayüz nesneleri oluştururken new anahtar sözcüğünü kullanmıyoruz.
NOT: Herhangi bir arayüzdeki üye elemanlar sınıfa public, static değil ve doğru geri dönüş tipinde geçirilmelidir.

Açık arayüz uygulama

C# dilinde var olan arayüzleri uygulamanın bir yolu daha vardır. Buna açık arayüz uygulama denir. Bunun avantajları şunlardır:

- Açık arayüz uygulama yöntemiyle istenirse sınıfa ait bazı üye elemanlara erişimi sınıf nesnelerine kapatırken, aynı üye elemanlara arayüz nesnesiyle erişimi mümkün kılabiliriz.
- Bir sınıfa birden fazla arayüz uygulandığında eğer arayüzlerde aynı isimli üye elemanlar varsa isim çakışmasının önüne geçebiliriz. Örnek program:

```
using System;
interface arayuz
{
   void Metot();
}
class sinif:arayuz
{
   void arayuz.Metot()
   {
      Console.WriteLine("Deneme");
   }
}
class mainMetodu
{
   static void Main()
   {
      sinif nesne=new sinif();
      ((arayuz)nesne).Metot();
   }
}
```

Burada sinif türünden olan nesne arayuz türüne dönüştürüldü ve arayuz nesnesiyle Metot() metoduna erişildi. Direkt nesne üzerinden erişilemezdi. Aynı programı şöyle de yazabilirdik:

```
using System;
interface arayuz
{
   void Metot();
}
class sinif:arayuz
{
   void arayuz.Metot()
   {
      Console.WriteLine("Deneme");
   }
}
class mainMetodu
{
   static void Main()
   {
```

```
        arayuz nesne=new sinif();
        nesne.Metot();
    }
}
```

Örnekler

Şimdi arayüzlerle ilgili öğrendiğimiz bilgileri pekiştirebileceğimiz karmaşık bir örnek yapalım.

```
using System;
using System.Collections;
class Koleksiyon:IEnumerable
{
    int[] Dizi;
    public Koleksiyon(int[] dizi)
    {
        this.Dizi=dizi;
    }
    IEnumerator IEnumerable.GetEnumerator()
    {
        return new ENumaralandırma(this);
    }
    class ENumaralandırma:IEnumerator
    {
        int indeks;
        Koleksiyon koleksiyon;
        public ENumaralandırma(Koleksiyon koleksiyon)
        {
            this.koleksiyon=koleksiyon;
            indeks=-1;
        }
        public void Reset()
        {
            indeks=-1;
        }
        public bool MoveNext()
        {
            indeks++;
            if(indeks<koleksiyon.Dizi.Length)
                return true;
            else
                return false;
        }
        object IEnumerator.Current
        {
            get
            {
                return(koleksiyon.Dizi[indeks]);
```

```
            }
        }
    }
}
class MainMetodu
{
    static void Main()
    {
        int[] dizi={1,2,3,8,6,9,7};
        Koleksiyon k=new Koleksiyon(dizi);
        foreach(int i in k)
            Console.Write(i+" ");
    }
}
```

Bu program kendi tanımladığımız herhangi bir sınıfın foreach döngüsüyle kullanılabilmesini sağlıyor. Yani `foreach(tur1 nesne1 in nesne2){}` deyiminde hem nesne1'in türünü (tur1) hem de nesne2'nin türünü kendi oluşturduğumuz türlerden (sınıf, yapı, vb.) biri yapabiliriz. Bu programda nesne2'nin kendi türümüzde olmasını sağlayacağız. Bu örnekten hemen sonraki örnekte de nesne1'in kendi türümüzden olmasını sağlayacağız. Programımıza dönecek olursak foreach döngüsünde nesne2'yi kendi türümüzden yapabilmemiz için bazı metotlar oluşturmamız gerekiyor. System.Collections isim alanındaki IEnumerable ve IEnumerator arayüzleri de bize bu metotları oluşturmaya zorluyor. Ayrıca programımızın ana hattına dikkat ederseniz iki sınıf iç içe geçmiş. C#'ta bu tür bir kullanım mümkündür. Lafı fazla uzatmadan bu programımızı satır satır incelemeye başlayalım.

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

Bu satırlarla System ve System.Collections isim alanlarındaki türlere direkt erişim hakkı elde ettik.

```
class Koleksiyon:IEnumerable
{
```

Bu satırlarla System.Collections isim alanındaki IEnumerable arayüzünü kullanan bir sınıf başlattık.

```
    int[] Dizi;
```

Bu satırla sınıfımıza (Koleksiyon) ait bir özellik bildirdik.

```
    public Koleksiyon(int[] dizi)
    {
        this.Dizi=dizi;
    }
```

Bu satırlarla sınıfımızın (Koleksiyon) yapıcı metodunu bildirdik. Yapıcı metodumuz int[] türünden bir parametre alıyor ve bu aldığı veriyi sınıfın bir özelliği olan Dizi'ye aktarıyor. Burada this anahtar sözcüğünün kullanılması zorunlu değildir ancak okunurluğu artırır.

```
    IEnumerator IEnumerable.GetEnumerator()
    {
        return new ENumaralandırma(this);
    }
```

Bu satırlarla sınıfımıza, IEnumerable arayüzündeki GetEnumerator() metodunu açık arayüz uygulama yöntemiyle geçiriyoruz. Metodun geri dönüş tipi IEnumerator arayüzü. Metodun gövdesinde ise ENumaralandırma sınıfının yapıcı metodu kullanılarak ENumaralandırma türünden bir nesne döndürülüyor. Buradan ENumaralandırma sınıfının yapıcı metodunun Koleksiyon sınıfı türünden bir nesne aldığını anlayabiliyoruz. Çünkü buradaki this anahtar sözcüğü bu GetEnumerator metoduna hangi nesne üzerinden erişildiğini temsil ediyor. Bu metot Koleksiyon sınıfında olduğuna göre bu metoda da bir Koleksiyon nesnesi üzerinden erişilmelidir. Burada bir terslik varmış gibi gözüküyor. O da metodun geri dönüş tipiyle geri döndürülülen verinin tipinin birbirine uymaması. Ancak programımızın sonraki kodlarına bakacak olursanız ENumaralandırma sınıfının IEnumerator arayüzünü kullandığını göreceksiniz. Dolayısıyla da bir IEnumerator tipinden nesneye bir ENumaralandırma nesnesi atanabilecek.

```
    class ENumaralandırma:IEnumerator
    {
```

Burada Koleksiyon sınıfının içinde ENumaralandırma sınıfını oluşturuyoruz ve bu sınıf da System.Collections isim alanındaki IEnumerator arayüzünü kullanıyor.

```
        int indeks;
        Koleksiyon koleksiyon;
```

Burada sınıfımıza (ENumaralandırma) iki tane özellik ekledik.

```
        public ENumaralandırma(Koleksiyon koleksiyon)
        {
           this.koleksiyon=koleksiyon;
           indeks=-1;
        }
```

Burada ENumaralandırma sınıfının yapıcı metodunu hazırladık. Yeri gelmişken belirtmek istiyorum. İç içe sınıflar iç içe gözükselerde aslında birbirinden bağımsızdır. İç içe sınıfların normal sınıflardan tek farkı içteki sınıfa dıştaki sınıfın dışından erişilmek istendiğinde görülür. Bu durumda içteki sınıfa `DisSinif.IcSinif` yazarak erişilebilir. Ancak tabii ki bu erişimin mümkün olabilmesi için iç sınıfın public olarak belirtilmesi gerekir.

```
        public void Reset()
        {
           indeks=-1;
        }
```

Burada sınıfımıza bir metot ekledik. Bu metodu sınıfımızın kullandığı IEnumerator arayüzü gerektiriyordu.

```
        public bool MoveNext()
        {
           indeks++;
           if(indeks<koleksiyon.Dizi.Length)
              return true;
           else
              return false;
        }
```

Burada sınıfımıza bir metot daha ekledik. Yine bu metodu da IEnumerator arayüzü gerektiriyordu.

```
        object IEnumerator.Current
        {
           get
```

```
            {
               return(koleksiyon.Dizi[indeks]);
            }
         }
```

Burada IEnumerator arayüzünün gerektirdiği bir sahte özelliği açık arayüz uygulama yöntemiyle hazırladık.

```
class MainMetodu
{
   static void Main()
   {
```

Artık programımızın çalışmaya başlayacağı kısmı yazmaya başlıyoruz.

```
      int[] dizi={1,2,3,8,6,9,7};
      Koleksiyon k=new Koleksiyon(dizi);
```

Dizimizi ve yeni bir Koleksiyon nesnesi oluşturduk.

```
      foreach(int i in k)
         Console.Write(i+" ");
```

Ve başardık. nesne2'yi kendi türümüz yaptık. Şimdi nesne1'i kendi türümüz yapacak programı yazalım:

```
using System;
class A
{
   public int Ozellik;
   public A(int a)
   {
      Ozellik=a;
   }
}
class Ana
{
   static void Main()
   {
      A[] dizi=new A[3];
      dizi[0]=new A(10);
      dizi[1]=new A(20);
      dizi[2]=new A(50);
      foreach(A i in dizi)
         Console.WriteLine(i.Ozellik);
   }
}
```

Gördüğünüz gibi kendi oluşturduğumuz sınıf türünden nesnelerle diziler oluşturabiliyoruz. Nasıl ki int[] ile int türündeki veriler bir araya geliyorsa bizim örneğimizde de A türünden nesneler bir araya geldi.

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Partial (kısmi) tipler

Şimdi daha önceden gördüğümüz bir şeyi hatırlatmak istiyorum. Kodlarımızı birden fazla dosyaya parçalayıp bu parçaları komut satırında hep birlikte derleyebiliriz. Yani programımızın bir sınıfını bir dosyaya, başka bir sınıfını başka bir dosyaya koyup komut satırında toplu derleme yaparsak bütün bu dosyalar sanki bir dosyadaymış gibi işlem görecektir. Aynı olayı isim alanı düzeyinde de yapabiliriz. Yani bir isim alanını bir dosyaya, başka bir isim alanını bir dosyaya koyup komut isteminde toplu derleme yaparsak yine iki isim alanı da aynı dosyadaymış gibi işlem görecektir. Ancak bu durum metot düzeyinde mümkün değildir. Çünkü metotlar mutlaka bir sınıfa ait olmalıdır. Aklınıza şöyle bir fikir gelebilir: "Metotları farklı dosyalarda aynı isimli sınıflara koyalım!". Ancak ne yazıkki böyle bir durumda dosyalar birleştirildiğinde derleyici aynı isimli sınıfın iki kez bildirildiğini anlayacak ve hata verecektir. İşte burada kısmi türler devreye giriyor. Derleyicinin aynı isimli türler (sınıf, yapı, arayüz, ...) gördüğünde hata vermek yerine türlerdeki üye elemanları tek isim altında birleştirmesini sağlamak için kısmi tipler kullanılır. Bir türün kısmi olduğunu belirtmek için `partial` anahtar sözcüğü kullanılır. Örnek:
bir.cs dosyası

```
partial class sinif
{
    ...
}
```

iki.cs dosyası

```
partial class sinif
{
    ...
}
```

Komut isteminden `csc bir.cs iki.cs` komutunu verirsek bu iki dosyadaki sınıfların içeriği birleştirilecektir. Kısmi türlerle ilgili bazı ufak bilgiler:

- Birleştirilmesini istediğimiz bütün aynı isimli türleri partial olarak bildirmeliyiz. Yalnızca birisini partial olarak bildirmek yeterli değildir.
- Sınıflar, yapılar ve arayüzler partial olarak bildirilebilir.
- Kısmi türlerden biri sealed ya da abstract anahtar sözcüğüyle belirtilmişse diğerinin de belirtilmesine gerek yoktur.
- Partial türlerin partial olarak bildirildi diye illaki başka bir dosyayla ilişkilendirilmesine gerek yoktur.
- Partial türler minimum üye eleman düzeyinde iş görürler. Yani aynı metodun gövdesinin bir kısmını bir dosyada, başka bir kısmını başka bir dosyada yazmak partial türler ile de mümkün değildir.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

İstisnai durum yakalama mekanizması

Muhtemelen şu ana kadar fark etmişsinizdir. Ancak yine de söylemek istiyorum. Programlarımızdaki hatalar ikiye ayrılır:

1. Derleme zamanı hataları
2. Çalışma zamanı hataları

Programımızda bir derleme zamanı hatası oluşursa derleme işlemi olmaz. Genellikle derleme zamanı hataları yanlış söz diziminden (sintaks) kaynaklanır ve tespiti kolaydır. Çalışma zamanı hataları ise daha çok mantıksal yanlışlıklardan kaynaklanır ve tespiti nispeten daha zordur. Örneğin kullanıcının girdiği iki sayıyı toplayan bir program yazdığımızı düşünelim. Kullanıcı sayıların yerine herhangi bir harf girerse programımız çalışma zamanı hatası verir ve kendisini sonlandırır. Böyle bir durumu engellemek için toplama işlemini yapmadan önce kullanıcının girdiği veriyi bir kontrolden geçirebiliriz. Örneğin kullanıcının girdiği stringin içinde foreach döngüsü yardımıyla dolaşıp herhangi bir rakam olmayan karakter varsa toplama işleminin yapılmamasını sağlayabiliriz. Ancak takdir edersiniz ki bu çok zahmetli bir yöntemdir. Üstelik bu kadar basit bir program için bu kadar zahmetli bir kod yazdığımızı düşünürsek büyük ölçekli projelerde zamanımızın çoğunu hata yakalamaya ayırmamız gerekirdi. İşte istisnai durum yakalama mekanizması böyle bir durumda zamanımızı büyük oranda kısaltır. İstisnai durum yakalama mekanizması programımızın bir çalışma zamanı hatasıyla karşılaştığında hata verip kendini sonlandırması yerine daha farklı işlemler yapılabilmesini sağlar. Şimdi kod yazmaya başlayabiliriz.

Hatanın türünden bağımsız hata yakalama

```
using System;
class deneme
{
   static void Main()
   {
      string a=Console.ReadLine();
      string b=Console.ReadLine();
      int toplam=Int32.Parse(a)+Int32.Parse(b);
      Console.WriteLine(toplam);
   }
}
```

Bu programın şimdiye kadar yazdığımız programlardan farkı yok. Kullanıcının girdiği değerlerden en az biri sayı değilse program hata verip sonlanacaktır. Bunu önlemek için programı şöyle yazabiliriz:

```
using System;
class deneme
{
   static void Main()
   {
      try
      {
         string a=Console.ReadLine();
         string b=Console.ReadLine();
         int toplam=Int32.Parse(a)+Int32.Parse(b);
         Console.WriteLine(toplam);
      }
      catch
      {
         Console.WriteLine("Sayı girmelisiniz!");
      }
   }
}
```

Bu programda try bloğunda herhangi bir çalışma zamanı hatasıyla karşılaşılır ise catch bloğundaki satırlar çalıştırılır ve akış catch bloğunun dışından devam eder. Eğer try bloğunda herhangi bir çalışma zamanı hatasıyla karşılaşılmazsa catch bloğundaki satırlar çalıştırılmaz ve programın akışı catch bloğunun dışından devam eder. try ve catch blokları aynı blok içinde olmalıdır. Ayrıca try'dan hemen sonra catch gelmelidir. Başka bir örnek:

```
using System;
class deneme
{
   static void Main()
   {
      try
      {
         string a=Console.ReadLine();
         string b=Console.ReadLine();
         int toplam=Int32.Parse(a)+Int32.Parse(b);
         Console.WriteLine(toplam);
      }
      catch
      {
         Console.WriteLine("Sayı girmelisiniz!");
      }
      finally
      {
         Console.WriteLine("Program sonlandı!");
      }
   }
}
```

Burada finally bloğunda hata oluşsa da oluşmasa da çalıştırılması istenen satırlar yazılır. finally bloğunu belirtmek zorunda değiliz. Ancak try ve catch blokları istisnai durum yakalama mekanizmasının çalışması için mutlaka gereklidir. Programımızın hata vermemesi için try bloğundan sonra catch ya da finally blokları gelmelidir. Ancak try bloğundan sonra finally bloğu gelirse istisnai durum yakalama mekanizması çalışmayacağı için böyle bir kullanım tavsiye edilmez.

Hatanın türüne bağlı hata yakalama

Farkındaysanız şimdiye kadar try bloğunda herhangi bir hata gerçekleşmesi durumunda catch bloğundaki satırların çalışmasını sağladık. Şimdi hatanın türüne göre farklı catch bloklarının çalıştırılmasını sağlayacağız. Ancak bunun için bazı ön bilgiler almanız gerekiyor.

C#'ta bütün hata türleri System isim alanındaki birer sınıfla temsil edilir ve o hata gerçekleştiğinde o sınıf türünden bir nesne fırlatılır. Şimdi konuyu anlamak için en baştaki örneğimizi biraz değiştirelim:

```
using System;
class deneme
{
   static void Main()
   {
      try
      {
         string a=Console.ReadLine();
         string b=Console.ReadLine();
         int toplam=Int32.Parse(a)+Int32.Parse(b);
         Console.WriteLine(toplam);
      }
      catch(FormatException nesne)
      {
         Console.WriteLine("Sayı girmelisiniz!");
      }
   }
}
```

System isim alanındaki FormatException metotlara yanlış formatta parametre verilmesiyle ilgilenir. Dolayısıyla da örneğimizde kullanıcı rakam yerine harf girerse Parse() metodu gerekli dönüşümü yapamayacağı için catch bloğu çalışacaktır. Başka bir örnek:

```
using System;
class deneme
{
   static void Main()
   {
      try
      {
         string a=Console.ReadLine();
         string b=Console.ReadLine();
         int toplam=Int32.Parse(a)+Int32.Parse(b);
         Console.WriteLine(toplam);
      }
```

```
        catch(FormatException nesne1)
        {
            Console.WriteLine("Sayı girmelisiniz!");
        }
        catch(IndexOutOfRangeException nesne2)
        {
            Console.WriteLine("Dizi sınırları aşıldı!");
        }
    }
}
```

IndexOutOfRangeException dizi sınırlarının aşımıyla ilgilenen bir sınıftır. Gördüğünüz gibi birden fazla catch bloğu tasarlayıp farklı hatalara göre programımızın gidişatını yönlendirebiliyoruz. Şimdi örneğimizi biraz daha geliştirelim:

```
using System;
class deneme
{
    static void Main()
    {
        try
        {
            string a=Console.ReadLine();
            string b=Console.ReadLine();
            int toplam=Int32.Parse(a)+Int32.Parse(b);
            Console.WriteLine(toplam);
        }
        catch(FormatException nesne)
        {
            Console.WriteLine("Şu hata meydana geldi: "+nesne.Message);
        }
    }
}
```

Hata sınıfları sonuçta bir sınıf olduğuna göre bu sınıfların kendine özgü üye elemanları olmalı. İşte Message özelliği hemen hemen bütün hata sınıflarında bulunur ve hatanın ne olduğunu belirten bir string döndürür (tabii ki İngilizce). Message gibi bütün hata sınıflarında bulunan bazı özellikler daha vardır:
Source Hata nesnesinin gönderildiği uygulama ya da dosyanın adıdır. (string)
StackTrace Hatanın oluştuğu yer hakkında bilgi verir. (string)
HelpLink Hatayla ilgili olan yardım dosyasını saklar. Tabii ki bunu programcı belirler. (string)
TargetSite İstisnai durumu fırlatan metot ile ilgili bilgi verir. (MethodBase)
InnerException catch bloğu içerisinden bir hata fırlatılırsa catch bloğuna gelinmesine yol açan istisnai durumun Exception nesnesidir.
ToString() Bu metot ilgili hataya ilişkin hata metninin tamamını string olarak döndürür.
Yukarıdaki üye elemanların tamamı static değildir. Şimdi çeşitli örnekler yapalım:

```
using System;
class deneme
{
```

```
    static void Main()
    {
        try
        {
            Metot();
        }
        catch(IndexOutOfRangeException nesne)
        {
            Console.WriteLine("Dizi sınırları aşıldı.");
        }
    }
    static void Metot()
    {
        int[] a=new int[2];
        Console.WriteLine(a[3]);
    }
}
```

Bu program ekrana `Dizi sınırları aşıldı.` yazacaktır. Yani bir metot başka bir metodun içinde kullanıldığı zaman adeta kullanılan metot yeni metodun içine kopyalanmış gibi işlem yapıldı. Başka bir örnek:

```
using System;
class deneme
{
    static void Main()
    {
        try
        {
            Metot();
        }
        catch(IndexOutOfRangeException nesne)
        {
            Console.WriteLine("Metodu kullananda hata yakalandı");
        }
    }
    static void Metot()
    {
        try
        {
            int[] a=new int[2];
            Console.WriteLine(a[3]);
        }
        catch(IndexOutOfRangeException nesne)
        {
            Console.WriteLine("Metodun kendisinde hata yakalandı.");
        }
    }
}
```

Bu program ekrana `Metodun kendisinde hata yakalandı.` yazacaktır. Yani hem metodun kendisinde hem de metodu kullanan metotta hata yakalama mekanizması kurulmuşsa hata durumunda metodun kendisindeki mekanizma çalışır. Mekanizma yalnızca herhangi birisinde olsaydı o metoda ait mekanizma çalışacaktı. Başka bir örnek:

```
using System;
class deneme
{
   static void Main()
   {
      try
      {
         int[] a=new int[2];
         Console.WriteLine(a[3]);
      }
      catch(IndexOutOfRangeException)
      {
         Console.WriteLine("Dizi sınırları aşıldı");
      }
   }
}
```

Gördüğünüz gibi catch'e parametre verirken illaki nesne oluşturulmasına gerek yok. Sadece sınıfın adı yazılması yeterli. Ancak tabii ki bu durumda sınıfa ait üye elemanlara erişemeyiz. Başka bir örnek:

```
using System;
class deneme
{
   static void Main()
   {
      for(;;)
      {
         try
         {
            Console.Write("Lütfen çıkmak için 0 ya da 1 girin: ");
            int a=Int32.Parse(Console.ReadLine());
            int[] dizi=new int[2];
            Console.WriteLine(dizi[a]);
            break;
         }
         catch
         {
            continue;
         }
      }
   }
}
```

Bu programda istisnai durum mekanizmasını son derece yaratıcı bir şekilde kullandık. Programı biraz inceleyince ne yapmak istediğimizi anlayacaksınız.

throw anahtar sözcüğü

Şu ana kadar hata nesnelerinin atılması hata gerçekleştiğinde otomatik olarak yapılıyordu. Şimdi ise throw anahtar sözcüğü sayesinde bunu manuel olarak yapabileceğiz. Yani tabiri caizse hata gerçekleşmemişse bile sistemin hata gerçekleşmiş gibi davranmasını sağlayacağız. throw anahtar sözcüğünü try bloğunun içinde kullanmalıyız. Aksi bir durumda program hata vermese de işimize yaramaz. Bir hata nesnesi throw anahtar sözcüğü yardımıyla şöyle fırlatılabilir:

```
throw new IndexOutOfRangeException("Dizinin sınırları aşıldı");
```

veya

```
IndexOutOfRangeException nesne=new IndexOutOfRangeException("Dizinin
sınırları aşıldı");
throw nesne;
```

Birincisinde Message özelliğinin değeri "Dizinin sınırları aşıldı" stringi olan yeni bir IndexOutOfRangeException nesnesi fırlatılıyor. İkincisinde ise önce nesne oluşturuluyor sonra bu nesne fırlatılıyor. Şimdi bunları az önceki üzerinde görelim:

```
using System;
class deneme
{
   static void Main()
   {
      for(;;)
      {
         try
         {
            Console.Write("Lütfen çıkmak için 0 ya da 1 girin: ");
            string a=Console.ReadLine();
            if(a=="0"||a=="1")
               break;
            else
               throw new IndexOutOfRangeException("Devam ediliyor");
         }
         catch(IndexOutOfRangeException nesne)
         {
            Console.WriteLine(nesne.Message);
            continue;
         }
      }
   }
}
```

Şimdi aynı örneği throw anahtar sözcüğünün değişik formuyla yapalım:

```
using System;
class deneme
{
   static void Main()
   {
      for(;;)
      {
         try
         {
            Console.Write("Lütfen çıkmak için 0 ya da 1 girin: ");
            string a=Console.ReadLine();
            if(a=="0"||a=="1")
               break;
            else
            {
               IndexOutOfRangeException nesne=new
IndexOutOfRangeException("Başa dönüldü");
               nesne.HelpLink="http://tr.wikibooks.org"; //Gördüğünüz
gibi bu yöntemle nesnenin özelliklerini değiştirebiliyoruz.
               throw nesne;
            }
         }
         catch(IndexOutOfRangeException nesne)
         {
            Console.WriteLine(nesne.Message);
            continue;
         }
      }
   }
}
```

İstisnai durum sınıflarında türeme

Aslında bütün istisnai durum sınıfları Exception sınıfından türetilmiştir. Ancak Exception sınıfındaki üye elemanlar sanal olarak bildirilmiş ve istisnai durum sınıfları da bu üye elemanları override etmiştir. Benzer şekilde çoklu türetmenin yapıldığı sınıflar da mevcuttur. Örneğin:

- SystemException sınıfı Exception sınıfından türemiştir.
- IOException sınıfı SystemException sınıfından türemiştir.
- FileNotFoundException sınıfı IOException sınıfından türemiştir.

Kalıtım konusunda da gördüğümüz gibi yavru sınıf nesneleri ana sınıf nesneleri yerine kullanılabilir. Dolayısıyla bir catch bloğu IOException hatalarını yakalıyorsa aynı zamanda FileNotFoundException hatalarını da yakalar. Ancak tabii ki hem IOException hem de FileNotFoundException catch'leri oluşturulmuşsa FileNotFoundException'ın catch'i çalıştırılacaktır.

NOT1: Birden fazla catch bloğu oluşturduğumuz durumlarda catch blokları art arda gelmelidir. Yani araya herhangi başka kod giremez.

NOT2: Bir C# programında aşağıdaki iki catch bloğu varsa ikincisi (parametreli olan) çalışır.

```
catch
{
}
catch(Exception nesne)
{
}
```

Ancak başka dillerden alınmış bir COM parçacığı söz konusuysa birincisi çalışır. COM parçacıkları çok ayrıntılı ve karmaşık olduğundan ve muhtemelen işiniz düşmeyeceğinden birinci olasılığı özümsemeniz yeterlidir.
NOT3: Birden fazla art arda gelen catch bloğunun parametreleri aynıysa programımız derlenmez.

İç içe geçmiş try blokları

try-catch-finally yapımızın şöyle olduğunu düşünelim:

```
try
{
   //A
   try
   {
      //B
   }
   catch
   {
      //C
      //İçteki catch bloğu
   }
   finally
   {
      //D
      //İçteki finally bloğu
   }
}
catch
{
   //Dıştaki catch bloğu
}
finally
{
   //Dıştaki finally bloğu
}
```

1. A bölgesinden bir hata fırlatıldığı zaman dıştaki catch bloğu tarafından yakalanır.
2. B bölgesinden bir hata fırlatıldığı zaman önce içteki catch bloğuna bakılır. Uygun tür varsa içteki catch bloğu çalıştırılır. Uygun tür yoksa dıştaki catch bloğu çalıştırılır.
3. C veya D bölgesinden bir hata fırlatıldığı zaman dıştaki catch bloğu tarafından yakalanır.
4. Tabii ki finally blokları bütün durumlarda çalıştırılır.
5. Benzer kurallar ikiden fazla iç içe geçmiş try blokları için de geçerlidir. Yani uygun tür varsa en içteki try bloğundaki hata en içteki catch tarafından yakalanır. Uygun tür yoksa basamak basamak dışa çıkılır.

İstisnai durum sınıfları

Şimdiye kadar birkaç istisnai durum sınıfı gördük. Şimdi bunları ve sık kullanılan birkaç sınıfı daha göreceğiz:
System.OutOfMemoryException Programın çalışması için yeterli bellek kalmadıysa fırlatılır.
System.StackOverflowException Stack bellek bölgesinin birden fazla metot için kullanılması durumunda fırlatılır. Genellikle kendini çağıran metotların hatalı kullanılmasıyla meydana gelir.
System.NullReferenceException Bellekte yer ayrılmamış bir nesne üzerinden sınıfın üye elemanlarına erişmeye çalışırken fırlatılır.
System.OverflowException Bir veri türüne kapasitesinden fazla veri yüklemeye çalışılırken fırlatılır.
System.InvalidCastException Tür dönüştürme operatörüyle geçersiz tür dönüşümü yapılmaya çalışıldığında fırlatılır.
System.IndexOutOfRangeException Bir dizinin olmayan elemanına erişilmeye çalışılırken fırlatılır.
System.ArrayTypeMismatchException Bir dizinin elemanına yanlış türde veri atanmaya çalışılırken fırlatılır.
System.DividedByZero Sıfıra bölme yapıldığı zaman fırlatılır.
System.ArithmeticException DividedByZero ve OverflowException bu sınıftan türemiştir. Hemen hemen matematikle ilgili tüm istisnaları yakalayabilir.
System.FormatException Metotlara yanlış formatta parametre verildiğinde fırlatılır.

Kendi istisnai durum sınıflarımızı oluşturmak

.Net Framework kütüphanesinde onlarca istisnai durum sınıfı vardır. Ancak istersek kendi istisnai durum sınıflarımızı da oluşturabiliriz. Kendi oluşturduğumuz sınıfları ApplicationException sınıfından türetiriz. Bu sınıf Exception sınıfından türemiştir. Dolayısıyla kendi oluşturduğumuz istisnai durum sınıflarına da Exception sınıfındaki Message, HelpLink gibi üye elemanlar otomatik olarak gelecektir. Bu üye elemanları istersek override edebilir, istersek olduğu gibi bırakabiliriz. İstersek de kendi üye elemanlarımızı yazabiliriz. İsterseniz şimdi örneğimize başlayalım. Örneğimizde öğrencilerin notlarını tutan bir sınıf tasarlayalım. Sonra bu sınıfın içinde iki tane hata sınıfı oluşturalım. Birinci sınıf öğrencinin notu 100'den büyükse, ikinci sınıf 0'dan küçükse hata verdirsin. Şimdi örneğimize başlayabiliriz:

```
using System;
class Notlar
{
   private int mNot;
   public int Not
   {
      get{return mNot;}
      set
      {
         if(value>100)
            throw new FazlaNotHatasi();
         else if(value<0)
            throw new DusukNotHatasi();
         else
            mNot=value;
      }
   }
   public class FazlaNotHatasi:ApplicationException
   {
      override public string Message
```

```
        {
            get{return "Not 100'den büyük olamaz.";}
        }
    }
    public class DusukNotHatasi:ApplicationException
    {
        override public string Message
        {
            get{return "Not 0'dan küçük olamaz.";}
        }
    }
}
class Ana
{
    static void Main()
    {
        try
        {
            Notlar a=new Notlar();
            Console.Write("Not girin: ");
            int b=Int32.Parse(Console.ReadLine());
            a.Not=b;
            Console.WriteLine("Notunuzu başarıyla girdiniz.");
        }
        catch(Exception nesne)
        {
            Console.WriteLine(nesne.Message);
        }
    }
}
```

Programı inceleyince muhtemelen kendi istisnai durum sınıflarımızı nasıl yaptığımızı anlayacaksınız. Ancak yine de özet bir bilgi vermek istiyorum. Hata sınıf(lar)ımızı asıl sınıfımızın içine koyuyoruz. Asıl sınıfımız içinde herhangi bir anda bu hata sınıflarımız türünden nesne fırlatabiliyoruz. Ayrıca bu programdaki catch'in Exception türünden nesne alması, ancak bu nesne üzerinden yavru sınıfa ait özelliğe erişilmesi geçen bölümde gördüğümüz sanal üye elemanlara çok iyi bir örnek. Şimdi örneğimizi biraz değiştirelim:

```
using System;
class Notlar
{
    //Buraya az önceki Notlar sınıfının aynısı gelecek.
}
class Ana
{
    static void Main()
    {
        try
        {
```

```
            Notlar a=new Notlar();
            Console.Write("Not girin: ");
            int b=Int32.Parse(Console.ReadLine());
            a.Not=b;
            Console.WriteLine("Notunuzu başarıyla girdiniz.");
        }
        catch(Notlar.FazlaNotHatasi nesne)
        {
            Console.WriteLine(nesne.Message);
        }
    }
}
```

Bu programda fazla not hataları yakalanacak ancak düşük not hataları yakalanmayacaktır. Eğer düşük not hatası oluşursa program çalışma zamanında şunun gibi bir uyarı verip kendini sonlandıracaktır.

```
Unhandled Exception: Notlar+DusukNotHatasi: Not 0'dan küçük olamaz.
   at Notlar.set_Not(Int32 value)
   at Ana.Main()
```

Buradan hata fırlattıran sınıfın Notlar sınıfındaki DusukNotHatasi sınıfı olduğunu, bu sınıfa ait Message özelliğinin değerinin "Not 0'dan küçük olamaz." olduğunu, hatanın Notlar sınıfındaki Not sahte özelliğine Int32 türünden veri atanırken oluştuğunu ve son tahlilde hatanın Ana sınıfındaki Main() metodunda çalışılırken oluştuğunu anlıyoruz. Şimdi şöyle basit bir örnek yapalım:

```
using System;
class deneme
{
    static void Main()
    {
        int a=Int32.Parse(Console.ReadLine());
        Console.WriteLine(3/a);
    }
}
```

Kullanıcının bu programda 0 girdiğini varsayarsak program çalışma zamanı hatasında şöyle bir çıktı verir:

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
   at deneme.Main()
```

Buradan hata fırlattıran sınıfın System isim alanındaki DivideByZeroException sınıfı olduğunu, bu sınıfın Message özelliğinin değerinin "Attempted to divide by zero." olduğunu ve hatanın deneme sınıfındaki Main() metodunda oluştuğunu anlıyoruz. Benzer şekilde çalışarak yeni hata sınıfları keşfedebilirsiniz.
NOT: Hatayı yakalayan birden fazla catch bloğu varsa ve bu catch bloklarının parametreleri arasında türeme varsa bu durumda catch bloklarının sıralamasının yavru sınıftan ana sınıfa doğru olması gerekir. Örneğin şu program derleme zamanı hatası verir:

```
using System;
class deneme
{
    static void Main()
```

```
    {
        try
        {
            int a=Int32.Parse(Console.ReadLine());
            Console.WriteLine(3/a);
        }
        catch(Exception nesne)
        {
            Console.WriteLine("Hata var!");
        }
        catch(DivideByZeroException nesne)
        {
            Console.WriteLine("Hata var!");
        }
    }
}
```

NOT: .Net kütüphanesindeki istisnai durum sınıflarının yapıcı metodu bir parametre alıyordu ve bu parametrede ilgili nesnenin Message özelliğinin değeri belirleniyordu. Tabii ki siz de kendi istisnai durum sınıflarınızın yapıcı metodu olmasını ve istediğiniz sayıda parametre almasını sağlayabilirsiniz.

NOT: try-catch ikilisiyle bir istisnai durum yakalanmadığı zaman ekrana yazılan hata yazısı ilgili istisnai durum sınıfının ToString() metodunun ürettiği değerdir. Kendi istisnai durum sınıflarımızda bu metodu override ederek yakalanmayan hatalarda ekrana çıkacak yazıyı değiştirebiliriz.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Temsilciler

Şimdiye kadar öğrendiklerimiz türler sınıflar, yapılar, enumlar ve arayüzlerdi. Şimdi temsilci isminde yeni bir tür daha öğreneceğiz. Temsilcilerin nesneleri oluşturulabilir ve bu nesneler metotları temsil ederler. Temsilci bildirimi `delegate` anahtar sözcüğü ile yapılır. Bir temsilci bildirim taslağı şu şekildedir:

```
delegate GeriDönüşTipi TemsilciİsmiAdı(parametreler);
```

Bu taslağa göre örnek bir temsilci bildirimi şöyle yapılır:

```
delegate int temsilci(int a,string b);
```

Temsilciler erişim belirleyicisi alabilir. Temsilciler genellikle programlarımıza belirli bir ifadeyle birden fazla metodu çağırabilme yeteneği vermek için kullanılır. Örnek program:

```
using System;
class deneme
{
   delegate void temsilci();
   static void Metot1()
   {
      Console.WriteLine("Burası Metot1()");
   }
   static void Metot2()
   {
      Console.WriteLine("Burası Metot2()");
   }
   static void Main()
   {
      temsilci nesne=new temsilci(Metot1);
      nesne();
      nesne=new temsilci(Metot2);
      nesne();
   }
}
```

Bu program ekrana alt alta `Burası Metot1()` ve `Burası Metot2()` yazacaktır. Yani bir sınıf nesnesi oluşturur gibi bir temsilci nesnesi oluşturuyoruz. Yapıcı metoda parametre verir gibi temsilci nesnesinin hangi metodu temsil edeceğini belirtiyoruz. Sonra bu temsilci nesnesi üzerinden temsil edilen metodu çalıştırıyoruz. Daha sonra temsilci nesnesinin değerini yani hangi metodu temsil ettiğini değiştirebiliyoruz. Burada dikkat etmemiz gereken nokta temsilci nesnesinin geri dönüş tipi ve parametre olarak kendiyle uyumlu metodu temsil edebileceğidir. Erişim belirleyicisi, static olup olmama durumu, vs. bu uyuma dâhil değildir. Bu örneğimizde parametre almayan ve geri dönüş tipi void olan temsilcimiz her iki metodu da temsil edebildi. Şimdi daha karmaşık bir örnek yapalım. Örneğimizde her satırın açıklaması yorum olarak verilmiştir.

```
using System;
class Temsilciler
{
   public delegate void KomutMetodu(); //Geri dönüş tipi void olan ve
parametre almayan bir temsilci bildirdik.
```

```
    public struct KomutYapisi //KomutYapisi isminde bir yapı başlattık.
    {
       public KomutMetodu KomutMetot; //Yapımıza geri dönüş tipi
KomutMetodu temsilcisi olan bir özellik ekledik.
       public string Komut; //Yapımıza bir özellik daha ekledik.
    }
    public static void Komut1() //Sınıfımıza bir metot ekledik.
    {
       Console.WriteLine("Komut1 çalıştı.");
    }
    public static void Komut2() //Sınıfımıza bir metot daha ekledik.
    {
       Console.WriteLine("Komut2 çalıştı.");
    }
    public static void Komut3() //Sınıfımıza bir metot ekledik.
    {
       Console.WriteLine("Komut3 çalıştı.");
    }
    public static void Komut4() //Sınıfımıza bir metot ekledik.
    {
       Console.WriteLine("Komut4 çalıştı.");
    }
    static void Main()
    {
       KomutYapisi[] komutlar=new KomutYapisi[4]; //KomutYapisi
nesnelerinden oluşan bir dizi oluşturduk. (4 elemanlı)
       komutlar[0].Komut="komut1"; //komutlar[0] nesnesinin Komut
özelliğine değer atadık.
       komutlar[0].KomutMetot=new KomutMetodu(Komut1);
 //Artık komutlar[0] nesnesinin KomutMetot özelliği Komut1'i temsil
ediyor. Aynı durumlar diğer komutlar için de geçerli.
       komutlar[1].Komut="komut2";
       komutlar[1].KomutMetot=new KomutMetodu(Komut2);
       komutlar[2].Komut="komut3";
       komutlar[2].KomutMetot=new KomutMetodu(Komut3);
       komutlar[3].Komut="komut4";
       komutlar[3].KomutMetot=new KomutMetodu(Komut4);
       Console.Write("Komut girin: ");
       string GirilenKomut=Console.ReadLine();
       for(int i=0;i<komutlar.Length;i++)//komutlar dizisi içinde dolaşmaya çıkıyoruz.
          if(GirilenKomut==komutlar[i].Komut)
             komutlar[i].KomutMetot();
    }
 }
```

Bu programdaki KomutYapisi yapısı hem komutu string olarak hem de komutun temsilcisini içeriyor. Bu yapının dört tane nesnesi oluşturuluyor. Bu nesnelerin üye elemanlarının her birine uygun değerler atanıyor. Sonra kullanıcının girdiği komut her bir string komutla karşılaştırılıyor. Eğer eşleşen olursa ilgili temsilci çalıştırılıyor. Bu

tür bir örnek komutlarla çalışan DOS gibi programlar için son derece uygundur.
NOT: Temsilciler sınıfların taşıdığı özelliklerin bazılarını taşırlar. Örneğin bir metot parametre olarak bir temsilci tipinden nesne kabul edebilir veya bir türün bir özelliği bir temsilci tipinden olabilir.

Çoklu temsilciler

Şimdiye kadar bir temsilci nesnesi aynı anda yalnızca bir tane metodu temsil edebiliyordu. Artık bir temsilci nesnesinin aynı anda birden fazla metodu temsil edebilmesini sağlayacağız. Bu sayede temsilci nesnesi çağrıldığında temsil ettiği bütün metotlar sırasıyla çalıştırılacaktır. Bir temsilci nesnesinin birden fazla metodu temsil etmesini sağlamak için += ve -= operatörleri kullanılır. += operatörü temsilci nesnesine metot eklemek için, -= operatörü ise temsilci nesnesinden metot çıkarmak için kullanılır. Aslında aynı iş için + ve - operatörlerini de kullanabilirdik. Ancak += ve -= bu işi biraz olsun kolaylaştırıyor. Örnek:

```
using System;
class deneme
{
   delegate void temsilci();
   static void Metot1()
   {
      Console.WriteLine("Metot1 çağrıldı.");
   }
   static void Metot2()
   {
      Console.WriteLine("Metot2 çağrıldı.");
   }
   static void Metot3()
   {
      Console.WriteLine("Metot3 çağrıldı.");
   }
   static void Main()
   {
      temsilci nesne=null;
      nesne+=new temsilci(Metot2);
      nesne+=new temsilci(Metot1);
      nesne+=new temsilci(Metot3);
      nesne();
      Console.WriteLine("***");
      nesne-=new temsilci(Metot1);
      nesne();
   }
}
```

Bu programın ekran çıktısı şöyle olacaktır:

```
Metot2 çağrıldı.
Metot1 çağrıldı.
Metot3 çağrıldı.
***
Metot2 çağrıldı.
```

```
Metot3 çağrıldı.
```

Yani metotların çalışma sırası metotların temsilci nesnesine eklenme sırasıyla aynıdır. Eğer programda Metot1 temsilci nesnesine tekrar eklenirse bu sefer Metot1() en sona geçecektir.
NOT:

```
nesne+=new temsilci(Metot2);
nesne-=new temsilci(Metot1);
```

satırlarıyla

```
nesne=nesne+new temsilci(Metot2);
nesne=nesne-new temsilci(Metot1);
```

satırları denktir.

Delegate sınıfı

Kendi yarattığımız temsilciler gizlice System isim alanındaki Delegate sınıfından türer. Dolayısıyla kendi yarattığımız temsilcilerin nesneleri üzerinden bu sınıfın static olmayan üye elemanlarına erişebiliriz. Ayrıca Delegate sınıfı özet bir sınıf olduğu için `new` anahtar sözcüğüyle bu sınıf türünden nesne yaratamayız. Yalnızca kendi yarattığımız temsilci nesneleri üzerinden bu sınıfın üye elemanlarına erişebiliriz. Şimdi isterseniz Delegate sınıfının bazı üye elemanlarını görelim.:
GetInvocationList() Bu metot bir çoklu temsilcideki bütün metotları bir System.Delegate dizisine aktarır.
DynamicInvoke(object[] parametreler) Bu metot bir temsilcinin temsil ettiği metodu çağırmak için kullanılır. GetInvocationList() metodu ile çoklu bir temsilcinin metotları bir Delegate dizisine atandığı zaman bu dizinin elemanlarını normal yollarla çalıştırmak mümkün olmaz. İşte DynamicInvoke() metodu bu dizinin elemanlarını çalıştırmak için kullanılabilir. Şimdi isterseniz GetInvocationList() ve DynamicInvoke() metotlarını bir örnek üzerinde görelim:

```
using System;
class deneme
{
   delegate void temsilci();
   static void Metot1()
   {
      Console.WriteLine("Burası Metot1()");
   }
   static void Metot2()
   {
      Console.WriteLine("Burası Metot2()");
   }
   static void Main()
   {
      temsilci nesne=null;
      nesne+=new temsilci(Metot1);
      nesne+=new temsilci(Metot2);
      Delegate[] dizi=nesne.GetInvocationList();
      dizi[0].DynamicInvoke();
      dizi[1].DynamicInvoke();
   }
```

```
}
```

Bu program alt alta `Burası Metot1()` ve `Burası Metot2()` yazacaktır. Eğer Delegate dizisindeki temsilci nesnelerinin temsil ettiği metotların parametreleri varsa bu parametreler DynamicInvoke() metoduna object dizisi olarak verilmelidir.

Combine(Delegate[] temsilciNesneleri)

Combine(Delegate temsilciNesnesi1, Delegate temsilciNesnesi2)

Birinci metot temsilciNesneleri adlı Delegate dizisindeki tüm metotları bir çoklu temsilci nesnesi olarak tutar. İkinci metot ise temsilciNesnesi1 ve temsilciNesnesi2 temsilci nesnelerindeki tüm metotları arka arkaya ekleyip bir temsilci nesnesi olarak tutar. Her iki metot da statictir. Örnek:

```
//Diğer kısımlar yukarıdaki programın aynısı. (Burası Main() metodu)
temsilci nesne=null;
nesne+=new temsilci(Metot1);
nesne+=new temsilci(Metot2);
Delegate[] dizi=nesne.GetInvocationList();
Delegate nesne2=Delegate.Combine(dizi);
nesne2.DynamicInvoke();
```

Şimdi ikinci metodu örnekleyelim:

```
//Diğer kısımlar yukarıdaki programın aynısı. (Burası Main() metodu)
temsilci nesne1=null;
nesne1+=new temsilci(Metot1);
nesne1+=new temsilci(Metot2);
temsilci nesne2=new temsilci(Metot1);
Delegate nesne3=Delegate.Combine(nesne1,nesne2);
nesne3.DynamicInvoke();
```

Bu program ekran çıktısı şöyle olmalıdır:

```
Burası Metot1()
Burası Metot2()
Burası Metot1()
```

NOT: Delegate sınıfı türünden bir nesnenin daima gizli türü olur, ve bu tür de daima bir temsilcidir. Yani Delegate sınıfı sadece bir kap vazifesi görür. Ayrıca Delegate sınıfı türünden bir nesneyle, bir temsilci nesnesinden farklı olarak normal metot çağrısı, += operatörüyle metot eklemesi ve -= operatörüyle metot çıkartımı yapılamaz. Delegate nesneleriyle bu işlemler yalnızca Delegate sınıfına ait olan üye elemanlarla yapılır. Ayrıca istersek tür dönüştürme operatörüyle Delegate nesnesini gizli türüne (ki bu bir temsilci oluyor) dönüştürebiliriz. Örnek:

```
using System;
class Ana
{
   delegate void temsilci();
   static void Metot1()
   {
      Console.WriteLine("Metot1");
   }
   static void Metot2()
   {
```

```
      Console.WriteLine("Metot2");
   }
   static void Metot3()
   {
      Console.WriteLine("Metot3");
   }
   static void Main()
   {
      temsilci t=new temsilci(Metot1);
      t+=new temsilci(Metot2);
      t+=new temsilci(Metot3);
      Delegate d=t;
      temsilci c=(temsilci)d;
      c();
   }
}
```

Bu program ekrana alt alta `Metot1`, `Metot2` ve `Metot3` yazar.

NOT: Tür dönüştürme operatörüyle bir nesneyi daima gizli türüne dönüştürebiliriz. Bu işlem için tür içinde bir explicit metot tanımlamaya gerek yoktur.

İsimsiz metotlar

Şu ana kadar öğrendiklerimize göre temsilcilerin var olma nedeni metotlardı. Yani bir metot olmadan bir temsilci de olmuyordu. Ancak artık bir metot olmadan işe yarayan bir temsilci tasarlayacağız. Örnek:

```
using System;
class isimsiz
{
   delegate double temsilci(double a,double b);
   static void Main()
   {
      //temsilci nesnesine bir kod bloğu bağlanıyor.
      temsilci t=delegate(double a,double b)
      {
         return a+b;
      };
      //temsilci nesnesi ile kodlar çalıştırılıyor.
      double toplam=t(1d, 9d);
      Console.WriteLine(toplam);
   }
}
```

Gördüğünüz gibi bir temsilci nesnesine bir metot yerine direkt kodlar atandı.

Temsilcilerde kalıtım durumları

C# 2.0'dan önce temsilci nesnelerinin temsil edeceği metotların prototipi temsilcinin prototipi ile aynı olmalıydı. Yani gerek geri dönüş tipi, gerekse de parametrelerin tipi arasında bir türeme söz konusu olsa bile tür uyumsuzluğu var sayılıyordu. C# 2.0 ve daha sonraki sürümlerde programcılar bu gereksiz tür uyumu derdinden kurtulmuştur. Örnek program:

```
using System;
delegate Ana temsilciAna();
delegate Yavru temsilciYavru();
class Ana
{
}
class Yavru:Ana
{
}
class program
{
   static Ana MetotAna()
   {
      return new Ana();
   }
   static Yavru MetotYavru()
   {
      return new Yavru();
   }
   static void Main()
   {
      temsilciAna nesneAna=new temsilciAna(MetotYavru);
      //ancak aşağıdaki kod hatalı
      //temsilciYavru nesneYavru=new temsilciYavru(MetotAna);
   }
}
```

Gördüğünüz gibi geri dönüş tipi Ana olan temsilciAna() temsilcisi türünden nesne geri dönüş tipi Yavru olan MetotYavru metodunu temsil edebildi. Ancak geri dönüş tipi Yavru olan temsilciYavru() temsilcisi türünden nesne geri dönüş tipi Ana olan MetotAna metodunu temsil edemezdi.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Olaylar

Olaylar geçen derste öğrendiğimiz temsilcilerle yakından ilişkilidir. Olaylar daha çok Windows uygulamaları geliştirirken karşımıza çıkar. Olayların ne demek olduğunu sanırım en iyi Windows uygulamalarından bir örnek vererek anlatabilirim. Düşünün ki bir form var ve bu forma bir buton getirdik. Bu yaptığımız işlem aslında Button sınıfı türünden bir nesne yaratmaktır. Button sınıfının içinde çeşitli özellikler vardır. Örneğin Text özelliği butonun üzerindeki yazıyı belirtir. Benzer şekilde Button sınıfının içinde olaylar da vardır. Örneğin Click olayı butonun tıklanmasını temsil eder. Button sınıfının bunun gibi daha farklı olayları da vardır. Bu olaylara bazı komutları bağlayarak olay gerçekleştiğinde bazı komutların çalıştırılmasını sağlarız. C#'ta olaylar event anahtar sözcüğüyle tanımlanır. Olaylar sınıfların bir üye elemanıdır. Bir olay bildirimi şöyle yapılır:

```
erişim_belirleyici event temsilci olay_ismi;
```

Şimdi olaylarla ilgili bir örnek yapalım:

```
using System;
delegate void OlayYoneticisi(); //Olay yöneticisi bildirimi
class Buton   //Olayın içinde bulunacağı sınıf bildirimi
{
   public event OlayYoneticisi Click; //Olay bildirimi
   public void Tiklandi()  //Olayı meydana getirecek metot
   {
      if(Click!=null)
         Click();
   }
}
class AnaProgram
{
   static void Main()
   {
      Buton buton1=new Buton();
      buton1.Click+=new OlayYoneticisi(Click); //Olay sonrası
işletilecek metotların eklenmesi
      buton1.Tiklandi(); //Olayın meydana getirilmesi.
   }
   //Olay sonrası işletilecek metot
   static void Click()
   {
      Console.WriteLine("Butona tıklandı.");
   }
}
```

Bu program ekrana `Butona tıklandı.` yazacaktır. Şimdi programımızı derinlemesine inceleyelim. Programımızdaki en önemli satırlar:

```
delegate void OlayYoneticisi();
```

ve Buton sınıfındaki

```
public event OlayYoneticisi Click;
```

satırıdır. Birinci satır bir temsilci, ikinci satır da bir olay bildirimidir. Bu iki satırdan anlamamız gereken Click olayı gerçekleştiğinde parametresiz ve geri dönüş tipi void olan metotların çalıştırılabileceğidir. Ana metottaki

```
buton1.Tiklandi();
```

satırı ile olay gerçekleştiriliyor. Tiklandi metoduna bakacak olursak;

```
public void Tiklandi()
{
   if(Click!=null)
      Click();
}
```

Burada bir kontrol gerçekleştiriliyor. Burada yapılmak istenen tam olarak şu: Eğer Click olayı gerçekleştiğinde çalışacak bir metot yoksa hiçbir şey yapma. Varsa olayı gerçekleştir.

```
buton1.Click+=new OlayYoneticisi(Click);
```

Main() metodundaki bu satırla da buton1'in Click olayına bir metot ekliyoruz. Yani buton1 nesnesinin Click olayı gerçekleştiğinde Click metodu çalışacak.

NOT: Örneğimizde olayı kendimiz gerçekleştirdik. Ancak Windows programlarında programcı genellikle olayın gerçekleşmesiyle değil olaydan sonra çalıştırılacak metotlarla ilgilenir.

NOT: Örneğimizde olayı `buton1.Click+=new OlayYoneticisi(Click);` satırıyla yalnızca bir metoda bağladık. İstersek tıpkı temsilcilerde yaptığımız gibi olaya birden fazla metot bağlayabilirdik.

NOT: Örneğimizde OlayYoneticisi temsilcisi geri dönüş tipi void olan ve herhangi bir parametre almayan metotları temsil etmektedir. Dolayısıyla olayın gerçekleşmesinin ardından bu özelliklere sahip metotlar çalıştırılabilecektir. Ancak bu şart değildir. OlayYoneticisi temsilcisi değişik özelliklere sahip metotları temsil edebilir. Bu durumda olaya o özelliklere sahip metotlar bağlanabilecektir.

NOT: Olaydan sonra çalıştırılacak metot ya da metotların static olma zorunluluğu yoktur. Örnek:

```
using System;
delegate void OlayYoneticisi(); //Olay yöneticisi bildirimi
class Buton   //Olayın içinde bulunacağı sınıf bildirimi
{
   public event OlayYoneticisi Click; //Olay bildirimi
   public void Tiklandi()  //Olayı meydana getirecek metot
   {
      if(Click!=null)
         Click();
   }
}
class AnaProgram
{
   static void Main()
   {
      Buton buton1=new Buton();
      AnaProgram nesne=new AnaProgram();
      buton1.Click+=new OlayYoneticisi(nesne.Click); //Olay sonrası
işletilecek metotların eklenmesi
      buton1.Tiklandi(); //Olayın meydana getirilmesi.
```

```
    }
    //Olay sonrası işletilecek metot
    void Click()
    {
        Console.WriteLine("Butona tıklandı.");
    }
}
```

Şimdi programı biraz daha geliştirelim:

```
using System;
delegate void OlayYoneticisi(); //Olay yöneticisi bildirimi
class Buton   //Olayın içinde bulunacağı sınıf bildirimi
{
    public event OlayYoneticisi Click; //Olay bildirimi
    public void Tiklandi()  //Olayı meydana getirecek metot
    {
        if(Click!=null)
            Click();
    }
}
class Pencere
{
    int a;
    public Pencere(int a)
    {
        this.a=a;
    }
    public void Click()
    {
        Console.WriteLine("Pencere sınıfındaki metot çağrıldı - "+a);
    }
}
class AnaProgram
{
    static void Main()
    {
        Buton buton1=new Buton();
        Pencere nesne1=new Pencere(1), nesne2=new Pencere(2);
        buton1.Click+=new OlayYoneticisi(Click); //Olay sonrası
işletilecek metotların eklenmesi
        buton1.Click+=new OlayYoneticisi(nesne1.Click);
        buton1.Click+=new OlayYoneticisi(nesne2.Click);
        buton1.Tiklandi(); //Olayın meydana getirilmesi.
    }
    //Olay sonrası işletilecek metot
    static void Click()
    {
```

```
        Console.WriteLine("Butona tıklandı.");
    }
}
```

Bu programın ekran çıktısı şöyle olacaktır:

```
Butona tıklandı.
Pencere sınıfındaki metot çağrıldı - 1
Pencere sınıfındaki metot çağrıldı - 2
```

Yani isimleri aynı olsa bile bellekteki farklı noktalarda bulunan her farklı metot olaya ayrı ayrı bağlanabilir. Başka bir örnek:

```
using System;
delegate void OlayYoneticisi(); //Olay yöneticisi bildirimi
class AnaProgram
{
    static void Main()
    {
        AnaProgram nesne=new AnaProgram();
        nesne.Olay+=new OlayYoneticisi(Metot); //Olay sonrası
işletilecek metotların eklenmesi
        nesne.Olay(); //Olayın gerçekleştirilmesi
    }
    //Olay sonrası işletilecek metot
    static void Metot()
    {
        Console.WriteLine("Butona tıklandı.");
    }
    event OlayYoneticisi Olay; //Olay bildirimi
}
```

Burada bütün üye elemanlar aynı sınıfta bildirildi. Aslında olay mantığını anlayabilmeniz için en basit örnek bu. Bu dersteki ilk örneğimizde

```
public void Tiklandi()
{
    if(Click!=null)
        Click();
}
```

gibi bir kontrol gerçekleştirmiştik. Çünkü C# derleyicisi programı satır satır derliyordu. Yukarıdaki satırlara gelindiğinde henüz olaya herhangi bir metot bağlanmamış olacak ve hata verecekti. Çünkü henüz `Click();` satırında hangi metodun çağrılacağını bilmiyordu. Bu örneğimizde ise Olay olayını herhangi bir metot üzerinden değil, direkt çağırdık. Gördüğünüz gibi olaylar temsilcilere oldukça benziyor.

NOT: Olaylar tıpkı metotlar, özellikler ve indeksleyiciler gibi bir arayüz elemanı olarak bildirilebilir. Bu durumda o arayüzü kullanan sınıfın o olayı içermesini zorlarız. Örnek:

```
using System;
public delegate int OlayYoneticisi();
interface IArayuz
```

```
{
    int Metot1();
    int Metot2();
    int sahteozellik
    {
        set;
        get;
    }
    int this[int indeks]
    {
        get;
    }
    event OlayYoneticisi Olay;
}
class deneme:IArayuz
{
    public int Metot1()
    {
        return 1;
    }
    public int Metot2()
    {
        return 2;
    }
    public int sahteozellik
    {
        set{}
        get{return 3;}
    }
    public int this[int indeks]
    {
        get{return indeks;}
    }
    public event OlayYoneticisi Olay;
    static void Main()
    {
        deneme nesne=new deneme();
        nesne.Olay+=new OlayYoneticisi(nesne.Metot1);
        Console.WriteLine(nesne.Olay());
    }
}
```

Biraz uzun bir örnek oldu ama aslında son derece basit bir örnek. Gördüğünüz gibi temsilciler arayüzlerin içine giremediği için dışarıda belirtilmiş. Böylelikle arayüzler konusunu bir tekrar etmiş olduk. Ayrıca olaylar özet ya da sanal olarak bildirilebilir. Bunları örneklemeye gerek görmüyorum.

NOT: Tıpkı temsilcilerde olduğu gibi bir olaydan metot çıkarmak için -= operatörü kullanılır.

add ve remove erişimcileri

Bazen bir olaya += operatörü ile metot eklendiğinde veya -= operatörü ile metot çıkarıldığında belirli kod bloklarının çalıştırılmasını isteyebiliriz. İşte bunun için ilgili olayın blokları içine add ve remove blokları koyulur. add ve remove bloklarını sahte özelliklerdeki set ve get bloklarına benzetebiliriz.

```
Örnek program:
 using System;
 class deneme
 {
    delegate void OlayYoneticisi();
    event OlayYoneticisi Olay
    {
       add
       {
          Console.WriteLine("Olaya metot eklendi.");
       }
       remove
       {
          Console.WriteLine("Olaydan metot çıkarıldı.");
       }
    }
    static void Main()
    {
       deneme d=new deneme();
       d.Olay+=new OlayYoneticisi(d.Metot1);
       d.Olay+=new OlayYoneticisi(d.Metot2);
       d.Olay-=new OlayYoneticisi(d.Metot1);
    }
    void Metot1(){}
    void Metot2(){}
 }
```

Bu programın ekran çıktısı şöyle olur:

```
Olaya metot eklendi.
Olaya metot eklendi.
Olaydan metot çıkarıldı.
```

Şimdi add ve remove erişimlerini bir olaya en fazla iki metodun bağlanabilmesi amacıyla kullanalım.

```
 using System;
 delegate void OlayYoneticisi(); //Olay yöneticimizi tanımladık.
 class Buton
 {
    OlayYoneticisi[] olay=new OlayYoneticisi[2]; //Olay yöneticimiz
türünden iki elemanlı bir dizi oluşturduk.
    public event OlayYoneticisi ButonKlik
    {
       add //Buradaki kodlar olaya metot eklendiğinde çalıştırılacak.
```

```
        {
            int i;
            for(i=0;i<2;++i)
                if(olay[i]==null) //Eğer olay[i]'nin temsil ettiği bir
metot yoksa
                {
                    olay[i]=value; //Buradaki value olaya eklenen metottur.
 (OlayYoneticisi temsilcisi türünden)
                    break;
                }
                if(i==2)
                    Console.WriteLine("Olaya en fazla iki metot
eklenebilir.");
        }
        remove //Buradaki kodlar olaydan metot çıkarıldığında
çalıştırılacak.
        {
            int i;
            for(i=0;i<2;++i)
                if(olay[i]==value) //Buradaki value olaydan çıkarılan
metottur. (OlayYoneticisi temsilcisi türünden)
                {
                    olay[i]=null;
                    break;
                }
          if(i==2)
                Console.WriteLine("Metot bulunamadı");
        }
    }
    public void Kliklendi()
    {
        for(int i=0;i<2;++i)
            if(olay[i]!=null)
                olay[i]();
    }
 }
 class Pencere
 {
    int PencereNo;
    public Pencere(int no)
    {
        PencereNo=no;
    }
    public void ButonKlik()
    {
        Console.WriteLine("{0} nolu pencere olayı algıladı.",PencereNo);
    }
```

```
}
public class OlayTest
{
   static void Main()
   {
      Buton buton=new Buton();
      Pencere p1=new Pencere(1);
      Pencere p2=new Pencere(2);

      //Geçerli ekleme:
      buton.ButonKlik+=new OlayYoneticisi(ButonKlik);
      buton.Kliklendi();
      Console.WriteLine();

      //Geçerli ekleme:
      buton.ButonKlik+=new OlayYoneticisi(p1.ButonKlik);
      buton.Kliklendi();
      Console.WriteLine();

      //Geçersiz ekleme (Olay dolu):
      buton.ButonKlik+=new OlayYoneticisi(p2.ButonKlik);
      buton.Kliklendi();
      Console.WriteLine();

      buton.ButonKlik-=new OlayYoneticisi(p1.ButonKlik);
      buton.Kliklendi();
      Console.WriteLine();

      buton.ButonKlik-=new OlayYoneticisi(ButonKlik);
      buton.Kliklendi();
      Console.WriteLine();

      //Geçersiz çıkarma (metot yok):
      buton.ButonKlik-=new OlayYoneticisi(ButonKlik);
      buton.Kliklendi();
   }
   public static void ButonKlik()
   {
      Console.WriteLine("Buton kliklendi");
   }
}
```

Bu programın ekran çıktısı şöyle olmalıdır:

```
Buton kliklendi

Buton kliklendi
1 nolu pencere olayı algıladı.
```

```
Olaya en fazla iki metot eklenebilir.
Buton kliklendi
1 nolu pencere olayı algıladı.

Buton kliklendi

Metot bulunamadı
```

NOTLAR:

1. add ve remove bloklarını içeren olaylar özet (abstract) olarak bildirilemez.
2. Bir olayda add ve remove blokları birlikte olmalıdır. Ayrı ayrı olamazlar.

Windows uygulamalarında olaylar

İleride göreceğimiz Windows uygulamaları geliştirirken olaylarla iç içe yaşayacağız. Daha önce demiştim ancak yine söylemek istiyorum. Forma bir buton getirdiğimizde aslında Button sınıfı türünden bir nesne yaratırız. Button sınıfının Click olayı vardır. İşte biz form tasarlama penceresinde (ileride göreceğiz) herhangi bir butona çift tıklarsak o butonun Click olayına bağlanan metodun içeriğini yazabileceğimiz bir kod penceresi açılır. Bu metodun genellikle prototipi şu şekildedir:

```
void EvenHandler(object kaynak, EventArgs e)
```

kaynak olayı hangi nesnenin çalıştırdığıdır. Yani forma bir buton getirdiğimizde, yani bir Button nesnesi oluşturduğumuzda, bu nesnenin bir adı olmalıdır. Forma buton getirmeyi kodlarla yapmadığımız için Visual Studio bu Button nesnesine button1, button2 gibi adlar verir. İşte kaynak demek bu buton nesnesinin adıdır. Button sınıfı türündendir. Kuşkusuz forma sadece buton getirmeyeceğimiz, başka form elemanları da getirebileceğimiz için bu parametre object türündedir. EventArgs ise System isim alanında bulunan bir sınıftır. Bu sınıftaki çeşitli özellikler olayla ilgili ayrıntılı bilgi içerirler. Yani Button sınıfının Click olayına otomatik olarak EventArgs sınıfının çeşitli özelliklerini olaya göre değiştiren gizli metotlar da bağlanır. Örneğin EventArgs sınıfındaki özelliklerle kullanıcının butonun hangi koordinatlarına tıkladığını öğrenebiliriz.

.Net olayları oluşturulurken System isim alanındaki EventHandler temsilcisi kullanılır. Doğal olarak bu temsilci yukarıdaki metot prototipindedir.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Önişlemci komutları

Şimdiye kadar gördüğümüz kodlar direkt olarak exe dosyasına katılıyordu. Şimdi göreceğimiz önişlemci komutlarının görevi ise sadece derleyiciye derlemenin nasıl yapılacağı hakkında bilgi vermektir. Önişlemci komutları derleme sonucu oluşan exe dosyasına aktarılmazlar. Önişlemci komutları # karakteri ile belirtilirler. C# kodlarından farklı olarak önişlemci komutları enter'la sonlandırılmalıdır. Sonlarında ; karakteri bulunmaz. Şimdi isterseniz bütün önişlemci komutlarını teker teker görelim.

#define

#define komutu bir sembol tanımlamak için kullanılır. Sembollerin tek başlarına bir anlamı yoktur. İleride göreceğimiz diğer önişlemci komutlarının çalışabilmeleri için gereklidirler. #define ile sembol tanımlama program kodlarının en başında, (using deyimlerinden de önce) olmalıdır. Örnek program:

```
#define ENGLISH
#define TURKISH
using System;
class Onislemci
{
   static void Main()
   {
   }
}
```

Bu programda ENGLISH ve TURKISH isimli iki sembol tanımlanmıştır. Diğer bütün önişlemci komutlarında olduğu gibi önişlemci komut parçaları arasında (örneğin `#define` ile `TURKISH` arasında) en az bir boşluk olmalıdır. Aynı sembolün tekrar tekrar tanımlanması hata verdirmez.

#undef

#define ile tanımlanmış bir sembolü kaldırmak için kullanılır. Olmayan bir sembolü kaldırmaya çalışmak hata verdirmez. Programın herhangi bir satırında olabilir. Örnek program:

```
#define ENGLISH
using System;
#undef ENGLISH
class Onislemci
{
   static void Main()
   {
   }
}
```

Sembollerin büyük ya da küçük harf olması zorunlu değildir. Ancak tamamen büyük harf okunurluğu artırdığı için tavsiye edilir. Sembollerin büyük-küçük harf duyarlılığı vardır.

#if ve #endif

#if ve #endif komutları bir sembolün tanımlanmış olup olmamasına göre birtakım kodların derlenip derlenmemesinin sağlanması için kullanılır. Yani son tahlilde kaynak kodun bir kısmı exe dosyasına ya eklenir ya eklenmez. Önişlemci komutlarında bloklar olmadığı için #endif komutu da #if komutunun açtığı kod bölgesini kapatmak için kullanılır. Yani #if ve #endif arasındaki kodlar koşula göre derlenip derlenmez. Örnek program:

```
#define ENGLISH
using System;
class Onislemci
{
   static void Main()
   {
      Console.WriteLine("Programa hoş geldiniz.");
      #if ENGLISH
      Console.WriteLine("Bu program ENGLISH");
      #endif
   }
}
```

Bu program ekrana şu çıktıyı verecektir.

```
Programa hoş geldiniz.
Bu program ENGLISH
```

Eğer programın başına `#define ENGLISH` satırını eklemeseydik program çıktısındaki ikinci satır olmayacaktı. #if önişlemci komutuyla mantıksal operatörleri de kullanabiliriz. Örnek program:

```
#define ENGLISH
#define TURKISH
using System;
class Onislemci
{
   static void Main()
   {
      Console.WriteLine("Programa hoş geldiniz.");
      #if ENGLISH && (!TURKISH)
      Console.WriteLine("Bu program ENGLISH");
      #endif
   }
}
```

Bu program sonucunda ekrana yalnızca `Programa hoş geldiniz.` satırı yazılacaktır. Şimdi bu mantıksal operatör durumlarını topluca görelim:
#if ENGLISH && TURKISH ENGLISH ve TURKISH varsa
#if ENGLISH || TURKISH ENGLISH veya TURKISH varsa
#if ENGLISH && (!TURKISH) ENGLISH var ama TURKISH yoksa
#if ENGLISH && (TURKISH==false) ENGLISH var ama TURKISH yoksa
Benzer şekilde çoklu koşullandırmalar da mümkündür. Örnek:
#if ENGLISH && TURKISH && FRANSIZCA

#else

#else önişlemci komutu C#'taki else ile aynı göreve sahiptir. Koşul sağlanmışsa bir kod bloğunun derlenmesini sağlayan #if komutu ile birlikte kullanılır. Örnek:

```
#define ENGLISH
using System;
class Onislemci
{
   static void Main()
   {
      Console.WriteLine("Programa hoş geldiniz.");
      #if ENGLISH
      Console.WriteLine("Bu program ENGLISH");
      #else
      Console.WriteLine("Bu program ENGLISH değil");
      #endif
   }
}
```

Gördüğünüz gibi #else komutu #if ile #endif arasında kullanılıyor. Başka bir örnek:

```
#define ENGLISH
using System;
#if ENGLISH
class Onislemci
{
   static void Main()
   {
      Console.WriteLine("Bu program ENGLISH");
   }
}
#else
class Onislemci
{
   static void Main()
   {
      Console.WriteLine("Bu program ENGLISH değil");
   }
}
#endif
```

Bu programda aynı isimli birden fazla sınıf aynı kaynak kodda olmasına rağmen program hata vermedi. Çünkü her halukarda bu sınıflardan yalnızca biri derlenecektir.

#elif

#elif C#'taki else if'in kısaltılmışıdır. Yani değilse bir de bu koşula bak anlamı verir. Örnek program:

```
#define ENGLISH
using System;
class Onislemci
{
   static void Main()
   {
      Console.WriteLine("Programa hoş geldiniz.");
      #if ENGLISH
      Console.WriteLine("Bu program ENGLISH");
      #elif TURKISH
      Console.WriteLine("Bu program TURKISH");
      #else
      Console.WriteLine("Bu program hem TURKISH, hem de ENGLISH
değil");
      #endif
   }
}
```

#elif komutlarının sayısını bir #if-#endif aralığında istediğimiz kadar artırabiliriz.
NOT: İç içe geçmiş #if-#elif-#else komutları da olabilir. Örnek program:

```
#define ENGLISH
using System;
class Onislemci
{
   static void Main()
   {
      #if ENGLISH
       #if TURKISH
      Console.WriteLine("Bu program hem ENGLISH hem TURKISH");
       #endif
      #elif TURKISH
      Console.WriteLine("Bu program sadece TURKISH");
      #endif
   }
}
```

#error

#error komutu derleyiciyi hata vermeye zorlamak için kullanılır. Örnek program:

```
#define ENGLISH
#define TURKISH
using System;
class Onislemci
{
```

```
    static void Main()
    {
        #if ENGLISH && TURKISH
            #error Hem TURKISH hem ENGLISH aynı anda olamaz.
        #endif
    }
}
```

Bu program kodumuz derlenmeyecek, hata mesajı olarak da ekrana `Hem TURKISH hem ENGLISH aynı anda olamaz.` yazacaktır.

#warning

#warning komutu #error komutu ile aynı mantıkta çalışır. Tek fark ekrana hata değil, uyarı verilmesidir. Yani programımız derlenir, ama uyarı verilir. Örnek program:

```
#define ENGLISH
#define TURKISH
using System;
class Onislemci
{
    static void Main()
    {
        #if ENGLISH && TURKISH
            #warning Hem TURKISH hem ENGLISH tavsiye edilmez.
        #endif
    }
}
```

#line

Eğer programımızda bir hata var ve derleyici de (csc.exe) bu hatayı tespit etmişse bize bu hatanın hangi dosyanın hangi satırında olduğunu verir. İstersek #line komutuyla bu dosya adını ve satır numarasını değiştirebiliriz. Yani tabiri caizse derleyiciyi kandırabiliriz. #line komutu kendinden hemen sonra gelen satırın özelliklerini değiştirmek için kullanılır. Örnek program:

```
using System;
class Onislemci
{
    static void Main()
    {
        #line 1 "YeniDosya.cs"
        int a="deneme";
    }
}
```

Bu programda int türünden olan a değişkenine yanlış türde sabit atadık. Bu hatayı bilinçli olarak yaptık. Derleyici hatanın 1. satırda ve YeniDosya.cs dosyasında olduğunu iddia edecektir. Çünkü artık derleyici `#line 1 "YeniDosya.cs"` komutundan hemen sonra gelen satırı 1. satır ve `YeniDosya.cs` dosyası olarak tanıyor.

#region ve #endregion

#region ve #endregion komutlarının, kodları Not Defteri'nde yazan bizler için pek bir önemi yok. Çünkü #region ve #endregion komutları kodları bölgelere ayırmaya yarıyor. Kod bölgesinin yanındaki + işaretine tıklayınca kodlar gösteriliyor, gösterilen kodların yanındaki - işaretine tıklayınca da kodlarımız gizleniyor. Bu komut Visual Studio veya Not Defteri'nden daha gelişmiş bir editör programı kullananlar için faydalı olabilir. Örnek:

```
using System;
class Onislemci
{
   static void Main()
   {
      #region YeniAlan
      //Bu kısmı yeni bir bölge yaptık
      #endregion
   }
}
```

Neden önişlemci komutları?

Önişlemci komutları genellikle belirli koşullara göre programımızın yalnızca bir kısmını derlemek için kullanılır. Örneğin bir programın hem İngilizce hem Türkçe versiyonunu aynı kaynak kodda bulunduruyorsak önişlemci komutlarını programın yalnızca Türkçe ya da yalnızca İngilizce versiyonunu derlemek amacıyla kullanabiliriz.

Bu kitabın diğer sayfaları

- C# hakkında temel bilgiler
- İlk programımız
- Değişkenler
- Tür dönüşümü
- Yorum ekleme
- Operatörler
- Akış kontrol mekanizmaları
- Rastgele sayı üretme
- Diziler
- Metotlar
- Sınıflar
- Operatör aşırı yükleme
- İndeksleyiciler
- Yapılar
- Enum sabitleri
- İsim alanları
- System isim alanı
- Temel I/O işlemleri
- Temel string işlemleri
- Kalıtım
- Arayüzler
- Partial (kısmi) tipler
- İstisnai durum yakalama mekanizması
- Temsilciler
- Olaylar
- Önişlemci komutları
- Göstericiler
- Assembly kavramı
- Yansıma
- Nitelikler
- Örnekler
- Şablon tipler
- Koleksiyonlar
- yield
- Veri tabanı işlemleri
- XML işlemleri
- Form tabanlı uygulamalar
- Visual Studio.NET
- Çok kanallı uygulamalar
- Linux'ta C# kullanımı
- Kaynakça
- Giriş sayfası
- **Ayrıca Bakınız:**

  ASP.NET

Göstericiler

Göstericiler (pointer) yapı nesnelerinin bellekteki adreslerini tutan bir veri tipidir. Temel veri türlerinden byte, sbyte, short, ushort, int, uint, long, ulong, float, double, decimal, char ve bool birer yapıdır. (string ve object ise birer sınıftır.) Bu yapıların yanında .Net Framework kütüphanesindeki diğer yapılar ve kendi oluşturduğumuz yapı nesneleriyle de göstericileri kullanabiliriz. Ancak C# normal yollarla gösterici kullanılmasına izin vermez. Programımızın herhangi bir yerinde gösterici kullanabilmek için o yeri `unsafe` anahtar sözcüğüyle belirtmemiz gerekir. unsafe anahtar sözcüğünü şu şekillerde kullanabiliriz:

- Bir sınıfı unsafe olarak belirtirsek o sınıf içinde gösterici kullanabiliriz.

```
unsafe class Sinif
{
    ...
}
```

- Herhangi bir metodun içinde bir unsafe bloğu oluşturarak o bloğun içinde gösterici kullanabiliriz.

```
int Metot()
{
   unsafe
   {
       ...
   }
}
```

- Normal bir metodu unsafe olarak belirterek o metot içinde gösterici kullanabiliriz.

```
unsafe int Metot()
{
    ...
}
```

- Özellikleri unsafe olarak belirterek yeni bir gösterici oluşturabiliriz. Bu durumda sınıfı unsafe olarak belirtmeye gerek kalmaz. Ancak bir metot içinde bir gösterici oluşturabilmek için ya o metodun unsafe olarak bildirilmesi ya da bir unsafe bloğunun açılması gerekir. Ayrıca içinde gösterici kullandığımız bir kaynak kod dosyası normal yollarla derlenemez. İçinde gösterici kullandığımız bir kaynak kod dosyamızı aşağıdaki iki yoldan biriyle derleriz.

```
csc /unsafe KaynakKod.cs
```

veya

```
csc -unsafe KaynakKod.cs
```

Gösterici bildirimi

Gösterici bildirimi ilgili yapı adının sonuna * işareti koyularak yapılır. Örnekler:

```
char* gosterici1;
int* gosterici2;
```

Bu göstericilerden birincisi bir char türünden nesnenin (değişkenin), ikincisi de bir int türünden nesnenin adresini tutabilir. Birden fazla aynı türde gösterici oluşturmak için normal veri türlerinde olduğu gibi virgül kullanılabilir. Örnek:

```
int* gosterici1, gosterici2;
```

Göstericileri sabit ifadeler ile de bildirebiliriz. Örnek:

```
double* gosterici=(double*)123456;
```

Burada bellekteki 123456 adresi gosterici göstericisine atanıyor. Ancak biz 123456 adresinde ne olduğunu bilmediğimiz için bu tür bir kullanım son derece tehlikelidir.

& operatörü

& operatörü yapı nesnelerinin bellekteki adreslerini üretir. Operandı hangi türdeyse o türün gösterici tipinde bir değer döndürür. Örnek:

```
int a=5;
int* gosterici;
gosterici=&a;
```

* operatörü

* operatörü adreslerdeki veriyi görmek veya değiştirmek için kullanılır. * operatörü hangi tür göstericiyle kullanıldıysa o türde nesne geri döndürülür. Örnek:

```
double a=5;
double* gosterici;
gosterici=&a;
*gosterici=10;
```

Yani kısaca & ve * operatörleri gösterici ve nesne arasında dönüşüm yapmak için kullanılır.

NOTLAR:

1-) Henüz adresi olmayan bir göstericiye değer atanamaz. Örneğin şu kod hatalıdır:

```
int* gosterici;
*gosterici=10;
```

2-) Ancak henüz bir değeri olmayan bir nesnenin adresi alınabilir. Yani şu kod hata vermez:

```
int a;
int* ptr=&a;
```

Göstericiler arasında tür dönüşümü

Göstericiler arasında bilinçli tür dönüşümü yapılabilir. Ancak dikkatli olunmalıdır. Çünkü büyük türün küçük türe dönüşümünde veri kaybı yaşanabilir. Küçük türün büyük türe dönüşümünde de küçük tür bellekte kendine ait olmayan alanlardan da alır. Dolayısıyla her iki durumda da istemediğimiz durumlar oluşabilir. Göstericiler arasında bilinçsiz tür dönüşümü ise imkansızdır. Örnek bir program:

```
using System;
class Gostericiler
{
   unsafe static void Main()
   {
        char c='A';
        int i=80;
        char* cg=&c;
        int* ig=&i;
        cg=(char*)ig;
        Console.WriteLine(*cg);
  }
}
```

Bu program sonucunda ekrana P yazılacaktır. (80'in Unicode karşılığı P'dir.) Bu programda herhangi bir veri kaybı gerçekleşmedi. Çünkü 80 char türünün kapasitesindeydi.

Dersin en başında söylemiştik: Göstericiler yapı nesnelerinin bellekteki adreslerini tutarlar. İşte bu adresi elde etmek için ilgili gösterici bir tam sayı türüne dönüştürülmelidir. Örnek:

```
using System;
class Gostericiler
{
   unsafe static void Main()
   {
        int i=80;
        int* ig=&i;
        uint adres=(uint)ig;
        Console.WriteLine("{0:X}",adres);
  }
}
```

Bu program sonucunda benim bilgisayarım 13F478 çıktısını verdi. Ancak bu sizde değişebilir. Çünkü işlemcinin verileri nereye koyacağını hiç birimiz bilemeyiz. Ayrıca adresi ekrana yazdırırken formatlama yapıp 16'lık düzende yazdırdığımın da farkında olmalısınız. Çünkü bellek adresleri genellikle 16'lık düzendeki sayılarla ifade edilir.

Göstericilerin adreslerini elde ederken uint türüne dönüşüm yapmak zorunlu değildir. İstediğiniz tam sayı türüne dönüşüm yapabilirsiniz. Ancak uint, bir bellek adresini tam olarak tutabilecek en küçük veri tipi olduğu için ilgili göstericiyi uint türüne dönüştürmeniz tavsiye edilir.

void göstericiler

void tipinden göstericilere her türden adres atanabilir. void türü nesnelerdeki object türüne benzer. Örnek kod

```
int* gosterici1;
void* gosterici2;
gosterici2=gosterici1;
```

sizeof operatörü

sizeof operatörü yapıların bellekte ne kadar yer kapladıklarını bayt türünden verir. Geri dönüş tipi inttir. Örnek kod:

```
Console.WriteLine(sizeof(int));
```

Bu kod ekrana 4 çıktısını verir. İsterseniz bir de şöyle bir örnek yapalım:

```
using System;
class Gostericiler
{
   struct yapi
   {
      public int Ozellik1;
      public int Ozellik2;
   }

   unsafe static void Main()
   {
      Console.WriteLine(sizeof(yapi));
   }
}
```

Bu program 8 çıktısını verir.

Gösterici aritmetiği

Göstericilerin bir adres kısmı, bir de tür kısmı olmak üzere iki kısmı bulunur. Tür kısmı 4 bayt yer kaplar. Adres kısmının kapladığı alan ise türe göre değişir. Örneğin tür int* ise adres kısmı 4 bayt yer kaplar ve gösterici toplam olarak 8 bayt yer kaplar. Herhangi bir göstericinin adres kısmıyla ilgili matematiksel işlem yapmaya gösterici aritmetiği denir. Göstericilerle matemetiksel işlem yapmak göstericinin türüne bağlıdır. int* türünden bir göstericiyi 1 ile toplarsak o göstericinin adres kısmı 4 ile toplanır. Yani herhangi bir göstericiyi herhangi bir tam sayı ile toplarsak ya da herhangi bir tam sayıyı herhangi bir göstericiden çıkarırsak ilgili tam sayı değil, ilgili tam sayının göstericinin türünün bayt cinsinden boyutu ile çarpımı işleme sokulur. Örnek:

```
using System;
class Gosterici
{
   unsafe static void Main()
   {
      int* g1=(int*)500;
      char* g2=(char*)500;
      double* g3=(double*)500;
      byte* g4=(byte*)500;
```

```
        g1+=2;
        g2+=5;
        g3+=1;
        g4+=6;
        Console.WriteLine("{0} {1} {2}
{3}",(uint)g1,(uint)g2,(uint)g3,(uint)g4);
    }
 }
```

Bu programın ekran çıktısı `508 510 508 506` olmalıdır. Göstericiler yalnızca tam sayılarla matematiksel işleme girerler. +, -, --, ++, -= ve += operatörleri göstericilerle kullanılabilir. void göstericiler matematiksel işleme girmezler.

İki gösterici birbirinden çıkarılabilir. Ancak bu durumda bir gösterici değil, long türünden bir nesne oluşur. Örnek:

```
using System;
class Gosterici
{
   unsafe static void Main()
   {
      int* g1=(byte*)500;
      int* g2=(byte*)508;
      long fark=g2-g1;
      Console.WriteLine(fark);
   }
}
```

Bu program sonucunda ekrana 2 yazılır. Çünkü göstericilerde çıkarma yapılırken şu formül kullanılır:

```
(Birinci göstericinin adresi - İkinci göstericinin adresi)/Ortak türün byte cinsinden boyutu
```

Bu formülden de anlıyoruz ki yalnızca aynı türdeki göstericiler arasında çıkarma yapılabilir. Eğer bu formül sonucunda bir tam sayı oluşmuyorsa ondalıklı kısım atılır. Göstericilerle kullanılabilecek diğer operatörler ==, <, > gibi karşılaştırma operatörleridir. Bu operatörler iki göstericinin adres bilgilerini karşılaştırıp true veya false'tan uygun olanını üretirler.

fixed anahtar sözcüğü

Garbage Collection mekanizması sınıf nesnelerinin adreslerini program boyunca her an değiştirebilir. Dolayısıyla bu sınıf nesnesi üzerinden erişilen ilgili sınıfa ait yapı türünden olan özelliklerin de adresleri değişecektir. Ancak istersek bir sınıf nesnesi üzerinden erişilen bir yapı nesnesinin bir blok boyunca değişmemesini isteyebiliriz. Bunu fixed anahtar sözcüğü ile yaparız. Şimdi klasik örneği yapalım (fixed'sız):

```
using System;
class ManagedType
{
   public int x;
   public ManagedType(int x)
   {
      this.x=x;
   }
}
```

```
class Gosterici
{
   unsafe static void Main()
   {
      ManagedType mt=new ManagedType(5);
      int* g=&(mt.x)
   }
}
```

Bu programın derlenmesine C# izin vermeyecektir. Çünkü x bir sınıf nesnesidir ve sınıf nesnelerinin adresleri program boyunca değişebilir. Biz burada bu sınıf nesnesine ait olan bir özelliğin (yapı türünden) adresini aldık. Ancak muhtemelen program esnasında nesnemizin adresi değişecek ve bu adrese başka veriler gelecek. İşte karışıklığı engellemek için C# bu programın derlenmesini engelledi. Şimdi örneğimizi şöyle değiştirelim:

```
using System;
class ManagedType
{
   public int x;
   public ManagedType(int x)
   {
      this.x=x;
   }
}
class Gosterici
{
   unsafe static void Main()
   {
      ManagedType mt=new ManagedType(5);
      fixed(int* g=&(mt.x))
      {
         //Bu blok boyunca x özelliğinin adresi değiştirilmeyecektir.
      }
   }
}
```

Birden fazla özelliği fixed yapmak için:

```
ManagedType mt1=new ManagedType(5);
ManagedType mt2=new ManagedType(10);
fixed(int* g1=&(mt1.x))
fixed(int* g2=&(mt2.x))
{
   //Bu blok boyunca x özelliğinin adresi değiştirilmeyecektir.
}
```

Aynı şeyi şöyle de yapabilirdik:

```
ManagedType mt1=new ManagedType(5);
ManagedType mt2=new ManagedType(10);
fixed(int* g1=&(mt1.x), g2=&(mt2.x))
```

```
{
   //Bu blok boyunca x özelliğinin adresi değiştirilmeyecektir.
}
```

Göstericiler ile dizi işlemleri

Diziler bir System.Array sınıfı türünden nesnedir. Dolayısıyla diziler bir managed type'dır. Yani dizi elemanlarının adreslerini fixed anahtar sözcüğünü kullanmadan elde edemeyiz. Çok boyutlu diziler de dâhil olmak üzere tüm dizilerin elemanları bellekte ardışıl sıralanır. Bu hem performansta az da olsa bir artış hem de az sonra dizi elemanlarının bellekteki adreslerini alırken kolaylık sağlar. Örnek program:

```
using System;
class Gosterici
{
   unsafe static void Main()
   {
      int[] a={1,2,3,4};
      fixed(int* g=&a[0])
      {
         for(int i=0;i<a.Length;i++)
            Console.WriteLine(*(g+i));
      }
   }
}
```

Bu program dizinin elemanlarını alt alta yazacaktır. Programda öncelikle dizinin ilk elemanının bellekteki adresini alıp g göstericisine atadık. Sonra a dizisinin eleman sayısı kadar dönen bir döngü başlattık. Döngünün her iterasyonunda ilgili göstericiyi birer birer artırdık. Yani adres her seferinde birer birer ötelendi. Gösterici aritmetiğinde de gördüğümüz gibi aslında adresler 4'er bayt ötelendi. Eğer gösterici aritmetiğinde öteleme türe bağımlı olmayıp teker teker olsaydı döngüde her dizinin türüne göre farklı atlayış miktarı belirlememiz gerekecekti ve bu da oldukça zahmetli olacaktı.

NOT: Göstericiler ile indeksleyici kullanılabilir. Göstericilerde şu eşitlikler vardır:

- *(g+0) == g[0]
- *(g+1) == g[1]
- *(g+2) == g[2]
- *(g+3) == g[3]
- vb.

Yani indeksleyici almış bir gösterici demek ilgili göstericinin başlangıç adresinin indeksleyicideki sayı kadar ötelenmiş hâlinin nesne karşılığı demektir. Yani g[1] bir nesne belirtirken g+1 bir gösterici belirtir.

Bir diziyle aynı dizinin ilk elemanının adresi birbiriyle aynıdır. Bunu ispatlamak için şu programı yazabiliriz:

```
using System;
class Gosterici
{
   unsafe static void Main()
   {
      int[] a={1,2,3,4};
      fixed(int* g1=a, g2=&a[0])
      {
```

```
            Console.WriteLine((uint)g1);
            Console.WriteLine((uint)g2);
        }
    }
}
```

Gördüğünüz gibi bir diziyi bir göstericiye aktarmak için & operatörünü kullanmamıza gerek kalmadı. Çünkü aslında bütün diziler aynı zamanda bir göstericidir. Bu program sonucunda ekrana iki tane birbiriyle aynı adres yazılacaktır.

Yönetilmeyen diziler

Şimdiye kadar gördüğümüz diziler System.Array sınıfı türünden birer nesne olduklarına göre bunlar yönetilen dizilerdir. Çünkü sınıflar yönetilen bir türdür. Yapılar, enum sabitleri, vs. ise yönetilmeyen türlerdir. Yönetilmeyen türlerle ilgili işlem yapmak daha hızlıdır. Çünkü yönetilmeyen türlerde C# bellek sorumluluğunu bize verir. Hatırlarsanız C# bir dizinin eleman sayısı aşıldığında çalışma zamanında IndexOutOfRangeException istisnai durumunu veriyordu. Yani dizimizin kendine ait olmayan bir bellek bölgesine müdahale etmesini engelliyordu. Bu çoğu durumda son derece güzel bir özelliktir. Ancak istersek C#'ın böyle bir durumda çalışma hatası vermemesini sağlayabiliriz. Bu yönetilmeyen dizilerle mümkün olur. Yönetilmeyen diziler yönetilen dizilere oranla daha hızlı çalışır. Yönetilmeyen diziler stackalloc anahtar sözcüğüyle belirtilir. stackalloc bize bellekte istediğimiz kadar alan tahsis eder ve bu alanın başlangıç adresini bir gösterici olarak döndürür. Örnek:

```
int* dizi=stackalloc int[10];
```

Gördüğünüz gibi new anahtar sözcüğü yerine stackalloc anahtar sözcüğü gelmiş. Bu komut ile bellekte kendimize 10*sizeof(int)=40 baytlık alan ayırdık. Örnek program:

```
using System;
class Gosterici
{
   unsafe static void Main()
   {
      int* dizi=stackalloc int[10];
      for(int i=0;i<10;i++)
         Console.WriteLine("*(dizi+{0})={1}",i,dizi[i]);
   }
}
```

Şimdi başka bir örnek:

```
using System;
class Gosterici
{
   unsafe static void Main()
   {
      int* dizi=stackalloc int[10];
      for(int i=0;i<50;i++)
         Console.WriteLine("*(dizi+{0})={1}",i,dizi[i]);
   }
}
```

Bu programda dizinin sınırları aşılmış olmasına rağmen program çalışmaya devam eder. Ancak başka programların bellekteki verilerine de müdahale etmiş oluruz. Bir programcı için pek tavsiye edilmeyen bir durumdur. Ayrıca normal dizilerde olduğu gibi stackalloc bir dizinin eleman sayısının derlenme zamanı bilinmesi zorunlu değildir. İstersek dizinin eleman sayısını kullanıcı belirleyebilir.

Yapı türünden göstericiler

Şimdiye kadar int, double gibi temel veri türleri tipinden göstericiler oluşturduk. Bölümün en başında da söylediğimiz gibi bütün yapılardan gösterici oluşturulabilir. Ancak bir şart vardır: İlgili yapının içinde geri dönüş tipi bir sınıf olan özellik olmamalıdır. Geri dönüş tipi bir sınıf olan metot ya da sahte özellik olabilir. Buradan anlıyoruz ki int, double vb. yapıların içinde geri dönüş tipi bir sınıf olan özellik yokmuş.

-> operatörü

-> operatörü bir gösterici üzerinden ilgili yapının üye elemanlarına erişmek için kullanılır. Şimdi yukarıdaki bilgilerle bu bilgiyi bir örnek üzerinde görelim:

```
using System;
struct Yapi
{
   //string k; ---> Bu kod eklenseydi program derlenemezdi.
   public int x;
   int s;
   public Yapi(int x, int s)
   {
      this.x=x;
      this.s=s;
   }
   public string Metot()
   {
      return "Deneme";
   }
   public string Ozellik
   {
      set{}
      get{return "Deneme";}
   }
}
class Prg
{
   unsafe static void Main()
   {
      Yapi a=new Yapi(2,5);
      Yapi* b=&a;
      Console.WriteLine(b->Metot()); //Gördüğünüz gibi göstericiler
üzerinden ilgili yapının üye elemanlarına erişebiliyoruz.
      Console.WriteLine(b->Ozellik);
      Console.WriteLine(b->x);
   }
```

```
}
```

Bildiğiniz üzere nesneler üzerinden yapıların üye elemanlarına erişmek için . operatörünü kullanıyorduk. Yani -> ile . operatörleri bir bakıma benzeşiyorlar.

Göstericiler ve string türü

Stringlerin göstericiler için özel bir anlamı vardır. Char türünden herhangi bir göstericiye bir string atanabilir. Bu durumda göstericiye stringin ilk karakterinin bellekteki adresi atanır. Stringlerin karakterleri tıpkı diziler gibi bellekte ardışıl sıralanır. Dolayısıyla ilk karakterinin adresini bildiğimiz bir stringin tüm karakterlerini elde edebiliriz. Örnek:

```
using System;
class Stringler
{
   unsafe static void Main()
   {
      fixed(char* g="Vikikitap")
      {
         for(int i=0;g[i]!='\0';i++)
            Console.WriteLine(g[i]);
      }
   }
}
```

Gördüğünüz gibi string bir sınıf olduğu için fixed anahtar sözcüğü kullanıldı. Ayrıca programdaki `for(int i=0;g[i]!='\0';i++)` satırı dikkatinizi çekmiş olmalı. Buradaki '\0' karakteri stringin sonuna gizlice eklenenen bir karakterdir. Biz bu karakterden stringin bittiğini anlıyoruz.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Assembly kavramı

Derlenmiş exe ve dll dosyalarına assembly denir. Çalışan programımız içinde bir assembly olabileceği gibi birden fazla assembly de olabilir. Programımızda birden fazla assembly olması demek programımızın ilişkide olduğu bir ya da daha fazla dll dosyası mevcut demektir. System isim alanındaki AppDomain sınıfı çoklu assembly işlemleriyle ilgilenirken, System.Reflection isim alanındaki Assembly sınıfı ise tek bir assembly ile ilgili işlemler yapılmasını sağlar. Şimdi bu iki sınıfın çeşitli özelliklerini inceleyelim:

- AppDomain sınıfının static CurrentDomain özelliği o an üzerinde çalışılan assembly grubunu bir AppDomain nesnesi olarak döndürür.
- AppDomain sınıfının static olmayan GetAssemblies() metodu, ilgili AppDomain nesnesindeki bütün assemblyleri bir Assembly dizisi olarak tutar.
- Assembly sınıfının static olmayan Location özelliği ilgili assemblynin sabit diskteki yolunu string olarak verir. Şimdi bu teorik bilgileri bir örnek üzerinde görelim:

```
using System;
using System.Reflection;
class Assemblyler
{
   static void Main()
   {
      AppDomain ad=AppDomain.CurrentDomain;
      Assembly[] assembly=ad.GetAssemblies();
      Console.WriteLine("Toplam assembly sayısı: "+assembly.Length);
      for(int i=0;i<assembly.Length;i++)
         Console.WriteLine("Assembly yeri: "+assembly[i].Location);
   }
}
```

Benim bilgisayarımda bu programın ekran çıktısı şöyle oldu:

```
Toplam assembly sayısı: 2
Assembly yeri: C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\mscorlib.dll
Assembly yeri: C:\Documents and Settings\Bekir Oflaz\new2.exe
```

mscorlib.dll dosyası şimdiye kadar sıklıkla kullandığımız System isim alanı ve bu isim alanına bağlı alt isim alanlarının içinde bulunduğu dosyadır. C# derlediğimiz her dosyaya bu dosyayı otomatik olarak ilişkilendirir. Bu dosyanın yolu kullandığınız .Net Framework sürümüne bağlı olarak sizde değişebilir.

Bazen programımızın ilişkiye girdiği dll dosyasının da ilişkiye girdiği başka bir dll dosyası olabilir. Bu durumda AppDomain sınıfının CurrentDomain özelliğiyle oluşturulan AppDomain nesnesine söz konusu bütün dll dosyaları dâhildir. Ancak tabii ki farklı assemblyler aynı dll dosyasını kullanıyorsa yalnızca bir tane dll AppDomain nesnesine dâhil edilir. Ayrıca AppDomain sınıfının GetAssemblies() metodu ile oluşturulan Assembly dizisinde sıra esas programda karşılaşılma sırasına göredir. Eğer programda System isim alanı kullanılmışsa ilk öğe mscorlib.dll'dir. Ondan sonra gelen öğe esas programımız, bundan sonra gelen öğeler ise karşılaşılma sırasına göre dll dosyalarıdır. Eğer herhangi bir dll dosyasını programımıza ilişkilendirir ancak söz konusu dll'i programımızda kullanmazsak söz konusu assembly programımıza dâhil değilmiş gibi hesaba katılır.

- Assembly sınıfının static GetExecutingAssembly() metodu ile o an çalışan assemblynin kendisi bir Assembly nesnesi olarak döndürülür.

- Assembly sınıfının static olmayan EntryPoint özelliği ilgili assemblynin başlangıç metodunu MethodInfo sınıfı türünden bir nesne olarak döndürür. MethodInfo sınıfını ileride göreceğiz. Başlangıç metodu çalıştırılabilir exe dosyalarında genellikle Main() metodudur. Dll dosyalarında ise başlangıç metodu yoktur. Bu yüzden bu özellik dll dosyaları için null değeri döndürür. Şimdi bu iki bilgiyi örnekleyelim:

```
using System;
using System.Reflection;
class Assemblyler
{
   static void Main()
   {
      Assembly nesne=Assembly.GetExecutingAssembly();
      Console.WriteLine("Assemblynin başlangıç metodu:
"+nesne.EntryPoint);
   }
}
```

- Yeri gelmişken söylemek istiyorum. Bir sınıfın ya da yapının bir nesnesini Console.Write() ya da Console.WriteLine() metodu ile ekrana yazdırırsak aslında o nesnenin ToString() metodunun geri dönüş değerini ekrana yazdırmış oluruz. ToString() metodu normalde object sınıfına bağlıdır ve ilgili nesnenin türünü string olarak döndürür. Ancak tabii ki istersek kendi sınıf ya da yapılarımızda bu metodu override edip ilgili nesnemizin Console.Write() ya da Console.WriteLine() ile ekrana yazdırılırken nasıl davranması gerektiğini ayarlayabiliriz. Örnek:

```
using System;
class Sinif
{
   override public string ToString()
   {
      return "Deneme";
   }
}
class Ana
{
   static void Main()
   {
      Sinif a=new Sinif();
      Console.WriteLine(a);
   }
}
```

Bu program sonucunda ekrana `Deneme` yazılacaktır. İstersek Sinif sınıfına yeni üye elemanlar ekleyip ekrana yazdırılacak şeyin nesneye bağlı olmasını sağlayabiliriz. Bu örneğimizde ekrana yazılacak şey nesneye bağlı değildir. Bu ipucundan sonra asıl konumuza dönebililiriz.

- Assembly sınıfının static `LoadFrom(string assembly_yolu)` metodu bir assembly nesnesi döndürür. Örnek:

```
using System;
using System.Reflection;
class deneme
```

```
 {
    static void Main()
    {
       string
yol=@"C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\mscorlib.dll";
       Assembly nesne=Assembly.LoadFrom(yol);
       Console.WriteLine("Assemblynin başlangıç metodu:
"+nesne.EntryPoint);
    }
 }
```

Bu program sonucunda assemblynin başlangıç noktası olarak ekrana hiçbir şey yazılmayacaktır. Çünkü null demek yokluk demektir.

- Assembly sınıfının static olmayan GetTypes() metodu ilgili assemblydeki tüm türleri (sınıf, yapı, vb.) bir System.Type dizisi olarak döndürür. System.Type sınıfını ileride göreceğiz. Ancak şimdilik buna basit bir örnek verelim:

```
 using System;
 using System.Reflection;
 class deneme
 {
    static void Main()
    {
       AppDomain ad=AppDomain.CurrentDomain;
       Assembly[] assembly=ad.GetAssemblies();
       for(int i=0;i<assembly.Length;i++)
       {
          Console.WriteLine("Assembly: "+assembly[i].Location);
          Console.WriteLine("Tür sayısı:
"+assembly[i].GetTypes().Length);
          Console.WriteLine("*******");
       }
    }
 }
```

Bu program bende şu çıktıyı verdi:

```
Assembly: C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\mscorlib.dll
Tür sayısı: 2319
*******
Assembly: C:\Documents and Settings\Bekir Oflaz\new2.exe
Tür sayısı: 1
*******
```

Buradan anlıyoruz ki mscorlib.dll dosyasında tam 2319 tür varmış. Sizdeki .Net Framework sürümüne göre bu sayı değişebilir.

- Assembly sınıfının static olmayan `GetType(string Tür)` metodu ilgili assemblydeki belirli bir türü System.Type nesnesi olarak tutar. Örnek:

```
using System;
using System.Reflection;
class deneme
{
   static void Main()
   {
      string
yol=@"C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\mscorlib.dll";
      Assembly nesne=Assembly.LoadFrom(yol);
      Type a=nesne.GetType("System.Console");
      Console.WriteLine(a);
   }
}
```

Gördüğünüz gibi türü bağlı olduğu isim alanıyla belirtmemiz gerekir.

Bu kitabın diğer sayfaları

- C# hakkında temel bilgiler
- İlk programımız
- Değişkenler
- Tür dönüşümü
- Yorum ekleme
- Operatörler
- Akış kontrol mekanizmaları
- Rastgele sayı üretme
- Diziler
- Metotlar
- Sınıflar
- Operatör aşırı yükleme
- İndeksleyiciler
- Yapılar
- Enum sabitleri
- İsim alanları
- System isim alanı
- Temel I/O işlemleri
- Temel string işlemleri
- Kalıtım
- Arayüzler
- Partial (kısmi) tipler
- İstisnai durum yakalama mekanizması
- Temsilciler
- Olaylar
- Önişlemci komutları
- Göstericiler
- Assembly kavramı
- Yansıma
- Nitelikler
- Örnekler
- Şablon tipler
- Koleksiyonlar
- yield
- Veri tabanı işlemleri
- XML işlemleri
- Form tabanlı uygulamalar
- Visual Studio.NET
- Çok kanallı uygulamalar
- Linux'ta C# kullanımı
- Kaynakça
- Giriş sayfası
- **Ayrıca Bakınız:**

  ASP.NET

Yansıma

Geçen derste assembly kavramını ve assemblylerle kullanılan bazı sınıfları incelemiştik. Şimdi ise bir assemblydeki tür ve üye eleman bilgilerini çalışma zamanında elde etmek demek olan yansıma konusunu inceleyeceğiz. Konumuza tam giriş yapmadan önce bir operatör ve bir metodu tanıtmak istiyorum:

- **typeof() operatörü** herhangi bir türü System.Type türüne çevirip tutar. Örnek:

```
Type a=typeof(int);
```

- **GetType() metodu** herhangi bir nesnenin türünü System.RuntimeType türünden tutar. static değildir. object sınıfının bir metodudur. Dolayısıyla bütün türlere kalıtım yoluyla geçmiştir. Örnek:

```
int a;
Console.WriteLine(a.GetType());
```

Bu kod ekrana System.Int32 yazacaktır. System.RuntimeType sınıfı System.Type sınıfından türemiştir. Dolayısıyla bir Type nesnesine bir RuntimeType nesnesini atayabiliriz. Ancak System.RuntimeType sınıfı private olarak bildirilmiştir. Dolayısıyla System.RuntimeType sınıfını kodlarımızda kullanamayız. Şimdi esas konumuza başlayabiliriz.

Type sınıfı

Type sınıfı System isim alanında bulunan özet (abstract) bir sınıftır. Dolayısıyla

```
Type t=new Type();
```

gibi bir ifade mümkün değildir. Type türünden nesneler belirli bir türü temsil ederler. Type sınıfındaki static olmayan IsClass, IsAbstract, IsEnum, IsInterface, IsNotPublic, IsPublic, IsSealed ve IsValueType özellikleri vardır. Bunlar aşağıdaki koşullar sağlanıyorsa true, sağlanmıyorsa false değer döndürür:

IsClass → İlgili tür sınıfsa
IsAbstract → İlgili tür özetse
IsEnum → İlgili tür enumsa
IsInterface → İlgili tür arayüzse
IsNotPublic → İlgili tür public değilse
IsPublic → İlgili tür publicse
IsSealed → İlgili tür sealed ise
IsValuType → İlgili tür yönetilmeyen tür ise

Ayrıca Type sınıfının static olmayan IsAssignableFrom(Type tür) metodu parametredeki tür nesnesinin ilgili tür nesnesine atanıp atanamayacağının anlaşılması için kullanılır. Atanabiliyorsa true, atanamıyorsa false değer döndürür. Örnek:

```
//yavru sınıfının ana sınıfından türediğini varsayalım.
Type a=typeof(ana);
Type y=typeof(yavru);
Console.WriteLine(a.IsAssignableFrom(y));
Console.WriteLine(y.IsAssignableFrom(a));
```

Bu kod alt alta True ve False yazacaktır.

MemberInfo sınıfı

Bir türdeki üye elemanları temsil etmek için kullanılır. Özet bir sınıftır, dolayısıyla new operatörüyle yeni MemberInfo nesnesi tanımlanamaz. MemberInfo sınıfının önemli özellikleri:

- **Type DeclaringType** İlgili üye elemanın içinde bulunduğu türü Type türünden döndürür.
- **MemberTypes MemberType** İlgili üye elemanın ne tür bir eleman olduğunu MemberTypes enumu türünden döndürür. MemberTypes enumu Constructor, Method, Event, Field ve Property sözcüklerini içerir. Bunlar şu anlamlara gelir:

  Constructor → yapıcı metot

  Method → metot

  Event → olay

  Field → özellik

  Property → sahte özellik
- **string Name** ilgili üye elemanın ismini string olarak verir.

Type sınıfının GetMembers() metodu kullanılarak bir türdeki bütün üye elemanlar MemberInfo türünden bir diziye aktarılır. Ayrıca yine Type sınıfının GetMember(string eleman) metodu, ismi belirtilen string olan üye elemanları bir MemberInfo dizisi olarak döndürür. Örnek bir program:

```
using System;
using System.Reflection;
class deneme
{
   static void Main()
   {
      Type t=typeof(string);
      MemberInfo[] elemanlar=t.GetMembers();
      foreach(MemberInfo mi in elemanlar)
      {
         if(mi.MemberType==MemberTypes.Method)
            Console.WriteLine(mi.Name);
      }
   }
}
```

Bu program String sınıfındaki tüm metotları ekrana yazacaktır.

MethodInfo sınıfı

Türlerdeki metotları temsil etmek için kullanılan bir sınıftır. Bu sınıf MemberInfo sınıfından türemiş olan MethodBase sınıfından türemiştir. Dolayısıyla MemberInfo sınıfındaki tüm üye elemanları taşır. Bu sınıfta ek olarak metotların geri dönüş tipini Type türünden döndüren ReturnType özelliği ve metotların parametrelerini ParameterInfo sınıfı türünden bir dizi olarak döndüren GetParameters() metodu da bu sınıfta bulunan üye elemanlardandır. Type sınıfının GetMethods() metodu bir türdeki tüm metotları bir MethodInfo sınıfı dizisi türünden döndürür. Ayrıca Type sınıfının GetMethods(BindingFlags kriterler) metodu bir türdeki belirli kriterlere uyan metotları bir MethodInfo dizisi olarak döndürür. BindingFlags System.Reflection isim alanındaki bir enumdur. BindingFlags enumundaki önemli sözcükler:

DeclaredOnly → Yalnızca ilgili türün kendi metotları diziye aktarılır. Kalıtım yoluyla gelenler aktarılmaz.

NonPublic → public olarak işaretlenmemiş metotlar

Public → public olarak işaretlenmiş metotlar

Static → static olarak işaretlenmiş metotlar

- Type sınıfının GetMethod(string metotismi) metodu ilgili türdeki stringte verilen metodu MethodInfo türünden döndürür.
- MethodInfo sınıfındaki IsStatic özelliği ilgili metot staticse true değilse false değer döndürür. Örnek program:

```
using System;
using System.Reflection;
class Yansima
{
   static void Main()
   {
      Type t=typeof(Array);
      MethodInfo[] metotlar=t.GetMethods();
      foreach(MethodInfo mi in metotlar)
      {
         Console.WriteLine("Metot ismi: "+mi.Name);
         Console.WriteLine("Geri dönüş tipi: "+mi.ReturnType);
         Console.WriteLine(mi.IsStatic?"Static":"Static değil");
      }
      Console.WriteLine("Toplam metot: "+metotlar.Length);
   }
}
```

Farkındaysanız programımızda çok önceden gördüğümüz ?: operatörünü kullandık. Bu operatör if ile 3-4 satırda yazabileceğimiz kodu tek satıra indirdi. Başka bir örnek:

```
using System;
using System.Reflection;
class Yansima
{
   static void Main()
   {
      AppDomain ad=AppDomain.CurrentDomain;
      Assembly[] assembly=ad.GetAssemblies();
      Type[] tipler=assembly[0].GetTypes();
      foreach(Type tip in tipler)
      {
         Console.WriteLine("Tip: "+tip.Name);
         Console.WriteLine("Metot sayısı: "+tip.GetMethods().Length);
         Console.WriteLine();
      }
   }
}
```

MethodInfo sınıfının IsStatic özelliği yanında IsAbstract, IsConstructor, IsPublic, IsPrivate, IsVirtual özellikleri de mevcuttur.

ParameterInfo sınıfı

Metotların parametreleriyle ilgilenen bir sınıftır. Bir metodun parametrelerini elde etmek için MethodInfo sınıfının GetParameters() metodu kullanılabilir. MethodInfo sınıfının GetParameters() metodu bir ParameterInfo dizisi döndürür. ParameterInfo sınıfının ParameterType özelliği ilgili parametrenin türünü Type türünden döndürür. Yine ParameterInfo sınıfının Name özelliği ilgili parametrenin ismini string olarak döndürür.

ConstructorInfo sınıfı

Bir türün yapıcı metoduyla ilgilenir. Type sınıfının GetConstructors() metodu bu sınıf türünden bir dizi döndürür. Bu sınıfa ait en önemli metot GetParameters() tır. ParameterInfo sınıfı türünden bir dizi döndürür.

PropertyInfo, FieldInfo ve EventInfo sınıfları

Bu sınıflar sırasıyla bir türdeki sahte özellikler, normal özellikler ve olaylarla ilgilenir. Type sınıfının GetProperties(), GetFields() ve GetEvents() metotları bu sınıflar türünden dizi döndürürler. Her üç sınıfta da Name özelliği mevcuttur. EventInfo sınıfının şu önemli özellikleri mevcuttur.

- IsMulticast → Olayın multicast olup olmadığını verir.
- EventHandlerType → Olayın türünü RuntimeType türünden verir. (Bir temsilci)

NOT: Type sınıfının GetMethods(), GetProperties() gibi metotları yalnızca public üye elemanları döndürür.
NOT: Gördüğümüz MethodInfo, EventInfo gibi sınıfların tamamı System.Reflection isim alanındadır. Yalnızca Type System isim alanındadır. Yine konuyu işlerken karşımıza çıkan enumlar da System.Reflection isim alanındadır.

Çalışma zamanında metot çağırma

Şimdiye kadar metotlarımızı derleme zamanında çağırdık. Ancak artık System.Reflection isim alanındaki sınıflar sayesinde metotlarımızı çalışma zamanında çağırabileceğiz. Bunun için MethodInfo ve ConstructorInfo sınıflarında bulunan

```
object Invoke(object nesne,object[] parametreler)
```

metodunu kullanacağız. Eğer ConstructorInfo sınıfındaki metodu çağırırsak çalışma zamanında nesne oluşturmuş oluruz. Invoke() metodunun birinci parametresi metodun hangi nesne üzerinden çağrılacağını belirtir. İkinci parametre ise çağrılacak metoda verilecek parametreleri içeren object tipinden bir dizidir. Çalışma zamanında çağrılacak metot static ise birinci parametre null olmalıdır. Benzer şekilde çağrılacak metot parametre almıyorsa ikinci parametre null olmalıdır. Programımız şu şekilde:

```
using System;
using System.Reflection;
class Deneme
{
   public static int Topla(int a, int b)
   {
      return a+b;
   }
}
class Ana
{
   static void Main()
   {
```

```
        object[] parametreler={9,5};
        int a=(int)MetotInvoke("Topla","Deneme",parametreler);
        Console.WriteLine(a);
    }
    static object MetotInvoke(string metot,string tip,object[] args)
//tip ile metodun hangi türün içinde olduğunu anlıyoruz.
    {
        Type t=Type.GetType(tip); //Aldığımız string hâlindeki tipi Type
 türüne dönüştürüyoruz.
        if(t==null)
            throw new Exception("Tür bulunamadı");
        MethodInfo m=t.GetMethod(metot); //Aldığımız string hâlindeki
metodu MethodInfo türüne dönüştürüyoruz.
        if(m==null)
            throw new Exception("Metot bulunamadı");
        if(m.IsStatic)
            return m.Invoke(null,args);
        object o=Activator.CreateInstance(t);
        return m.Invoke(o,args);
    }
 }
```

Bu program sonucunda ekrana 14 yazılacaktır. Burada MetotInvoke() metodu kendimizin oluşturduğu bir metottur. Kendiniz bu metoda farklı isimler verebilirsiniz. Bu metodun Invoke() metoduna iletilecek veriler için bir "ulak" vazifesi görmektedir. Yani bu metodun yaptığı tek iş Invoke() metoduna gerekli dönüşümleri yaparak ve gerekli hata durumlarını göz önünde bulundurrarak gerekli verileri iletmektir. Programın kritik satırları şunlardır:

```
        if(m.IsStatic)
            return m.Invoke(null,args);
        object o=Activator.CreateInstance(t);
        return m.Invoke(o,args);
```

Birinci ve ikinci satırda çalıştırılmak istenen metodun static mi değil mi olduğu kontrol ediliyor. Eğer staticse metoda herhangi bir nesne üzerinden erişilmiyor. Üçüncü satırda System isim alanındaki Activator sınıfındaki CreateInstance() metoduyla yeni bir nesne oluşturuluyor. Bu metot parametre olarak nesnenin hangi türden oluşturulacağını belirten Type türden bir tür alır. Son satırda ise oluşturulan söz konusu nesneyle ve parametrelerle metot çağrılıyor.

Bu kitabın diğer sayfaları

- C# hakkında temel bilgiler
- İlk programımız
- Değişkenler
- Tür dönüşümü
- Yorum ekleme
- Operatörler
- Akış kontrol mekanizmaları
- Rastgele sayı üretme
- Diziler
- Metotlar
- Sınıflar
- Operatör aşırı yükleme
- İndeksleyiciler
- Yapılar
- Enum sabitleri
- İsim alanları
- System isim alanı
- Temel I/O işlemleri
- Temel string işlemleri
- Kalıtım
- Arayüzler
- Partial (kısmi) tipler
- İstisnai durum yakalama mekanizması
- Temsilciler
- Olaylar
- Önişlemci komutları
- Göstericiler
- Assembly kavramı
- Yansıma
- Nitelikler
- Örnekler
- Şablon tipler
- Koleksiyonlar
- yield
- Veri tabanı işlemleri
- XML işlemleri
- Form tabanlı uygulamalar
- Visual Studio.NET
- Çok kanallı uygulamalar
- Linux'ta C# kullanımı
- Kaynakça
- Giriş sayfası
- **Ayrıca Bakınız:**

ASP.NET

Nitelikler

Nitelikler koda bir eleman hakkında ekstra bilgi eklemek için kullanılan elemanın bildiriminden hemen önce yazılan özel kodlardır. C#'ta hazır nitelikler olduğu gibi kendi niteliklerimizi de yazabiliriz. Şimdi isterseniz önce hazır niteliklerden başlayalım.

Hazır nitelikler

Conditional niteliği

Bu nitelik yalnızca metotlarla kullanılır. Önişlemci komutlarında gördüğümüz sembol programda varsa ilişkili metot çağrılır, nitelikteki sembol programda yoksa metot çağrılmaz. Bu nitelik System.Diagnostics isim alanında bulunur. Örnek program:

```
#define TURKCE
using System;
using System.Diagnostics;
class Deneme
{
   [Conditional("TURKCE")]
   static void Turkce()
   {
      Console.WriteLine("Bu program Türkçedir.");
   }
   [Conditional("ENGLISH")]
   static void English()
   {
      Console.WriteLine("Bu program İngilizcedir.");
   }
   static void Main()
   {
      Turkce();
      English();
   }
}
```

Bu program ekrana programın Türkçe olduğunu yazacaktır. Yani program hata vermedi, yalnızca English() metodunun çalıştırılmasını engelledi. Çünkü ilgili sembol programda yoktu.

DllImport niteliği

Bu nitelik bir metodun .Net ortamında geliştirilmediğini belirtmek için kullanılır. .Net kütüphanesini değil de Windows API'lerini kullanabilmek için bu nitelik gereklidir. Ayrıca ilgili metodun dışarıdan alındığını belirtmek için extern anahtar sözcüğünü kullanırız. Bu nitelik System.Runtime.InteropServices isim alanında bulunur.

DllImport niteliğini bir Windows API fonksiyonu olan MessageBox() üzerinde görebiliriz. Bu fonksiyonu C# programımızda kullanabilmek için DllImport niteliği ile belirtmeli, ayrıca extern anahtar sözcüğü ile de ayrı bir belirtim yapmalıyız. MessageBox Windows programlarında sıkça görebileceğimiz uyarı penceresi anlamına gelir ve üzerinde çeşitli uyarı metinlerinin olduğu bir uyarı penceresi çıkarır. Kullanıcı ekrandaki "Tamam" ve/veya "İptal" gibi butonları tıklayabilir. Şimdi bunları kodsal olarak örnekleyelim.

```
using System;
using System.Runtime.InteropServices;
class Nitelikler
{
   [DllImport("User32.dll")]
   public static extern int MessageBox(int aile, string mesaj, string
baslik, int tip);
   static void Main()
   {
      MessageBox(0,"Bu bir uyarı mesajıdır","C#",1);
   }
}
```

Gördüğünüz gibi kendi metodumuzu oluşturmuş gibi gözüküyoruz. Ancak aslında tek yaptığımız şey User32.dll dosyasından ilgili fonksiyonu alıp programımıza bir metotmuş gibi geçirmek. MessageBox() metodunun son parametresini 0-6 aralığında bir tam sayı yapıp değişik buton kombinasyonları elde edebilirsiniz.

Obsolete niteliği

Bu nitelikle bir metodun çağrılması durumunda hata ya da uyarı verilmesi sağlanır. Önişlemci komutlarındaki #error ve #warning komutlarına benzer. İki kullanımı vardır:

```
[Obsolete("Uyarı mesajı")]
```

veya

```
[Obsolete("Uyarı mesajı", bool hata)]
```

Birinci örnekte eğer ilgili metot çağrılırsa derleyici uyarı verecektir. Yani program derlenecektir. İkinci örnekte ise eğer hata parametresi true ise program hata verecektir. Yani program derlenmeyecektir. Eğer ikinci örneğin hata parametresi false olursa birincisinden farkı kalmaz. İlk parametreler ise ekrana yazılan hata metnini belirlememizi sağlar. Obsolete niteliği System isim alanında bulunur.

AttributeUsage niteliği

Bu nitelik ile kendi yazdığımız niteliklerin hangi elemanlarla kullanılabileceğini belirleriz. Parametre olarak AttributeTargets isimli enum nesnesi almaktadır. Bu enumun içerdiği sözcükler şunlardır: All, Assembly, Class, Constructor, Delegate, Enum, Event, Field, Interface, Method, Module, Parameter, Property, ReturnValue, Struct. Bu sözcüklerden tek anlamını çıkaramadığınız tahminimce Module'dür. Module sözcüğü isim alanlarını ifade eder. Örnek

```
[AttributeUsage(AttributeTargets.Method)]
```

Burada niteliğimizin yalnızca metotlara uygulanabileceğini belirttik. Şimdilik buna kafa takmanıza gerek yok. Az sonra kendi niteliklerimizi oluştururken AttributeUsage niteliğini uygulamalı olarak göreceksiniz.

Kendi niteliklerimizi oluşturmak

Nitelikler aslında System.Attribute sınıfından türetilmiş sınıflardan başka bir şey değildir. O hâlde biz de kendi niteliklerimizi System.Attribute sınıfından türetme yaparak oluşturabiliriz. Örnek:

```
using System;
[AttributeUsage(AttributeTargets.Method)]
class KendiNiteligimiz:Attribute
{
   public string Mesaj;
   public KendiNiteligimiz(string mesaj)
   {
      Mesaj=mesaj;
   }
}
class AnaProgram
{
   [KendiNiteligimiz("Mesajımız")]
   static void Metot(){}
   static void Main(){}
}
```

Bu programda kendi niteliğimizi oluşturduk ve bu niteliğimizi bir metot üzerinde uyguladık. Niteliklerle ilgili diğer ilginç bir nokta ise niteliği kullanırken parametre verirken nitelikteki özelliklere değer verilebilmesidir. Örneğin:

```
[KendiNiteligimiz("Mesajımız",konu="C#",no=5.6)]
```

Burada niteliğimiz yalnızca bir parametre almasına rağmen diğer parametreler nitelikteki public özelliklere değer atamıştır. Bunu bir örnek üzerinde görelim:

```
using System;
[AttributeUsage(AttributeTargets.Method)]
class OzelNitelik:Attribute
{
   public string Mesaj;
   public int No;
   public string Konu;
   public OzelNitelik(string mesaj)
   {
      Mesaj=mesaj;
   }
}
class Deneme
{
   [OzelNitelik("Mesajımız",No=5,Konu="C#")]
   static void Metot(){}
   static void Main(){}
}
```

İlk parametre normal parametre olmalıdır. Diğerlerinden bazıları ya da tamamı olmayabilir. İlk parametre dışındakilerin kendi arasındaki sırası önemli değildir.

Bir elemana ilişkin nitelikleri yansıma yoluyla elde etmek için Attribute sınıfının static GetCustomAttribute() metodu kullanılabilir. Örnek:

```
using System;
class Yansima
{
   static void Main()
   {
      Type t=typeof(Array);
      Attribute[] dizi=Attribute.GetCustomAttributes(t);
      foreach(Attribute a in dizi)
         Console.WriteLine(a);
   }
}
```

Benim bilgisayarımda program şu çıktıyı verdi:

```
System.SerializableAttribute
System.Runtime.InteropServices.ComVisibleAttribute
```

Yani System.Array sınıfı bu iki niteliği kullanıyormuş.

Bu kitabın diğer sayfaları

- C# hakkında temel bilgiler
- İlk programımız
- Değişkenler
- Tür dönüşümü
- Yorum ekleme
- Operatörler
- Akış kontrol mekanizmaları
- Rastgele sayı üretme
- Diziler
- Metotlar
- Sınıflar
- Operatör aşırı yükleme
- İndeksleyiciler
- Yapılar
- Enum sabitleri
- İsim alanları
- System isim alanı
- Temel I/O işlemleri
- Temel string işlemleri
- Kalıtım
- Arayüzler
- Partial (kısmi) tipler
- İstisnai durum yakalama mekanizması
- Temsilciler
- Olaylar
- Önişlemci komutları
- Göstericiler
- Assembly kavramı
- Yansıma
- Nitelikler
- Örnekler
- Şablon tipler
- Koleksiyonlar
- yield
- Veri tabanı işlemleri
- XML işlemleri
- Form tabanlı uygulamalar
- Visual Studio.NET
- Çok kanallı uygulamalar
- Linux'ta C# kullanımı
- Kaynakça
- Giriş sayfası
- **Ayrıca Bakınız:**

  ASP.NET

Örnekler

Bu kısımda yeni bir şey öğrenmeyeceğiz. Sadece önceki gördüğümüz konuları pekiştirecek birkaç örnek yapacağız.

Örnek-1

Bu örneğimizde kendimizi C# programcılarının faydalanması için DLL hazırlayan bir programcı gibi düşüneceğiz. Bu DLL'de bir sınıf ve bu sınıfın içinde de iki metot olacak. Bu metotlardan birisi toplama, birisi çarpma yapacak. Ayrıca DLL'de bu iki metoda ek olarak bu iki metodun isimlerinin listesini tutan bir metot ve DLL'i kullanan programcının parametreye verdiği metoda göre ilgili metodun ne işe yaradığını string olarak tutan bir metot daha olacak. Şimdi DLL'imizi yazmaya başlayabiliriz.

```
using System;
using System.Reflection;
public class Islem
{
   [Nitelik("Toplamaya yarar")]
   public static int Topla(int a,int b)
   {
      return a+b;
   }
   [Nitelik("Çarpmaya yarar")]
   public static int Carp(int a,int b)
   {
      return a*b;
   }
   [Nitelik("Bu sınıftaki metotları listeler")]
   public static string MetotListesi()
   {
      Assembly a=Assembly.GetExecutingAssembly();
      Type[] t=a.GetTypes();
      MethodInfo[]
mi=t[0].GetMethods(BindingFlags.Static|BindingFlags.Public|BindingFlags.Instance|BindingFlags.DeclaredOnly);
      string GeriDonus="";
      foreach(MethodInfo m in mi)
         GeriDonus+=m.Name+"\n";
      return GeriDonus;
   }
   [Nitelik("Metotlar hakkında bilgi verir.")]
   public static string bilgi(string metot)
   {
      Type t=typeof(Islem);
      MethodInfo mi=t.GetMethod(metot);
      Attribute[] dizi=Attribute.GetCustomAttributes(mi);
      Nitelik a=(Nitelik)dizi[0];
      return ((mi.IsStatic)?"static ":"")+mi.ToString()+" ---
"+a.Bilgi;
   }
```

```
}
[AttributeUsage(AttributeTargets.Method)]
class Nitelik:Attribute
{
   public string Bilgi;
   public Nitelik(string bilgi)
   {
      Bilgi=bilgi;
   }
}
```

Bu DLL dosyasını edinen programcının bu DLL dosyasını refere ederek şöyle bir program yazdığını düşünürsek:

```
using System;
class ana
{
   static void Main()
   {
      Console.WriteLine(Islem.Topla(5,10));
      Console.WriteLine(Islem.Carp(5,10));
      Console.WriteLine(Islem.MetotListesi());
      Console.WriteLine(Islem.bilgi("Topla"));
      Console.WriteLine(Islem.bilgi("Carp"));
      Console.WriteLine(Islem.bilgi("MetotListesi"));
      Console.WriteLine(Islem.bilgi("bilgi"));
   }
}
```

Bu programın ekran çıktısı şöyle olur.

```
15
50
Topla
Carp
MetotListesi
bilgi

static Int32 Topla(Int32, Int32) --- Toplamaya yarar
static Int32 Carp(Int32, Int32) --- Çarpmaya yarar
static System.String MetotListesi() --- Bu sınıftaki metotları listeler
static System.String bilgi(System.String) --- Metotlar hakkında bilgi verir.
```

Programdaki en kritik nokta

```
Nitelik a=(Nitelik)dizi[0];
```

satırıdır. Burada Attribute türünden olan dizi[0], bu sınıftan türeyen Nitelik sınıfına bilinçli olarak dönüştürüldü. Bilinçsiz dönüştürülemezdi. Ayrıca şu satır da ilginizi çekmiş olmalı:

```
MethodInfo[]
mi=t[0].GetMethods(BindingFlags.Static|BindingFlags.Public|BindingFlags.Instance|BindingFlags.DeclaredOnly);
```

Bu satırda ilgili türdeki static, public, static olmayan ve türeyerek gelmemiş olan metotlar mi dizisine atanıyor. Aslında burada tek yapmak istediğimiz object sınıfından gelen metotları elemekti. Ancak bu metot yalnızca BindingFlags.DeclaredOnly'yi kabul etmez. Metotların diğer özelliklerinin de açık şekilde belirtilmesi gerekir. Eğer bu enumu hiç kullanmasaydık bu sefer tüm public metotlar diziye atanacaktı.

Örnek-2

Bu örneğimizde plugin (eklenti) tabanlı bir program yazacağız. Yani isteyen programcılar DLL yazarak programımızın yapabileceği şeyleri artırabilecekler. Sonra yazdıkları DLL'i programımıza tanıtacaklar. Tabii ki DLL'i tanıma işini programımız yapacak. DLL'i yazan kişi sadece programımıza DLL'i tanıma komutu verecek. Ancak programımızın tanıyacağı DLL'in bir standartı olacak. Örneğin metot isimleri, metotların geri dönüş tipleri vs. bizim istediğimiz gibi olacak. Bu yüzden içinde istediğimiz arayüzün olduğu DLL'i programcılar önce edinecek sonra bu arayüze göre DLL'i yazacaklar. Öbür türlü programımızda karmaşa oluşurdu. Şimdi öncelikle içinde arayüzün olduğu ve programcılara dağıtacağımız DLL'imizi yazalım:

Arayüz DLL'ini yazma

```
public interface Gereksinimler
{
   string Bilgi();
}
```

Bu dosyayı DLL olarak derleyelim. Gördüğünüz gibi programcıların yazacakları DLL Bilgi() isminde bir metot içermeli. Arayüzler konusundan hatırlayacağınız gibi bir arayüzün elemanları erişim belirleyicisi alamaz ve static olamaz. Ayrıca arayüzü uygulayan sınıf arayüzdeki üyeleri kendine public, static değil, doğru geri dönüş tipinde, parametre sayısı, sırası ve türleri de aynı olacak şekilde geçirmelidir.

Eklentiyi yazma

Şimdi kendimizi eklentiyi yazan programcı yerine koyalım. Arkadaşımızdan arayüz DLL'ini edindik ve bu DLL'e göre DLL'imizi yazacağız. Yine sınıfa Sinif ismini vermemiz gerektiğini arkadaşımızdan öğrendik.

```
public class Sinif:Gereksinimler
{
   public string Bilgi()
   {
      return "Eklentili versiyon.";
   }
}
```

Artık eklentiyi yazdık. Tabii ki bu DLL derlenirken arayüz DLL'i ile ilişkilendirilmelidir.

Esas programımızı yazma

Şimdi esas programımıza sıra geldi. Bu program eklenti yükleyip yüklemeyeceğimizi soracak. Eğer yüklemek istersek yolunu isteyecek. DLL'in yolunu verdikten sonra artık programımız ilgili DLL'deki kütüphaneyi kullanabilir hâle gelecek. Şimdi programı yazmaya başlayabiliriz:

```
using System;
using System.Reflection;
class AnaProgram
{
   static void Main()
   {
      Console.Write("Eklenti kurmak istiyor musunuz? (e|h): ");
      string a=Console.ReadLine();
      if(a=="h")
      {
         Console.WriteLine("Eklenti kurulmadı.");
         return;
      }
      else if(a=="e")
      {
         EklentiYukleme();
      }
      else
         throw new Exception("Yalnızca e veya h girmeliydiniz.");
   }
   static void EklentiYukleme()
   {
      Assembly a;
      try
      {
         Console.Write("Lütfen DLL dosyasının sabit diskteki adresini
girin: ");
         a=Assembly.LoadFrom(Console.ReadLine());
      }
      catch
      {
         Console.WriteLine("Belirttiğiniz dosya yok. Lütfen tekrar
deneyiniz.");
         EklentiYukleme();
         return;
      }
      Type[] t=a.GetTypes();
      MethodInfo[] mi=t[0].GetMethods();
      if(t[0].Name!="Sinif"||mi[0].ToString()!="System.String
Bilgi()")
         throw new Exception("Eklentideki elemanlar/türler uyumsuz");

Console.WriteLine(mi[0].Invoke(Activator.CreateInstance(t[0]),null));
```

```
    }
}
```

Bu program kullanıcıdan bir DLL dosyası adresi aldı. Bu adresteki DLL'i bir Assembly nesnesi olarak tuttu. Sonra bu Assembly nesnesindeki tipleri bir Type dizisi olarak tuttu. Sonra bu Type dizisinin ilk elemanını bir MethodInfo dizisi olarak tuttu. Sonra ilgili türün ve metot dizisinin ilk elemanının istenildiği gibi olup olmadığı kontrol edildi. Eğer istenilen tür ve metotsa ilgili metot çağrıldı. Eğer istenilen tür ve metot değilse hata verip programı sonlandırdık.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Şablon tipler

Şimdiye kadar bir sınıftaki tür bilgileri sınıf bildirimi esnasında belliydi. Ancak artık şablon tipler sayesinde sınıftaki istediğimiz türlerin nesne yaratımı sırasında belli olmasını sağlayacağız. Üstelik her nesne için söz konusu türler değişebilecek. Şablon tipler sayesinde türden bağımsız işlemler yapabileceğiz. Şimdi şablon tipleri bir örnek üzerinde görelim:

```
using System;
class Sinif<SablonTip>
{
   public SablonTip Ozellik;
}
class AnaProgram
{
   static void Main()
   {
      Sinif<int> s1=new Sinif<int>();
      s1.Ozellik=4;
      Sinif<string> s2=new Sinif<string>();
      s2.Ozellik="deneme";
      Console.WriteLine(s1.Ozellik+" "+s2.Ozellik);
   }
}
```

Bu programda s1 nesnesinin Ozellik özelliğinin geri dönüş tipi int, ancak s2 nesnesinin Ozellik özelliğinin geri dönüş tipi stringdir. Benzer şeyi metotların geri dönüş tipleri, metotların parametre tipleri vb. için de yapabilirdik. Bu programdaki Sinif sınıfı türden bağımsız bir sınıftır. Yani bu sınıfla istediğimiz türdeki verilerle çalışabiliriz. Bu

programcılar için büyük esneklik sağlar. Sınıflar, arayüzler, yapılar, temsilciler ve metotlar şablon tip olarak bildirilebilir.

Sınıf aşırı yükleme

Bir sınıf (veya yapı, arayüz vs.) birden fazla şablon tip alabilir. Örnek:

```
class Sinif<SablonTip1, SablonTip2>
```

Bu durumda bu sınıf türünden nesne şunun gibi yaratılır:

```
Sinif<int,string> s=new Sinif<int,string>();
```

Aynı isim alanında isimleri aynı olsa bile farklı sayıda şablon tipi olan sınıflar bildirilebilir. Buna sınıf aşırı yükleme denir.

Şablon tipler arasında türeme

Şablon tipler arasında türeme ile ilgili çok fazla kombinasyon düşünebiliriz. Ancak biz burada karşımıza en çok çıkacak iki kombinasyondan bahsedeceğiz:

```
class AnaSinif<T>
{
   //...
}
class YavruSinif<T,Y>:AnaSinif<T>
{
   //Gördüğünüz gibi yavru sınıf en az ana sınıfın şablon tipini
içerdi.
}
class YavruSinif2<T>:AnaSinif<int>
{
   /*Gördüğünüz gibi yavru sınıf ana sınıfın şablon tipine belirli bir
tür koydu. Böylelikle yavru sınıftaki ana sınıftan
   gelen T harfleri int olarak değiştirilecektir.*/
}
```

Şablon tipler ve arayüzler

Arayüzlerin de şablon tipli versiyonları yazılabilir. Örneğin System.Collections.Generic isim alanında birçok şablon tipli arayüz bulunmaktadır. Bunlardan birisi de IEnumerable arayüzünün şablon tipli versiyonudur. IEnumerable arayüzünün şablon tipli versiyonu aşağıdaki gibidir.

```
interface IEnumerable<T>:IEnumerable
{
   IEnumerator<T> GetEnumerator();
}
```

Buradan anlıyoruz ki bu arayüzü uygulayan sınıf geri dönüş tipi IEnumerator<T> olan GetEnumerator() metodunu ve IEnumerable arayüzünün içerdiği tüm üyeleri içermeli. Bu arayüzü bir sınıfta şöyle uygulayabiliriz:

```
class Sinif:IEnumerable<int>
{
```

```
    IEnumerator IEnumerable.GetEnumerator()
    {
        //...
    }
    IEnumerator<int> IEnumerable<int>.GetEnumerator()
    {
        //...
    }
}
```

Burada hem IEnumerable hem de IEnumerable<T> arayüzlerinin metotları uygulandı. Çünkü IEnumerable<T> arayüzü, IEnumerable arayüzünden türemişti.

Şablon tiplerin metotlara etkisi

Bildiğiniz gibi sınıf düzeyinde bir şablon tip belirlediğimizde bu şablon tip metotların geri dönüş tipine, parametre tiplerine koyulabilir. Ancak bunun sonucunda bazen bir çakışma oluşabilir. Örneğin:

```
class Sinif<T>
{
    public int Metot(T a)
    {
        return 0;
    }
    public int Metot(int a)
    {
        return 1;
    }
}
```

Bu sınıf türünden nesneyi şöyle tanımlayıp kullanırsak;

```
Sinif<int> a=new Sinif<int>();
Console.WriteLine(a.Metot(10));
```

Sizce ne olur? Bu gibi bir durumda Metot(int a) metodu çalışır. Yani normal tipli versiyon şablon tipli versiyonu gizler.
NOT: Şablon tipler metotlarla kullanıldığı gibi özellik, indeksleyici ve olaylarla da kullanılabilir.

default operatörü

Bildiğimiz gibi şablon tipler herhangi bir tipi temsil ederler. Bu yüzden C#, yalnızca bazı türlere özgü olan operatörler (+, -, *, ...) kullanılmasına izin vermediği gibi şablon tip türünden bir nesne yaratılıp bu nesneye bir değer verilmesine engel olur. Örnekler:

```
class Sinif<T>
{
    public int Metot(T a)
    {
        return a+2;
    }
```

```
}
```

Bu sınıf derlenmez. Çünkü T tipinin hangi tip olduğunu bilmiyoruz. + operatörü bu tip için aşırı yüklenmiş olmayabilir. Bu yüzden C# bu gibi tehlikeli durumları önlemek için bu tür bir kullanımı engeller. Başka bir kullanım:

```
class Sinif<T>
{
   public int Metot(T a)
   {
      T nesne=0;
   }
}
```

Yine burada da bir hata söz konusudur. Çünkü her tipten nesneye 0 atanmayabilir. Benzer şekilde yapı nesnelerine de null değer atanamaz. Gelelim default operatörüne. default operatörü bir şablon tipin varsayılan değerini elde etmek için kullanılır. Örnek:

```
class Sinif<T>
{
   public void Metot()
   {
      T nesne=default(T);
   }
}
```

Varsayılan değer bazı türler için 0 (int, short, float vs.) bazı türler için null (tüm sınıflar) bool türü için de false'tur.

Kısıtlar

Daha önce şablon tip türünden nesnelerle herhangi bir operatör kullanılamayacağını, bu şablon türden nesnelere herhangi bir sabit atanamayacağını vs. söylemiştik. Bunlara paralel olarak şablon tür nesneleriyle herhangi bir üye de (metot, özellik vs.) çalıştıramayız. Çünkü şablon tipin ne olduğunu bilmediğimiz için olmayan bir metodu çalıştırmaya çalışıyor olabiliriz. Tüm bunların sebebi aslında şablon sınıf türünden nesne yaratırken şablonu herhangi bir tür yapabilmemizdi. Eğer şablona koyulabilecek türleri kısıtlarsak bunları yapabiliriz. C#'ta şu kısıtlar bulunmaktadır:
struct şablon tip yalnızca yapılar olabilir.
class şablon tip yalnızca sınıflar olabilir.
new() şablon tip yalnızca nesnesi yaratılabilen tiplerden olabilir. (tür abstract, static, vb. olamaz)
türetme şablon tip mutlaka belirtilen bir türden türemiş olmalıdır.
interface şablon tip mutlaka belirtilen bir arayüzü uygulamalıdır.
Kısıtlar where anahtar sözcüğüyle yapılmaktadır. Şimdi türetme kısıtına bir örnek verelim:

```
class A{}
class B:A{}
class C:A{}
class D<T> where T:A{}
class AnaProgram
{
   static void Main()
   {
      D<B> nesne1=new D<B>();
```

```
        D<C> nesne2=new D<C>();
        //Aşağıdaki olmaz.
        //D<int> nesne3=new D<int>();
    }
}
```

Türeme kısıtı sayesinde şablon tip nesnesiyle bir metodu çağırabiliriz. Bunun içinde ana sınıfa o metodu koyarız, bu sınıftan türeyen yavru sınıflarda o metodu override ederiz. Son olarak şablon tipe ana sınıftan türeme zorunluluğu getiririz. Ana sınıftan türeyen bütün sınıflar söz konusu metodu içereceği için artık C#, bu metodu şablon tip nesneleri üzerinden çağırmamıza izin verecektir.

Ancak halen new operatörüyle şablon tip türünden nesne oluşturamayız. Bunun için şablon tipe new() kısıtını eklemeliyiz. Yani şablon tipe yalnızca nesnesi oluşturulabilen türler koyulabilecek. Örnek:

```
class Sinif<T> where T:new()
{}
```

Artık bu sınıfın içinde T türünden nesneleri new operatörüyle oluşturabiliriz. Tabii ki C# şablon tipe nesnesi oluşturulamayan bir tür koyulmasını engelleyecektir. Başka bir önemli kısıt ise arayüz kısıtıdır. Arayüz kısıtı sayesinde şablon tipe koyulan tipin mutlaka belirli bir arayüzü kullanmış olmasını sağlarız. Örnek program:

```
using System;
class Karsilastirici<T> where T:IComparable<T>
{
   public static int Karsilastir(T a,T b)
   {
      return a.CompareTo(b);
   }
}
class Program
{
   static void Main()
   {
      int s1=Karsilastirici<int>.Karsilastir(4,5);
      int s2=Karsilastirici<float>.Karsilastir(2.3f,2.3f);
      int s3=Karsilastirici<string>.Karsilastir("Ali","Veli");
      int
s4=Karsilastirici<DateTime>.Karsilastir(DateTime.Now,DateTime.Now.AddDays(1));
      Console.WriteLine("{0} {1} {2} {3}",s1,s2,s3,s4);
   }
}
```

Bu program ekrana `-1 0 -1 -1` çıktısını verecektir. .Net Framework kütüphanesindeki IComparable arayüzünde CompareTo() metodu bulunmaktadır. Dolayısıyla bu arayüzü uygulayan her sınıfta da bu metot bulunur. Bu sayede de C# şablon tip nesnesinden ilgili metodun çağrılmasına izin vermiştir. Diğer önemli iki kısıt ise şablon tipe yalnızca bir sınıf koyulabilmesini sağlayan class ve aynı şeyi yapılar için yapan struct kısıtlarıdır. Örnekler:

```
class Sinif1<T> where T:class{}
class Sinif2<T> where T:struct{}
```

Tabii ki bir şablon tipe birden fazla kısıt eklenebilir. Bu durumda kısıtlar virgülle ayrılır. Örnek:

```
class Sinif<T> where T:class,IComparable,new(){}
```

Kısıtlarda diğerlerinin sırası önemli değildir. Ancak -varsa- new() kısıtı en sonda olmalıdır. Bir sınıfa eklenen birden fazla şablon tip varsa her biri için ayrı ayrı kısıtlar koyulabilir. Örnek:

```
class Sinif<T,S> where T:IComparable,IEnumerable where S:AnaSinif
```

NOT: Main() metodu mutlaka şablon tipli olmayan bir sınıfın içinde olmalıdır. Ayrıca Main() metodu ya bir sınıfın içinde ya da bir yapının içinde olmalıdır. Diğer bir deyişle çalışabilir her program en az bir sınıf ya da yapı içermelidir.

Şablon tipli metotlar

Şimdiye kadar şablon tipler sınıf seviyesinde tanımlanmıştı. Halbuki şablon tipler metot ve temsilci düzeyinde de tanımlanabilir. Örnek:

```
using System;
class karsilastirma
{
   static void Main()
   {
      Console.WriteLine(EnBuyuk<int>(4,5));
      Console.WriteLine(EnBuyuk<string>("Ali","Veli"));
   }
   static T EnBuyuk<T>(T p1,T p2) where T:IComparable
   {
      T geridonus=p2;
      if(p2.CompareTo(p1)<0)
         geridonus=p1;
      return geridonus;
  }
}
```

Bu program alt alta 5 ve Veli yazacaktır.

Şablon tipi çıkarsama

Az önceki örneğin sadece Main() metodunu alalım:

```
   static void Main()
   {
      Console.WriteLine(EnBuyuk<int>(4,5));
      Console.WriteLine(EnBuyuk<string>("Ali","Veli"));
   }
```

Burada parametrelerin türleri belli olduğu hâlde ayrıca int ve string türlerini de belirttik. İstersek bunu şöyle de yazabilirdik:

```
   static void Main()
   {
      Console.WriteLine(EnBuyuk(4,5));
      Console.WriteLine(EnBuyuk("Ali","Veli"));
```

```
    }
```

Bu programda metot şöyle düşünecektir: "Benim parametrelerim T türünden, o hâlde yalnızca parametreme bakarak T'nin hangi tür olduğunu bulabilirim." Gördüğünüz gibi metotların aşırı yükleyerek saatlerce uğraşarak yazabileceğimiz programları şablon tipli metotlar kullanarak birkaç dakikada yazabiliyoruz.

Şablon tipli temsilciler

Temsilciler de şablon tip alabilirler. Bu sayede temsilcinin temsil edebileceği metot miktarını artırabiliriz. Örnek:

```
using System;
delegate T Temsilci<T>(T s1,T s2);
class deneme
{
   static int Metot1(int a,int b){return 0;}
   static string Metot2(string a,string b){return null;}
   static void Main()
   {
      Temsilci<int> nesne1=new Temsilci<int>(Metot1);
      Temsilci<string> nesne2=new Temsilci<string>(Metot2);
      Console.WriteLine(nesne1(1,2));
      Console.WriteLine(nesne2("w","q"));
   }
}
```

Temsilci şablon tipleri de kısıt alabilirler. Örnek:

```
delegate T Temsilci<T>(T s1,T s2) where T:struct
```

Burada T yalnızca bir yapı olabilir. Yani bu temsilcinin temsil edeceği metodun parametreleri ve geri dönüş tipi yalnızca bir yapı olabilir.

null değer alabilen yapı nesneleri

Bildiğiniz gibi yapı nesneleri null değer alamaz. Örneğin şu kod hatalıdır:

```
int a=null;
```

Ancak System isim alanındaki Nullable<T> yapısı sayesinde yapı nesnelerinin de null değer alabilmesini sağlayabiliriz. System.Nullable<T> yapısı şu gibidir:

```
public struct Nullable<T> where T:struct
{
   private T value;
   private bool hasValue;
   public T Value{get{...}}
   public bool HasValue{get{...}}
   public T GetValueOrDefault(){...}
}
```

Bu yapıya göre null değer alabilen yapı nesneleri şöyle oluşturulur:

```
Nullable<int> a=new Nullable<int>();
a=5;
a=null;
Nullable<double> b=new Nullable<double>(2.3);
Console.WriteLine("{0}{1}",a,b);
```

Gördüğünüz gibi değerler yapıcı metot yoluyla ya da normal yolla verilebiliyor. Nullable<T> yapısının GetValueOrDefault() metodu ilgili nesnenin değeri null ise ilgili nesnenin tipinin varsayılan değerini döndürür (int için 0), ilgili nesnenin değeri null değilse ilgili nesnenin değerini olduğu gibi döndürür. Yine aynı yapıya ait Value özelliği ilgili nesnenin değerini döndürür. Ancak az önceki metottan farklı olarak nesnenin değeri null ise null döndürür. Eğer bu özelliğin döndürdüğü değer null ise çalışma zamanı hatası oluşacaktır. Çünkü bir yapı nesnesi null alamaz. Nullable tipler daha çok veri tabanı işlemlerinde kullanılır. Çünkü veri tabanındaki int, double vb. gibi veri tipleri null değer alabilir. Nullable tipler veri tabanındaki verileri programımıza olduğu gibi aktarmak için kullanılabilir.

? işareti

C#'ı tasarlayan mühendisler bizim nullable tiplere çokça ihtiyaç duyabileceğimizi düşünmüş olacaklar ki ? işaretini geliştirmişler. ? takısı kısa yoldan nullable tipte nesne oluşturmak için kullanılır. Örneğin:

```
Nullable<double> d=10;
```

ile

```
double? d=5;
```

satırları birbirine denktir. Yani aşağıdaki gibi bir satır mümkündür:

```
double? d=null;
```

Nullable nesneleri normal nesnelere tür dönüştürme operatörünü kullanarak dönüştürebiliriz. Ancak nullable nesnenin değeri null ise çalışma zamanı hatası alırız. Örnek:

```
int? a=5;
int b=(int)a;
```

Benzer şekilde tür dönüşüm kurallarına uymak şartıyla farklı dönüşüm kombinasyonları da mümkündür:

```
int? a=5;
double b=(double)a;
```

Normal ve nullable nesneler arasında ters dönüşüm de mümkündür. Normal nesneler nullable nesnelere bilinçsiz olarak dönüşebilir. Örnek:

```
int a=5;
int? b=a;
```

Nullable nesneler operatörler ile kullanılabilir. Örnek:

```
int? a=5;
int? b=10;
int c=(int)(a+b);
```

Burada a+b ifadesinin ürettiği değer yine int? türünden olduğu için tür dönüşüm operatörü kullanıldı.

?? operatörü

?? operatörü Nullable<T> yapısındaki GetValueOrDefault() metoduna benzer şekilde çalışır. Örnek:

```
int? a=null;
int b=a??0;
```

Burada eğer a null ise ?? operatörü 0 değerini döndürür. ?? operatörünün döndürdüğü değer normal (nullable olmayan) tiptedir. Eğer ilgili nullable nesne null değilse olduğu değeri döndürür. Başka bir örnek:

```
int? a=null;
int b=a??50;
```

Burada ise eğer a null ise 50 döndürülür. Yani ?? operatöründe GetValueOrDefault() metodundan farklı olarak ilgili nesne null olduğunda döndürülecek değeri belirleyebiliyoruz.

Bu kitabın diğer sayfaları

- C# hakkında temel bilgiler
- İlk programımız
- Değişkenler
- Tür dönüşümü
- Yorum ekleme
- Operatörler
- Akış kontrol mekanizmaları
- Rastgele sayı üretme
- Diziler
- Metotlar
- Sınıflar
- Operatör aşırı yükleme
- İndeksleyiciler
- Yapılar
- Enum sabitleri
- İsim alanları
- System isim alanı
- Temel I/O işlemleri
- Temel string işlemleri
- Kalıtım
- Arayüzler
- Partial (kısmi) tipler
- İstisnai durum yakalama mekanizması
- Temsilciler
- Olaylar
- Önişlemci komutları
- Göstericiler
- Assembly kavramı
- Yansıma
- Nitelikler
- Örnekler
- Şablon tipler
- Koleksiyonlar
- yield
- Veri tabanı işlemleri
- XML işlemleri
- Form tabanlı uygulamalar
- Visual Studio.NET
- Çok kanallı uygulamalar
- Linux'ta C# kullanımı
- Kaynakça
- Giriş sayfası
- **Ayrıca Bakınız:**

  ASP.NET

Koleksiyonlar

Koleksiyonlar nesneleri gruplamaya yarayan sınıflardır. Yani bir koleksiyon nesnesi birden fazla nesneyi temsil eder. .Net Framework kütüphanesindeki koleksiyon sınıfları System.Collections isim alanındadır. Koleksiyonlarla diziler arasında şu farklar vardır:

- Diziler System.Array türünden birer nesnedir.
- Koleksiyonlar ise System.Collections isim alanındaki çeşitli sınıflar türünden birer nesnedir.
- Koleksiyonlar dizilere göre bize daha fazla özgürlük verir. Çünkü koleksiyonlar içinde her türden nesne tutabiliriz. Ayrıca koleksiyonların boyutu sabit değildir. Eleman eklendikçe boyut dinamik olarak artar. Koleksiyonlar bize bunun gibi benzer avantajlar da verir. Konuyu işledikçe bu avantajları göreceksiniz.

Koleksiyon sınıflarının tamamı IEnumerable arayüzünü uygulamıştır. Dolayısıyla tüm koleksiyon nesneleri foreach döngüsüyle kullanılabilir. Ayrıca bunun haricinde her koleksiyon sınıfı kendine özgü birtakım arayüzleri de uygulamış olabilir. Örneğin birazdan göreceğimiz ArrayList sınıfı IList ve ICollection arayüzlerini de uygulamıştır. Bu arayüzlerin her birinin farklı anlamları vardır. Şimdi bu arayüzleri görelim:
ICollection Bu arayüz bir koleksiyonda bulunabilecek en temel üye elemanları içerir. System.Collections isim alanındaki bütün koleksiyon sınıfları bu arayüzü uygulamıştır. ICollection arayüzü IEnumerable arayüzünden türemiştir. Dolayısıyla ICollection arayüzünü uygulayan bir sınıf IEnumerable arayüzünü de uygulamış demektir.
IComparer Bu arayüz iki nesnenin karşılaştırılabilmesini sağlayan Compare() metodunu içerir. Bu metot iki nesnenin nasıl karşılaştırılacağını belirler. Compare() metodunun prototipi şu şekildedir:

```
int Compare(object o1,object o2)
```

Compare metodu -1, 0 ya da 1 değeri döndürür. Karşılaştırma genellikle büyüklük-küçüklük ilişkisine göre yapılır.
IDictionary Bu arayüzü anahtar-kilit ilişkisini uygulayan koleksiyon sınıfları uygular. Anahtar-kilit ilişkisini daha sonra göreceğiz. IDictionary arayüzü ICollection arayüzünden türemiştir.
IDictionaryEnumerator IDictionary arayüzünün özelleşmiş bir şeklidir. İleride göreceğiz.
IEnumerable İlgili koleksiyon nesnesinin foreach döngüsüyle kullanılabilmesini sağlayan arayüzdür.
IEnumerator IEnumerable arayüzüyle son derece ilişkilidir. IEnumerable arayüzü IEnumerator arayüzünün kullanılmasını zorlar. Aslında foreach döngüsünün kurulabilmesi için gerekli üye elemanlar IEnumerator arayüzündedir. IEnumerable arayüzü sadece IEnumerator arayüzünü zorlamak için aracı vazifesi görür.
IList Koleksiyon nesnelerine bir indeks numarasıyla erişilebilmesini sağlayan arayüzdür. IList ICollection arayüzünden türemiştir.
IHashProvider İleride göreceğimiz Hashtable koleksiyon sınıfıyla ilgilidir. System.Collections isim alanındaki arayüzlerden sonra artık aynı isim alanındaki koleksiyon sınıflarına geçebiliriz. Bunlar ArrayList, Hashtable, SortedList, BitArray, Stack ve Queue sınıflarıdır. Şimdi bu sınıfları teker teker inceleyelim.

ArrayList sınıfı

Koleksiyon sınıflarının en genel amaçlı olanıdır. Dizilere oldukça benzer. Dizilerden tek farkı eleman sayısının sabit olmaması ve yalnızca object tipinden nesneleri saklamasıdır. ArrayList sınıfı ICollection, IEnumerable, IList ve ICloneable arayüzlerini uygular. ArrayList sınıfının aşırı yüklenmiş üç tane yapıcı metodu vardır. Bunlar:

```
ArrayList al=new ArrayList();
```

Bu şekilde kapasitesi olmayan bir ArrayList nesnesi oluşturulur. Tabii ki koleksiyona yeni nesneler eklendikçe bu kapasite dinamik olarak artacaktır.

```
ArrayList al=new ArrayList(50);
```

Bu şekilde 50 kapasiteli bir ArrayList nesnesi oluşturulur. Yine kapasite aşılırsa dinamik olarak artırılır.

```
ArrayList al=new ArrayList(ICollection ic);
```

Burada ICollection arayüzünü uygulamış herhangi bir sınıf nesnesi parametre olarak verilir.

Daha önceden dediğim gibi ArrayList sınıfı nesneleri object olarak tutar. Yani bu koleksiyondan nesneleri elde ettiğimizde object hâldedirler. Genellikle bu nesneleri kullanabilmek için orijinal türe tür dönüşüm operatörüyle dönüşüm yapmak gerekir. Bir ArrayList nesnesine nesne eklemek için Add() metodu kullanılır. Bu metot parametre olarak object nesnesi alır ve bu nesneyi koleksiyonun sonuna ekler. Aynı şekilde bir ArrayList koleksiyonundan eleman çıkarmak için Remove() metodu kullanılır. Parametre olarak verilen object nesnesi koleksiyondan çıkarılır. Eğer nesne bulunamazsa hata oluşmaz, program işleyişine devam eder. Ayrıca herhangi bir ArrayList koleksiyonunun kapasitesini öğrenmek için ArrayList sınıfının Capacity özelliği kullanılır. Şimdi bütün bu üye elemanları bir örnek üzerinde görelim:

```
using System;
using System.Collections;
class Koleksiyonlar
{
   static void Main()
   {
      ArrayList al=new ArrayList();
      Console.WriteLine("Kapasite: "+al.Capacity);
      al.Add(5);
      Console.WriteLine("Kapasite: "+al.Capacity);
      al.Add("Deneme");
      Console.WriteLine("Kapasite: "+al.Capacity);
      al.Add('D');
      Console.WriteLine("Kapasite: "+al.Capacity);
      al.Add(1.2f);
      Console.WriteLine("Kapasite: "+al.Capacity);
      al.Add(3.4);
      Console.WriteLine("Kapasite: "+al.Capacity);
      al.Add(1.2m);
      Console.WriteLine("Kapasite: "+al.Capacity);
      al.Add("Vikikitap");
      Console.WriteLine("Kapasite: "+al.Capacity);
      al.Add('e');
      Console.WriteLine("Kapasite: "+al.Capacity);
      al.Add(2);
      Console.WriteLine("Kapasite: "+al.Capacity);
      EkranaYaz(al);
      al.Remove("Deneme");
      Console.WriteLine("Kapasite: "+al.Capacity);
      EkranaYaz(al);
  }
  static void EkranaYaz(ArrayList al)
  {
     foreach(object o in al)
        Console.Write(o.ToString()+" ");
     Console.WriteLine();
```

```
    }
 }
```

Bu programın ekran çıktısı şöyle olmalıdır:

```
Kapasite: 0
Kapasite: 4
Kapasite: 4
Kapasite: 4
Kapasite: 4
Kapasite: 8
Kapasite: 8
Kapasite: 8
Kapasite: 8
Kapasite: 16
5 Deneme D 1,2 3,4 1,2 Vikikitap e 2
Kapasite: 16
5 D 1,2 3,4 1,2 Vikikitap e 2
```

Bu ekran çıktısından şunları anlıyoruz:

- Kapasite başta 0, sonra 4, sonra da gerekli olduğunda kapasite ikişer kat artırılıyor.
- Remove() metoduyla koleksiyonunun elemanını silmek kapasiteyi değiştirmiyor. Yani Remove() metoduyla yapılan şey koleksiyondan bir nesne silip onun yerine sona bir null eklemek.

NOT: C#'ta geçersiz türler arasında dönüşüm yapmak yasaktır. Örneğin bir object nesnesinin gizli türü string değilse bunu

```
string s=(string)o;
```

şeklinde stringe çeviremeyiz. Bir ArrayList koleksiyonuna nesne yerleştirmek (araya sıkıştırmak) için Insert metodu kullanılır. Prototipi şu şekildedir:

```
void Insert(int indeks,object deger)
```

Bu metotla ilgili koleksiyonun indeks nolu elemanına deger koyulur. Bu indeksteki önceki eleman ve bu indeksten sonraki elemanlar ileriye bir birim ötelenir. Bildiğimiz üzere bu normal dizilerde mümkün değildi. Sabit kapasiteli ArrayList koleksiyonu oluşturmak içinse ArrayList sınıfının static FixedSize metodu kullanılır. Örnek:

```
ArrayList al=ArrayList.FixedSize(ArrayList koleksiyon);
```

Yani burada önceden tanımlanmış bir ArrayList nesnesi başka bir ArrayList nesnesine sabit kapasiteli olarak atanıyor. Eğer al koleksiyonunun kapasitesi aşılırsa tıpkı dizilerdeki gibi istisnai durum meydana gelir.
NOT: ArrayList koleksiyonunun elemanlarına tıpkı dizilerdeki gibi indeksleyici ile de erişilebilir.
Şimdi istersek ArrayList sınıfının diğer önemli üye elemanlarına bir göz atalım. Önce özellikler:
Count İlgili ArrayList nesnesindeki toplam eleman sayısını verir. Capacity özelliği ile karıştırılmamalıdır. Count özelliği "doldurulmuş" eleman sayısını verir. Geri dönüş tipi inttir.
IsFixedSize İlgili ArrayList nesnesinin sabit kapasiteli olup olmadığını verir. Eğer sabit kapasiteli ise true, değilse false değer döndürür.
IsReadOnly İlgili ArrayList nesnesinin salt okunur olup olmadığını verir. Eğer salt-okunur ise true, değilse false değer döndürür. Salt-okunur ArrayList nesneleri elde etmek için ArrayList sınıfının static ReadOnly() metodu kullanılır. Kullanımı FixedSize() metodu ile aynıdır. Sıra geldi metotlara:

AddRange(ICollection ic) Bu metot ile parametredeki koleksiyonun tüm elemanları ilgili koleksiyonun sonuna eklenir. Parametrenin ICollection tipinden olması demek bu sınıfın ICollection arayüzünü uygulaması gerekliliğini belirtir. Bildiğiniz gibi System.Generic isim alanındaki bütün koleksiyonlar ICollection arayüzünü uygulamıştır.
Sort() Koleksiyonları sıralamak için kullanılır. Eğer koleksiyon salt-okunur ise istisnai durum oluşur. Üç farklı aşırı yüklenmiş hâli vardır:

void Sort() İlgili koleksiyonu ilgili koleksiyonun Compare() metodu kullanarak sıralar. Tabii ki ilgili koleksiyon IComparer arayüzünü uygulamalıdır.

void Sort(IComparer ic) IComparer arayüzünü uygulayan başka bir sınıfın Compare() metodu kullanılarak sıralama yapılır.

void Sort(int indeks,int adet,IComparer ic) Üsttekinden tek farkı bunda sadece indeksten itibaren adet kadar elemanın sıralanmasıdır.

int BinarySearch(object o) Sıralanmış koleksiyonlarda arama yapılır. Eğer o nesnesi bulunursa bulunulan indeksle geri dönülür, bulunamazsa negatif bir sayıyla geri dönülür. Bu metodun aşağıdaki gibi iki versiyonu daha vardır:

int BinarySearch(object o,IComparer ic) Arama için başka bir sınıfın Compare() metodu kullanılır. Bu metot daha çok kullanıcının tanımladığı türden nesneler içeren koleksiyonlarda arama yapmak için kullanılır.

int BinarySearch(int indeks,int adet,object o,IComparer ic) Üstekinden tek farkı indeksten itibaren adet kadar eleman içinde arama yapılmasıdır.

void Reverse() İlgili koleksiyon ters çevrilir. Başka bir kullanımı daha vardır.

void Reverse(int indeks,int adet) Ters çevirme işlemi yalnızca indeksten itibaren adet kadar elemana uygulanır.

void Clear() İlgili koleksiyonun tüm elemanları temizlenir. Dolayısıyla Count özelliği sıfırlanırken Capacity özelliğinin değeri değişmez.
void TrimToSize() Koleksiyonun kapasitesi eleman sayısından fazla olduğu durumlarda gereksiz kısımlar silinir. Dolayısıyla kapasiteyle eleman sayısı eşitlenir.
bool Contains(object o) Parametre ile verilen nesnenin ilgili koleksiyonda bulunup bulunmadığı kontrol edilir. Eğer varsa true, yoksa false değer döndürür.
void CopyTo() ArrayList nesnesindeki elemanları normal tek boyutlu dizilere (System.Array) kopyalamak için kullanılır. Üç farklı kullanımı vardır:

void CopyTo(Array a) Metodu çağıran ArrayList nesnesindeki tüm elemanlar a dizisine ilk indeksten itibaren kopyalanır.

void CopyTo(Array a,int indeks) Üsttekiyle aynı göreve sahiptir. Tek farkı kopyalamanın indeks. elemandan itibaren yapılmasıdır.

void CopyTo(int indeks,Array a,int dizi_indeks,int adet) Metodu çağıran ArrayList nesnesindeki indeksten itibaren adet kadar eleman a dizisinde dizi_indeks indeksli eleman ve sonrasına kopyalanır.

ArrayList GetRange(int indeks,int adet) Metodu çağıran ArrayList nesnesinin indeks nolu elemanından itibaren adet kadar elemanı yeni bir ArrayList nesnesi olarak tutulur.
int IndexOf(object o) Parametre olarak verilen o nesnesinin ilgili ArrayList nesnesinde ilk bulunduğu indeksi döndürür. Object nesnesi bulunamazsa negatif değer döndürülür.
object[] ToArray() İlgili ArrayList nesnesindeki elemanları bir object dizisi olarak tutar. Farklı bir versiyonu daha vardır:

Array ToArray(Type t) İlgili ArrayList nesnesindeki elemanları t türünden bir dizi olarak tutar. Yani gerekli tür dönüşüm işlemleri otomatik olarak yapılır.

void RemoveAt(int indeks) İlgili ArrayList nesnesindeki indeks nolu elemanı siler. Belirtilen indeks 0'dan küçük ya da koleksiyonun Count değerinden büyükse istisnai durum oluşur.

Hashtable sınıfı

Array ve ArrayList sınıflarında dizideki elemanlara belirli bir indeks numarası ile erişiriz. Ancak bu bazen uygun yöntem olmayabilir. Çünkü hangi nesnenin hangi indekse karşılık geldiğini bilmemiz gerekir. Halbuki System.Collections isim alanındaki Hashtable sınıfında koleksiyonumuzun her nesnesine istediğimiz bir ismi verip koleksiyonumuzun elemanlarına bu isimlerle erişebiliriz. Bunun diğer adı anahtar-değer ilişkisidir. Yani bir anahtarla yalnızca bir nesneyi ifade ederiz. Örnek:

```
Hashtable h=new Hashtable();
h.Add("Anahtar1","Değer1");
h.Add("Anahtar2","Değer2");
```

Gördüğünüz gibi her Hashtable nesnesinin iki kısmı var. Bunlardan birisi ilgili nesnenin ismi (anahtar) ötekisi de nesnenin kendisi (değer). Bu ikisinin tipi string olmak zorunda değildir. istenilen tür olabilir. İlgili Hashtable nesnesine eleman eklerken Add() metodunda ilk parametre anahtar, ikinci parametre değerdir. Hashtable nesnesine yukarıdakinin yanı sıra aşağıdaki gibi de eleman ekleyebiliriz:

```
h["Anahtar1"]="Değer1";
h["Anahtar2"]="Değer2";
```

İstersek daha sonra Hashtable nesnesindeki anahtarların değerlerini değiştirebiliriz. Örnek:

```
h["Anahtar1"]="Yeni Değer";
```

Burada Anahtar1'in değerini değiştirdik. Add() metoduyla var olan bir anahtarı eklemeye çalışmak istisnai durum oluşmasına neden olur. Hashtable sınıfının Count özelliği ilgili Hashtable nesnesindeki anahtar-değer çifti sayısını döndürür. Ayrıca bir Hashtable nesnesindeki tüm anahtarları elde etmek için Keys özelliği kullanılır. Bu özellik ilgili Hashtable nesnesindeki tüm anahtarları bir ICollection nesnesi (bir koleksiyon) olarak döndürür. Values özelliği ise aynı şeyi değerler için yapar. Örnek program:

```
using System;
using System.Collections;
class hashtable
{
   static void Main()
   {
      Hashtable Sozluk=new Hashtable();
      Sozluk.Add("araba","car");
      Sozluk["kitap"]="book";
      ICollection anahtarlar=Sozluk.Keys;
      foreach(string s in anahtarlar)
         Console.WriteLine(s);
      ICollection degerler=Sozluk.Values;
      foreach(string s in degerler)
         Console.WriteLine(s);
      Console.WriteLine("Toplam kelime: "+Sozluk.Count);
   }
}
```

Bu programın çıktısı şu şekildedir:

```
kitap
araba
book
car
Toplam kelime: 2
```

Bu programı şöyle de yazabiliriz:

```
using System;
using System.Collections;
class hashtable
{
   static void Main()
   {
      Hashtable Sozluk=new Hashtable();
      Sozluk.Add("araba","car");
      Sozluk["kitap"]="book";
      IDictionaryEnumerator ide=Sozluk.GetEnumerator();
      while(ide.MoveNext())
         Console.WriteLine(ide.Key+" = "+ide.Value);
      Console.WriteLine("Toplam kelime: "+Sozluk.Count);
   }
}
```

Yani Hashtable sınıfının static olmayan GetEnumerator() metodu IDictionaryEnumerator arayüzü türünden bir nesne döndürüyor. Bu nesne üzerinden MoveNext() metoduna erişiyoruz. MoveNext() metodu ilgili IDictionaryEnumerator nesnesinin temsil ettiği anahtar-değer çiftini bir öteleyip true döndürüyor. Eğer öteleyemiyorsa false döndürüyor. Bir Hashtable nesnesinden bir anahtar-değer çiftini silmek için Hashtable sınıfının Remove() metodu kullanılır. Parametre olarak anahtarı alır. Örnek:

```
Sozluk.Remove("araba");
```

Bu satır Sozluk koleksiyonundaki "araba" anahtarını ve buna karşılık gelen değeri silecektir. Eğer ilgili koleksiyon salt-okunur ise, sabit kapasiteli ise, belirtilen anahtar yok ya da değeri null ise istisnai durum oluşur. IsReadOnly ve IsFixedSize özellikleri Hashtable sınıfında da bulunmaktadır. Hashtable sınıfının diğer önemli metotları:

void Clear() İlgili Hashtable nesnesindeki tüm anahtar-değer çiftlerini siler. Dolayısıyla Hashtable nesnesinin Count özelliğinin değeri 0 olur.

bool ContainsKey(object anahtar) Parametre ile verilen anahtar ilgili Hashtable nesnesinde varsa true, yoksa false değer döndürür. Aynı sınıftaki Contains() metodu da aynı işleve sahiptir.

bool ContainsValue(object deger) Parametre ile verilen değer ilgili Hashtable nesnesinde varsa true, yoksa false değer döndürür.

void CopyTo(Array dizi,int indeks) İlgili Hashtable nesnesindeki anahtar-değer çiftlerini bir boyutlu Array dizisine indeks indeksinden itibaren kopyalar. Kopyalama işleminin yapıldığı dizi DictionaryEntry yapısı türündendir. Bu yapıdaki Key ve Value özellikleri ilgili Hashtable nesnesindeki anahtar ve değerleri tutar. İlgili Hashtable nesnesindeki sadece anahtarları bir diziye kopyalamak için;

```
hashtable.Keys.CopyTo(dizi,indeks);
```

Aynı işlemi sadece değerler için yapmak içinse;

```
hashtable.Values.CopyTo(dizi,indeks);
```

SortedList sınıfı

SortedList koleksiyonunda elemanlara hem anahtar-değer çiftli hem de ArrayList sınıfında olduğu gibi indeksli olarak erişebiliriz. Ancak SortedList sınıfının önceki gördüğümüz iki sınıftan en büyük farkı elemanların daima sıralı olmasıdır. Çeşitli şekillerde SortedList nesneleri oluşturulabilir:

```
SortedList sl1=new SortedList();
SortedList sl2=new SortedList(50);
SortedList sl3=new SortedList(IDictionary id);
```

Birincisinde henüz kapasitesi olmayan bir SortedList nesnesi oluşturulur. İkincisinde başlangıç kapasitesi 50 olan bir SortedList nesnesi oluşturulur. Üçüncüsünde ise IDictionary arayüzünü uygulamış olan bir sınıfın nesnesi parametre olarak alınır ve bu sınıfın (koleksiyonun) elemanlarından SortedList nesnesi oluşturulur. Daha önce gördüğümüz Hashtable sınıfı IDictionary arayüzünü uygulamıştır. Üçünde de kapasite aşılırsa otomatik olarak kapasite artırılır. SortedList sınıfında Hashtable sınıfındaki Add(), Clear(), Contains(), ContainsKey(), ContainsValue(), GetEnumerator() ve Remove() metotları bulunur. Kendine özgü metotları ise

object GetByIndex(int indeks) İlgili SortedList nesnesindeki indeks nolu anahtar-değer çiftinin değer kısmı döndürülür.

object GetKey(int indeks) İlgili SortedList nesnesindeki indeks nolu anahtar-değer çiftinin anahtar kısmı döndürülür.

IList GetKeyList() İlgili SortedList nesnesindeki anahtarlar IList arayüzü türünden döndürülür.

IList GetValueList() İlgili SortedList nesnesindeki değerler IList arayüzü türünden döndürülür.

int IndexOfKey(object anahtar) İlgili SortedList nesnesindeki parametrede verilen anahtarın hangi indekste olduğu döndürülür. Anahtar bulunamazsa negatif değer döndürülür.

int IndexOfValue(object deger) Yukarıdaki şeyi değerler için yapar.

void SetByIndex(int indeks,object deger) İlgili SortedList nesnesindeki indeks nolu elemanın değeri deger olarak değiştirilir. Anahtarda değişme olmaz.

void TrimToSize() İlgili SortedList nesnesinin kapasitesi eleman sayısına eşitlenir. SortedList sınıfının özellikleri ise Capacity, Count, IsFixed, IsReadOnly, Keys ve Values'dir. Keys ve Values özellikleri ile Hashtable sınıfında olduğu gibi ICollection nesnesi elde edilir. İlgili SortedList nesnesindeki elemanlara erişmek için Keys ve Values özellikleri kullanılabileceği gibi DictionaryEntry yapısı da kullanılabilir. Bu yapı ilgili koleksiyondaki değer ve anahtarları temsil eden özellikler içerir. Örneğe geçmeden önce SortedList sınıfının sıralamayı nasıl yaptığını söylemek istiyorum:

Herhangi bir SortedList nesnesiyle her işlem yapıldıktan sonra ilgili koleksiyon sıralanır. Sıralama anahtarlara göredir. Dolayısıyla biz anahtarların sırasını değiştiremeyiz. Şimdi SortedList sınıfıyla ilgili bir örnek yapalım:

```
using System;
using System.Collections;
class sortedlist
{
   static void Main()
   {
      string[] kelimeler={"araba","masa","defter","kitap","okul"};
      SortedList sozluk=new SortedList();
      sozluk.Add("araba","car");
      sozluk.Add("masa","table");
```

```
        sozluk.Add("kalem","pencil");
        sozluk["kitap"]="book";
        sozluk["bilgisayar"]="computer";
        EkranaYaz(sozluk);
        for(int i=0;i<kelimeler.Length;i++)
           if(sozluk.ContainsKey(kelimeler[i]))
              Console.WriteLine(kelimeler[i]+"="+sozluk[kelimeler[i]]);
           else
              Console.WriteLine(kelimeler[i]+" sözlükte bulunamadı.");
        Console.WriteLine();
        sozluk.Add("doğru","line");
        EkranaYaz(sozluk);
        Console.WriteLine();
        Console.WriteLine("Sıralı listede toplam {0} eleman
bulunmaktadır.",sozluk.Count);
        Console.WriteLine("Bu elemanlar: ");
        ICollection anahtarlar=sozluk.Keys;
        foreach(string s in anahtarlar)
           Console.Write(s+"-");
     }
     static void EkranaYaz(SortedList sl)
     {
        foreach(DictionaryEntry de in sl)
           Console.WriteLine("{0,-12}:{1,-12}",de.Key,de.Value);
        Console.WriteLine();
     }
  }
```

BitArray sınıfı

BitArray içinde yalnızca bool türünden nesneler saklayabilen bir koleksiyondur. Bir BitArray nesnesi şu yollarla oluşturulabilir.

```
BitArray ba1=new BitArray(BitArray ba2);
BitArray ba3=new BitArray(bool[] b);
BitArray ba4=new BitArray(byte[] by);
BitArray ba5=new BitArray(int[] i);
BitArray ba6=new BitArray(int boyut);
BitArray ba7=new BitArray(int boyut,bool deger);
```

Birincisinde başka bir BitArray nesnesinden yeni bir BitArray nesnesi oluşturulur. Yani tabiri caizse kopyalama işlemi yapılmış olur. İkincisinde bir bool dizisinden yeni bir BitArray nesnesi oluşturulur.

Üçüncüsünde bir byte dizisinden yeni bir BitArray nesnesi oluşturulur. Bir bayt 8 bittir. Yani bir baytın içinde 8 tane 1 veya 0 vardır. İşte bu 1 veya 0'lar BitArray koleksiyonuna true ve false olarak geçirilir. Yani son tahlilde BitArray dizisindeki eleman sayısı 8*by.length olur.

Dördüncüsünde bir int dizisinden yeni bir BitArray nesnesi oluşturulur. Bir int nesnesinin içinde 32 tane 1 veya 0 vardır. Çünk int 32 bitlik bir veri tipidir. Bu 1 veya 0'lar BitArray koleksiyonuna true veya false olarak geçirilir. Yani son tahlilde BitArray dizisindeki eleman sayısı 32*i.Length olur.

Beşincisinde boyut tane false eleman içeren bir BitArray nesnesi oluşturulur. Altıncısında boyut tane degerden oluşan bir BitArray nesnesi oluşturulur.

BitArray sınıfı ICollection ve IEnumerable arayüzlerini uygulamıştır. Örnek bir program:

```
using System;
using System.Collections;
class bitarray
{
   static void Main()
   {
      int[] i={1};
      BitArray ba=new BitArray(i);
      EkranaYaz(ba);
   }
   static void EkranaYaz(BitArray ba)
   {
      IEnumerator ie=ba.GetEnumerator();
      int i=0;
      while(ie.MoveNext())
      {
         i++;
         Console.Write("{0,-6}",ie.Current);
         if(i%8==0)
            Console.WriteLine();
      }
      Console.WriteLine();
   }
}
```

Bu programın ekran çıktısı şöyle olmalıdır:

```
True  False False False False False False False
False False False False False False False False
False False False False False False False False
False False False False False False False False
```

Bu programda ba'yı ekrana klasik foreach ile de yazdırabilirdik. Ancak böyle bir yöntemin de olduğunu bilmenizde fayda var.

BitArray sınıfının Count ve Length adlı iki özelliği vardır. İkisi de ilgili BitArray nesnesinin eleman sayısını verir. Ancak Count salt-okunur olmasına rağmen Length değildir. Yani Length özelliği ile ilgili BitArray nesnesinin eleman sayısını değiştirebiliriz. Eğer var olan eleman sayısını azaltırsak sonundan gerekli sayıda eleman silinecektir. Eğer eleman sayısını artırısak sona gerekli sayıda false eleman eklenir. BitArray sınıfının önemli metotları:
bool Get(int indeks) İlgili BitArray nesnesindeki parametre ile verilen indeksteki eleman geri döndürülür.
void Set(int indeks,bool deger) İlgili BitArray nesnesindeki parametre ile verilen indeksteki eleman deger olarak değiştirilir.
void SetAll(bool deger) İlgili BitArray nesnesindeki tüm elemanlar deger olarak değiştirilir.
BitArray And(BitArray ba) İlgili BitArray nesnesindeki elemanlarla parametredeki BitArray nesnesindeki elemanları karşılıklı olarak VE işlemine sokar. Sonucu yine bir BitArray nesnesi olarak tutar. Eğer iki nesnenin eleman sayıları eşit değilse istisnai durum oluşur.

BitArray Or(BitArray ba) And() metoduyla aynı mantıkta çalışır. Tek değişen elemanların VE değil de VEYA işlemine sokulmasıdır.
BitArray Xor(BitArray ba) And() ve Or() metotlarıyla aynı mantıkta çalışır. Tek değişen elemanların ÖZEL VEYA işlemine sokulmasıdır.
BitArray Not() İlgili BitArray nesnesindeki elemanların değilini alır ve yeni bir BitArray nesnesi olarak döndürür.

Stack sınıfı

System.Collections isim alanındaki Stack sınıfı bilgisayar bilimlerinde çokça karşımıza çıkabilecek yığınlarla çalışmamızı sağlar. Yığınları üst üste koyulmuş kitaplar gibi düşünebiliriz. En üstteki kitabı rahatlıkla alıp okuyabiliriz. Ancak okuyacağımız kitap ortalarda veya altlarda ise önce okuyacağımız kitabın üstündeki kitapları teker teker yığından çıkarıp sonra okuyacağımız kitabı almalıyız. Öbür türlü kitap destesi devrilebilir. İşte yığınlarda da aynı mantık söz konusudur. Yığınlara koleksiyonlara bu mantığın eklenmiş hâlidir diyebiliriz. Normal koleksiyonlarda istediğimiz her an koleksiyonun istediğimiz elemanına erişebiliyorduk. Ancak yığınlarda bir anda yığının yalnızca bir elemanına erişebiliriz. Daha alttaki elemanlara erişebilmek için üstteki elemanların teker teker yığından çıkarılması gerekir. Yığınların mantığını şu cümleyle özetleyebiliriz: "En önce giren en son çıkar, en son giren en önce çıkar." Yani yığına eklenen her bir elemanın birbirinin üstüne geldiğini söyleyebiliriz. Stack sınıfında tıpkı diğer koleksiyonlardaki gibi eleman eklendikçe otomatik olarak kapasite artar. Bir Stack nesnesi şu yollardan biriyle olışturulabilir.

```
Stack s1=new Stack();
Stack s2=new Stack(int kapasite);
Stack s3=new Stack(ICollection ic);
```

Birincisinde klasik bir Stack nesnesi oluşturulur. İkincisinde kapasitesi kapasite olan bir Stack nesnesi oluşturulur. Tabii ki kapasite aşılırsa otomatik olarak kapasite artırılır. Üçüncüsünde ICollection arayüzünü kullanan başka bir koleksiyon nesnesinden yeni bir Stack nesnesi oluşturulur. Stack sınıfının önemli üye elemanları:
object Pop() İlgili yığındaki en üstteki elemanı döndürür ve yığından çıkarır.
object Peek() İlgili yığındaki en üstteki elemanı döndürür ama yığından çıkarmaz.
void Push(object o) İlgili yığının en üstüne o elemanını ekler.
void Clear() İlgili yığındaki bütün elemanları siler.
object[] ToArray() İlgili yığındaki elemanlar object dizisi olarak döndürülür.
bool Contains(object o) Eğer o nesnesi ilgili yığında varsa true, yoksa false döndürülür.
int Count Bu özellik ilgili yığındaki eleman sayısını verir. Peek() ve Pop() metotları ilgili yığın boşken kullanılırsa istisnai durum oluşur. Bir yığındaki elemanların tamamına bir koleksiyonmuş gibi erişmek için Stack sınıfının GetEnumerator() metodu kullanılır. Bu metot bir IEnumerator nesnesi döndürür. IEnumerator arayüzünün MoveNext() metoduyla bir while döngüsü kurarsak ilgili yığındaki tüm elemanlara erişebiliriz. Örnek:

```
using System;
using System.Collections;
class stack
{
   static void Main()
   {
      Stack s=new Stack();
      s.Push(5);
      s.Push(10);
      EkranaYaz(s);
   }
```

```
    static void EkranaYaz(Stack s)
    {
        IEnumerator ie=s.GetEnumerator();
        while(ie.MoveNext())
            Console.WriteLine(ie.Current);
    }
}
```

Bu programdaki EkranaYaz() metodunu şöyle de yazabilirdik:

```
static void EkranaYaz(Stack s)
{
    Stack yeniyigin=(Stack)s.Clone();
    while(yeniyigin.Count>0)
        Console.WriteLine(yeniyigin.Pop().ToString());
}
```

Buradaki Clone() metodu bir yığının kopyasını object olarak alır. Dolayısıyla bu yeni yığını bir Stack nesnesine atamak için tekrar Stack türüne dönüşüm yapmak gerekir. Yine EkranaYaz() metodunu klasik foreach döngüsüyle de yazabilirdik:

```
static void EkranaYaz(Stack s)
{
    foreach(object o in s)
        Console.WriteLine(o);
}
```

Queue sınıfı

Queue sınıfı Stack sınıfına oldukça benzer. Ancak Stack sınıfının tam tersi bir mantığa sahiptir. Queue sınıfında "En önce giren en erken çıkar, en son giren en geç çıkar." mantığı geçerlidir. Bunu şöyle kavrayabiliriz: Kendimizi bankada bir veznedar olarak düşünelim. Kuyrukta bekleyen birçok kişi var. Tabii ki kuyruktaki en öndeki kişinin işi en erken bitecek ve kuyruktan en önce o ayrılacaktır. Kuyruğun en arkasındaki kişinin işi ise en geç bitecektir. Bizim veznedar olarak en arkadaki kişinin işini halledebilmemiz için o kişinin önündeki kişilerin de işlerini bitirmemiz gerekir. Queue sınıfının önemli üye elemanları:
object Dequeue() İlgili kuyruğun en başındaki elemanı döndürür ve kuyruktan çıkarır.
void Enqueue(object o) o nesnesi ilgili kuyruğun en sonuna eklenir.
object Peek() İlgili kuyruğun en başındaki elemanı döndürür ama kuyruktan çıkarmaz.
int Count İlgili kuyruktaki eleman sayısını verir.
object[] ToArray() İlgili kuyruk bir object dizisi olarak tutulur.
void TrimToSize() İlgili kuyruğun kapasitesiyle eleman sayısı eşitlenir.
Yeni bir Queue nesnesi şöyle oluşturulabilir.

```
Queue q1=new Queue();
Queue q2=new Queue(int kapasite);
Queue q3=new Queue(ICollection ic);
```

Bir Queue nesnesinin içinde Stack nesnelerinin içinde dolaştığımız gibi dolaşabiliriz. Buna örnek verilmeyecektir.

Şablon tipli koleksiyonlar

Şimdiye kadar BitArray hariç bütün koleksiyon sınıflarında koleksiyona eklenen nesne objecte dönüştürülüp saklanır. Sonra bu nesneyi elde etmek istediğimizde ise tekrar orijinal türe dönüşüm yapmak gerekir. Bunun sonucunda hem performans kaybı (tür dönüşümlerinden dolayı) hem de tür emniyetsizliği oluşmaktadır. Tür emniyetsizliğinin nedeni ise ilgili koleksiyonda her türden veri saklayabilmemizdir. Sözlük için kullanacağımız bir koleksiyonun yalnızca string nesneleri kabul etmesi gerekir. Ancak klasik koleksiyonlarda her türden nesne atayabilmemiz bu kontrolü imkansız kılmaktadır. Şablon tipli koleksiyon sınıfları System.Collections.Generic isim alanındadır. Şimdiye kadar gördüğümüz bütün koleksiyonların şablon tipli versiyonları bu isim alanında bulunmaktadır. Şablon tipli koleksiyon sınıflarının karşılığı oldukları normal koleksiyon sınıflarından şablon tipli olmaları dışında fazla farkı yoktur. Burada bu şablon tipli koleksiyon sınıfları kısaca tanıtılacaktır.

List<T> sınıfı

ArrayList sınıfının şablon tipli versiyonudur. Örnek bir List<T> nesnesi yaratımı:

```
List<int> l=new List<int>();
```

Dictionary<T,V> sınıfı

Hashtable sınıfının şablon tipli versiyonudur. Örnek bir Dictionary<T,V> nesnesi yaratımı:

```
Dictionary<int,string> d=new Dictionary<int,string>();
```

SortedDictionary<K,V> sınıfı

SortedList sınıfının şablon tipli versiyonudur. Örnek bir SortedDictionary<K,V> nesnesi yaratımı:

```
SortedDictionary<int,string> sd=new SortedDictionary<int,string>();
```

IEnumerable<T> ve IEnumerator<T> arayüzleri

Klasik versiyonlarından türetilen bu arayüzler şablon tipli koleksiyon sınıflarına arayüzlük yapmaktadır. Kendimiz de şablon tipli koleksiyon sınıfları yazmak istiyorsak bu arayüzleri kullanmalıyız.

Queue<T> ve Stack<T> sınıfları

Queue ve Stack sınıflarını şablon tipli versiyonlarıdır.

Koleksiyonlara kendi nesnelerimizi yerleştirmek

Şimdiye kadar koleksiyonlara string, int gibi .Net Framework kütüphanesindeki hazır sınıfların nesnelerini koyduk. Halbuki kendi oluşturduğumuz sınıfların nesnelerini de koleksiyonlara koyabiliriz. Ancak bu durumda bir sorun ortaya çıkar. Derleyici koleksiyonu sıralarken nasıl sıralayacak? Bu sorunu çözmek için sınıfımıza IComparable arayüzünü uygulattırırız. Bu arayüz önceden de bildiğiniz gibi CompareTo() metodunu içermektedir. Şimdi programımızı yazmaya başlayalım:

```
using System;
using System.Collections;
class Sayi:IComparable
{
   public int sayi;
   public Sayi(int sayi)
   {
```

```
        this.sayi=sayi;
    }
    public int CompareTo(object o)
    {
        Sayi s=(Sayi)o;
        return sayi.CompareTo(s.sayi);
    }
}
class AnaProgram
{
    static void Main()
    {
        ArrayList al=new ArrayList();
        Sayi s1=new Sayi(10);
        Sayi s2=new Sayi(20);
        Sayi s3=new Sayi(-23);
        al.Add(s1);
        al.Add(s2);
        al.Add(s3);
        al.Sort();
        foreach(Sayi s in al)
            Console.WriteLine(s.sayi);
    }
}
```

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

yield

Hatırlarsanız foreach döngüsü şöyle kuruluyordu:

```
foreach(Tür a in b)
{
   ...
}
```

Burada b'nin kendi yarattığımız bir sınıf türünden olması için bu kendi yarattığımız sınıfın IEnumerable arayüzünü uygulaması gerekiyordu. Bu arayüz geri dönüş tipi IEnumerator olan bir metot içeriyordu. Ve daha bir sürü karmaşık şey vardı. Şimdiye kadar IEnumerable ve IEnumerator arayüzlerinin nasıl birlikte çalıştığını anlamamış olabilirsiniz. Ancak aynı mantıkla çalışan çok daha basit bir program yazınca bunun nasıl olduğunu anlayacaksınız:

```
interface i1
{
   i2 Metot();
}
interface i2{}
class Program:i1
{
   public i2 Metot()
   {
      return new AltProgram();
   }
   class AltProgram:i2{}
   static void Main(){}
}
```

Bu programda i1 arayüzü geri dönüş tipi i2 arayüzü olan Metot() isimli metodu içeriyor. Asıl sınıfımız i1 arayüzünü uyguluyor. Geri dönüş tipi i2 arayüzü olan Metot() metodumuzu sınıfımıza yerleştiriyoruz. Programımızdaki en kritik nokta metodumuzun geri dönüş tipi i2 arayüzü olmasına rağmen, bizim AltProgram sınıfı türünden bir nesne döndürmemiz. Çünkü AltProgram sınıfını incelerseniz bunun da i2 arayüzünü kullandığını göreceksiniz. Yani sınıflardaki türeme kuralları arayüz uygulamada da geçerli. Bir sınıf nesnesi, sınıfın kullandığı bir arayüz nesnesine atanabiliyor. Bu kuralı da şu basit programla özetleyebiliriz:

```
interface Arayuz{}
class Sinif:Arayuz{}
class AnaProgram
{
   static void Main()
   {
      Arayuz a;
      Sinif s=new Sinif();
      a=s;
   }
}
```

Ancak bir arayüz nesnesini new operatörüyle oluşturamayız. Çünkü new operatörü nesne için bellekte yer açmak demektir. Arayüzlerin pratikte bellekte yer kaplaması imkansızdır. O yüzden derleyici new operatörüyle arayüz

nesnelerinin oluşturulmasını engeller. Konu buraya gelmişken bir şey söylemek istiyorum. Aslında sadece tanım yapmak istiyorum:

```
Sinif a;
Sinif a=new Sinif();
```

Yukarıdaki iki satırdan birincisinde bir referans oluşturulur. İkincisinde ise nesne oluşturulur. Aslında bu ikisinin arasındaki farkı biliyordunuz, burada sadece bunların isimlerini söyledim. Şimdi konuyu fazla dağıtmadan Arayüzler konusunda gördüğümüz ve kendi oluşturduğumuz sınıf türünden bir nesnenin foreach döngüsüyle kullanılabilmesini sağlayan programı tekrar yazmak istiyorum:

```
using System;
using System.Collections;
class Koleksiyon:IEnumerable
{
   int[] Dizi;
   public Koleksiyon(int[] dizi)
   {
      this.Dizi=dizi;
   }
   IEnumerator IEnumerable.GetEnumerator()
   {
      return new ENumaralandırma(this);
   }
   class ENumaralandırma:IEnumerator
   {
      int indeks;
      Koleksiyon koleksiyon;
      public ENumaralandırma(Koleksiyon koleksiyon)
      {
         this.koleksiyon=koleksiyon;
         indeks=-1;
      }
      public void Reset()
      {
         indeks=-1;
      }
      public bool MoveNext()
      {
         indeks++;
         if(indeks<koleksiyon.Dizi.Length)
            return true;
         else
            return false;
      }
      object IEnumerator.Current
      {
         get
         {
```

```
            return(koleksiyon.Dizi[indeks]);
         }
      }
   }
}
class MainMetodu
{
   static void Main()
   {
      int[] dizi={1,2,3,8,6,9,7};
      Koleksiyon k=new Koleksiyon(dizi);
      foreach(int i in k)
         Console.Write(i+" ");
   }
}
```

Burada az önceki basit örneklerde gördüğümüz kurallar var.

yield

Az kalsın hatırlatma yapacağız derken asıl konumuzu unutuyorduk.:) İşte şimdi asıl konumuza başlıyoruz. yield deyimleri kendi oluşturduğumuz sınıf türünden nesnelerin kısa yoldan foreach döngüsüyle kullanılabilmesini sağlar. Örnek program:

```
using System;
using System.Collections;
class Sinif:IEnumerable
{
   public IEnumerator GetEnumerator()
   {
      yield return 1;
      yield return 2;
      yield return 3;
   }
}
class Program
{
   static void Main()
   {
      Sinif s=new Sinif();
      foreach(int i in s)
         Console.WriteLine(i);
   }
}
```

Bu programın ekran çıktısı alt alta 1, 2 ve 3 olacaktır. Gördüğünüz gibi IEnumerator sınıfını yazmamıza gerek kalmadı. Her yield return sözcüğü bir iterasyon belirtir. Bu örneğimizde yield return anahtar sözcüğünün yanında int türden nesne koyduk. Ancak her türden nesne koyabilirsiniz. yield return anahtar sözcükleri mutlaka geri dönüş tipi IEnumerator olan bir metodun içinde kullanılmalıdır. Başka bir örnek:

```
using System;
using System.Collections.Generic;
using System.Collections;
class Say:IEnumerable<int>
{
   public int KacarKacar;
   public int KactaBitsin;
   IEnumerator<int> IEnumerable<int>.GetEnumerator()
   {
      for(int a=0;a<=KactaBitsin;a+=KacarKacar)
         yield return a;
   }
   IEnumerator IEnumerable.GetEnumerator()
   {
      return null;
   }
}
class AnaProgram
{
   static void Main()
   {
      Say s=new Say();
      s.KacarKacar=10;
      s.KactaBitsin=150;
      foreach(int i in s)
         Console.WriteLine(i);
   }
}
```

Bu program 0'dan 150'ye kadar sayıları 10'ar 10'ar alt alta yazacaktır. yield return anahtar sözcüğüne benzer çalışan bir de yield break anahtar sözcüğü vardır. yield return ile yeni bir iterasyon başlatılırken yield return ile tüm iterasyonlardan çıkılır. Bu anahtar sözcüğü döngülerdeki break anahtar sözcüğüne benzetebiliriz. yield return'ü ise continue'ye benzetebiliriz. Örnek:

```
using System;
using System.Collections;
class Sinif:IEnumerable
{
   public IEnumerator GetEnumerator()
   {
      yield return 1;
      yield return 2;
      yield break;
      yield return 3;
   }
}
class Program
{
```

```
    static void Main()
    {
        Sinif s=new Sinif();
        foreach(int i in s)
            Console.WriteLine(i);
    }
}
```

Bu programla ekrana 1 ve 2 yazılır, 3 yazılmaz.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Veri tabanı işlemleri

C#'ta veri tabanı işlemleri System.Data isim alanındaki ve bu isim alanının altındaki alt isim alanlarındaki türlerle yapılır. System.Data isim alanına programcılar ADO.NET ismini vermişlerdir. Yani bir yerde ADO.NET duyarsanız aslında System.Data isim alanından bahsetmektedir. System.Data isim alanıyla veri tabanlarına iki şekilde erişilebilir.

1. MSSQL veri tabanlarına direkt erişim.
2. OLEDB protokolünü destekleyen veri tabanlarına OLEDB protokolü ile erişim.

Tabii ki bağlanacağınız veri tabanı MSSQL ise daha hızlı olması açısından birinci yöntemi seçmeniz tavsiye edilir. Ancak daha genel bir yol olduğundan bu bölümde ikinci yöntem üzerinde daha çok duracağız. Popüler tüm veri tabanları OLEDB protokolünü desteklemektedir. OLEDB protokolüyle MSSQL veri tabanlarına da erişebilirsiniz. Ayrıca OLEDB protokolüyle Access dosyalarına da bir veri tabanıymış gibi bağlanabilirsiniz.

Şimdi isterseniz System.Data isim alanı ve bu isim alanındaki alt isim alanları hakkında kısa bilgiler verelim:
System.Data Veri tabanlarındaki verilerle çalışmak için gerekli temel türler bu isim alanındadır. Veri tabanlarına bağlanmak için gerekli türler bu isim alanında değildir.
System.Data.Common İleride göreceğiz.
System.Data.OleDb OLEDB protokolünü destekleyen veri tabanlarına bağlanmak için gerekli türler barındırır.
System.Data.SqlClient OLEDB kullanmadan direkt MSSQL veri tabanlarına bağlanmak için gerekli türler barındırır.
System.Data.SqlTypes MSSQL veri tabanlarındaki veri türlerini içerir. Tabii ki veri tabanından veri çekerken veya veri tabanına veri kaydederken C#'a özgü veri türlerini (string, int, ...) kullanabiliriz. Ancak MSSQL'e özgü veri türlerini kullanmamız bize artı performans sağlar.

Veri tabanına bağlanma

Burada çeşitli veri tabanlarına bağlanma hakkında üç örnek verilecektir:

```
 //veri kaynağına erişmek için çeşitli bilgiler hazırlanır.
 string
kaynak="Provider=SqlOleDb;server=SunucuAdı;uid=KullanıcıAdı;pwd=Şifre;database=VeriTabanıAdı";
 //kaynak stringi kullanılarak bağlantı nesnesi oluşturulur.
 OleDbConnection baglanti=new OleDbConnection(kaynak);
 //OleDbConnection sınıfının static olmayan Open() metoduyla bağlantı
aktif hâle getirilir.
 baglanti.Open();
```

Bu örneğimizde OLEDB protokolünü destekleyen bir veri tabanına bağlandık. OleDbConnection sınıfı System.Data.OleDb isim alanındadır.

```
 string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=dosya.mdb";
 OleDbConnection baglanti=new OleDbConnection(kaynak);
 baglanti.Open();
```

Burada ise OLEDB protokolünü kullanarak bir Access dosyasına bir veri tabanıymış gibi bağlandık. Gördüğünüz gibi öncekinden tek farkı kaynak stringi.

```
 string
kaynak="server=SunucuAdı;uid=KullanıcıAdı;pwd=Şifre;database=VeriTabanıAdı";
 SqlConnection baglanti=new SqlConnection(kaynak);
 baglanti.Open();
```

Burada ise OLEDB protokolünü kullanmadan direkt olarak bir MSSQL veri tabanına bağlandık. SqlConnection sınıfı System.Data.SqlClient isim alanındadır. Biz ders boyunca ikinci yöntemi kullanacağız. Yani Access dosyalarına erişeceğiz. OLEDB protokolünü kullanarak normal bir veri tabanına bağlanıp işlem yapma ile aynı şeyi Access dosyalarına yapma arasındaki tek fark bağlanma şeklidir. Yani veri tabanına veya Access dosyasına OLEDB protokolü ile bir kere bağlandıktan sonra veri tabanından veri çekme/veri tabanına veri kaydetme vb. işlemler tamamen aynıdır. OLEDB protokolünü kullanmadan bağlanılan MSSQL veri tabanlarıyla işlemler yapmak kısmen farklı olsa da işlemler büyük ölçüde aynı mantıkla yapılır. Şimdi isterseniz bir veri tabanına bağlanma örneği yapalım:

```
 using System;
 using System.Data.OleDb;
 class vt
 {
    static void Main()
    {
       string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=dosya.mdb";
       OleDbConnection baglanti=new OleDbConnection(kaynak);
       baglanti.Open();
       Console.WriteLine("Bağlantı sağlandı...");
       baglanti.Close();
    }
 }
```

Bu örneğin hata vermeden çalışabilmesi için programımızla aynı klasörde dosya.mdb dosyasının olması gerekir. Access'te bu dosyayı oluşturun. Dosyamız çeşitli alanları olan bir tablo içersin, bu alanlara da veri türleriyle uyumlu olmak şartıyla istediğiniz verileri koyun. OleDbConnection sınıfının static olmayan Close() metodu ilgili bağlantıyı kesip bağlantının kullandığı sistem kaynaklarının serbest bırakılmasını sağlar. Eğer bağlantı sağlanmışsa programın ekrana `Bağlantı sağlandı...` çıktısını vermesi gerekir. Eğer bağlantı sağlanamamışsa istisnai durum oluşması gerekir.

ÖNEMLİ NOT: Bu ders boyunca sizin SQL dilini ve Access programını başlangıç seviyesinde bildiğiniz varsayılmıştır. Örneklerde olabildiğince basit SQL cümleleri kullanılmıştır.

NOT: System.Data isim alanı şimdiye kadar gördüğümüz isim alanlarının aksine mscorlib.dll assemblysinde değil, System.Data.dll assemblysindedir. Bazı .Net Framework sürümlerinde derleyici bu dosyayı otomatik olarak her dosyaya refere etmesine rağmen bazı .Net Framework sürümlerinde refere etmemektedir. Eğer program derleme hatası verirse bir de bu dosyayı refere etmeyi deneyin.

OleDbCommand sınıfı

OleDbCommand sınıfı bir veri tabanına komut göndermek için kullanılır. OleDbCommand sınıfı System.Data isim alanındadır. Bir veri tabanına gönderilmesi için üç şekilde komut oluşturabiliriz.

```
string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=dosya.mdb";
OleDbConnection baglanti=new OleDbConnection(kaynak);
baglanti.Open();
string sorgu="select * from tablo";
OleDbCommand komut=new OleDbCommand(sorgu,baglanti);
```

Bu örneğimizde OleDbCommand sınıfının yapıcı metodu bir OleDbConnection ve bir de string nesnesi aldı. String nesnesi SQL cümlesini içeriyor. Şimdi ikinci yönteme geçelim.

```
string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=dosya.mdb";
OleDbConnection baglanti=new OleDbConnection(kaynak);
baglanti.Open();
OleDbCommand komut=new OleDbCommand("Tablo",baglanti);
komut.CommandType=CommandType.TableDirect;
```

CommandType System.Data isim alanında bulunan bir enumdur. OleDbCommand sınıfının static olmayan CommandType özelliğinin tipi CommandType enumudur. Bu örneğimizde veri tabanına bir SQL cümlesi göndermektense bir tabloyu tamamen programa çekmek istedik.

```
string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=dosya.mdb";
OleDbConnection baglanti=new OleDbConnection(kaynak);
baglanti.Open();
OleDbCommand komut=new OleDbCommand("StorProsedur",baglanti);
komut.CommandType=CommandType.StoredProcedure;
komut.Parameters.Add("@Yas","30");
```

Bu örnekte ise bir stor prosedür çalıştırdık. Parametreyi de Parameters alt sınıfındaki Add() metoduyla verdik.

NOT: Bu üç örneğimizde OleDbConnection nesnesini ve komutu yapıcı metotta verdik. İstersek şöyle de verebilirdik:

```
string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=dosya.mdb";
OleDbConnection baglanti=new OleDbConnection(kaynak);
baglanti.Open();
```

```
string sorgu="select * from tablo";
OleDbCommand komut=new OleDbCommand();
komut.Connection=baglanti;
komut.CommandText=sorgu;
```

NOT: OleDbCommand sınıfının CommandType özelliğine CommandType.Text ilk değeri verilmiştir. Bu da komutun bir SQL cümlesi olduğunu belirtir.
Şimdiye kadar güzel. Ancak halen bazı şeyler eksik. Artık komut elimizde var. Ancak halen komutu çalıştırmadık. Belli bir komutu (OleDbCommand nesnesini) çalıştırmak için OleDbCommand sınıfının 4 farklı static olmayan metodu vardır.
int ExecuteNonQuery() Bu metot veri tabanındaki kayıtlarda değişiklik (ekleme-silme-değiştirme) yapan komutları çalıştırmak için kullanılır. Metot tabloda etkilenen (silinen-eklenen-değiştirilen) kayıt sayısını döndürür. Örnek:

```
string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=dosya.mdb";
OleDbConnection baglanti=new OleDbConnection(kaynak);
baglanti.Open();
string sorgu="INSERT INTO Tablo (Ad,Soyad) VALUES ('Mehmet','Kaplan')";
OleDbCommand komut=new OleDbCommand(sorgu,baglanti);
Console.WriteLine(komut.ExecuteNonQuery()+" tane ekleme yapıldı.");
baglanti.Close();
```

object ExecuteScalar() Bu metot veri tabanından tek bir veri elde eden komutları çalıştırmak için kullanılır. Örneğin SQL dilinin COUNT deyimi tablodaki kayıt sayısını verir. Örneğin `SELECT COUNT(*) FROM Tablo` SQL cümlesini çalıştırmak için bu metot kullanılabilir. Örnek:

```
string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=dosya.mdb";
OleDbConnection baglanti=new OleDbConnection(kaynak);
baglanti.Open();
string sorgu="SELECT COUNT(*) FROM Tablo";
OleDbCommand komut=new OleDbCommand(sorgu,baglanti);
Console.WriteLine((int)komut.ExecuteScalar()+" tane kayıt var.");
baglanti.Close();
```

OleDbDataReader ExecuteReader() Bu metot veri tabanından bir tablo çeken komutları çalıştırmak için kullanılır. Genellikle SELECT deyimleri bu metotla çalıştırılır. Bir OleDbDataReader nesnesi döndürür.

OleDbDataReader sınıfı

OleDbCommand sınıfının static olmayan ExecuteReader() metodu ile oluşan tabloları tutmaya yarayan bir sınıftır. OleDbDataReader nesneleri new operatörüyle oluşturulamaz. OleDbDataReader nesneleri oluşturmak için OleDbCommand sınıfının static olmayan ExecuteReader() metodunu kullanabiliriz. OleDbDataReader sınıfı System.Data.OleDb isim alanındadır. Şimdi isterseniz Access'te şöyle bir tablo oluşturalım:

id	ad	soyad	not
1	ali	yılmaz	70
2	mehmet	süzen	90
3	zafer	kaplan	100
4	mehmet	oflaz	45
5	ayşe	yılmaz	34
6	hatice	özdoğan	100
7	emine	şanlı	20

Bu tabloya ogrenci ismini verelim. Dosyayı kaynak kodumuzla aynı klasörde oluşturup adını dosya.mdb koyalım. Şimdi aşağıdaki programı yazalım.

```
using System;
using System.Data;
using System.Data.OleDb;
class vt
{
   static void Main()
   {
      string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=dosya.mdb";
      OleDbConnection baglanti=new OleDbConnection(kaynak);
      baglanti.Open();
      string sorgu="SELECT * FROM ogrenci";
      OleDbCommand komut=new OleDbCommand(sorgu,baglanti);
      OleDbDataReader cikti=komut.ExecuteReader();
      string format="{0,-10}{1,-10}{2,-10}";
      Console.WriteLine(format,"Adı","Soyadı","Notu");
      Console.WriteLine("".PadRight(30,'-'));
      while(cikti.Read())
      {
         Console.WriteLine(format,cikti[1],cikti[2],cikti[3]);
      }
      cikti.Close();
      baglanti.Close();
   }
}
```

Bu program ogrenci tablosundaki id hariç tüm verileri bir tablo şeklinde konsol ekranına yazar. Bu programda Temel string işlemleri konusunda gördüğümüz yazı formatlamayı kullandık. Aynı konuda gördüğümüz String sınıfına ait PadRight() metodunu ise ekrana bir karakteri belirli sayıda yazdırmak amacıyla kullandık. Asıl konumuza dönecek olursak OleDbDataReader sınıfının static olmayan Read() metodu ilgili tablodaki aktif kayıtı bir birim öteler ve true döndürür. Eğer aktif kayıt ötelenemiyorsa false döndürür. OleDbDataReader sınıfında tanımlı indeksleyici sayesinde aktif kayıttaki farklı alanlar tutulabilir. Bu indesleyici object türünde nesneler döndürür. Son olarak OleDbConnection ve OleDbDataReader nesneleriyle işimiz bittiğinde ilgili nesnenin Close() metoduyla sistemin ilgili tablo/bağlantı için ayırdığı sistem kaynaklarını işlemciye iade etmemiz gerekiyor. İstersek OleDbDataReader sınıfına ait indeksleyiciyi şöyle de kullanabilirdik:

```
Console.WriteLine(format,cikti["ad"],cikti["soyad"],cikti["not"]);
```

Gördüğünüz gibi indeksleyiciye bir indeks numarası vermektense direkt olarak sütunun adını da verebiliyoruz. Aynı programdaki while döngüsünü şöyle de kurabilirdik.

```
while(cikti.Read())
{

Console.WriteLine(format,cikti.GetString(1),cikti.GetString(2),cikti.GetInt32(3));
}
```

Burada klasik indeksleme yönteminin aksine türler belirtilmiştir. İndeksleyici geriye object nesnesi döndürüyordu. Eğer veri tabanındaki sütunların tipini biliyorsak bu şekilde bir kullanım tür dönüşümü olmadığından dolayı işlemler daha hızlı gerçekleşeceği için tavsiye edilir. Microsoft Office Access 2003 sürümündeki veri tipleri, C# karşılıkları ve bunları programa geçirmek için gerekli OleDbDataReader metotları şöyledir:

Access	C#	İlgili metot
Tamsayı	short	GetInt16()
Uzun Tamsayı	int	GetInt32()
Bayt	byte	GetByte()
Çift	double	GetDouble()
Tek	float	GetFloat()
Ondalık	decimal	GetDecimal()
Metin	string	GetString()
Evet/Hayır	bool	GetBoolean()
Tarih/Saat	DateTime	GetDateTime()

OleDbDataAdapter, DataSet, DataTable, DataRow ve DataColumn sınıfları

Artık şimdiye kadar öğrendiğimiz sınıflarla bir veri tabanından veri çekip veri tabanına veri kaydedebiliyoruz. Zaten bir veri tabanıyla yapabileceğimiz işlemler bu kadar. Ancak birazdan göreceğimiz OleDbDataAdapter, DataSet, DataTable, DataRow ve DataColumn sınıfları bu işlemleri yaparken bize daha fazla seçenek sunuyor. Ayrıca bu sınıflar bize offline çalışmanın kapılarını açıyor. Yani öncelikle veri tabanından bir veri çekiyoruz, bu veriyi kendi makinamıza aktarıyoruz, bu işlemden sonra veri tabanıyla bağlantımızın kesilmesi programımızın işleyişine engel değil. Sonra çektiğimiz veri üzerinde istediğimiz oynamaları yapıyoruz. Sonra değiştirilmiş verileri tekrar veri tabanına yazıyoruz. Bu şekilde veri tabanıyla işlemlerimiz, veri tabanının bulunduğu bilgisayarla programın bulunduğu bilgisayar arasında fazla git-gel olmadığı için daha hızlı gerçekleşiyor. Şimdi offline çalışmayla ilgili sınıfları önce kısaca inceleyelim:

OleDbDataAdapter OLEDB protokolünü destekleyen veri tabanlarından veri çekmek ve değiştirilmiş verileri aynı veri tabanına tekrar yazmak için kullanılır. Offline çalışmayla ilgili sınıflar içinde veri tabanıyla fiziksel olarak iletişimde olan tek sınıftır. OleDbDataAdapter sınıfının çektiği veri tek bir veri olabileceği gibi bir ya da daha fazla tablo da olabilir. Hatta istersek OleDbDataAdapter sınıfıyla bir veri tabanının tamamını da programımıza aktarabiliriz. OleDbDataAdapter sınıfı System.Data.OleDb isim alanındadır. System.Data.SqlClient isim alanındaki SqlDataAdapter sınıfı ise OLEDB'siz bağlanılan MSSQL veri tabanları için aynı şeyi yapar. OleDbDataAdapter ile SqlDataAdapter sınıflarının arayüzleri aynıdır. Dolayısıyla birazdan OleDbDataAdapter sınıfını incelerken aynı zamanda SqlDataAdapter sınıfını da incelemiş olacağız.

DataSet OleDbDataAdapter sınıfının veri tabanından çektiği verileri programda offline olarak tutmaya yarar.

System.Data isim alanındadır.
DataTable Bir DataSet'teki bir tabloyu temsil eder. System.Data isim alanındadır.
DataRow Bir tablodaki tek bir satırı (kaydı) temsil eder. System.Data isim alanındadır.
DataColumn Bir tablodaki tek bir sütunu temsil eder. System.Data isim alanındadır.

Veri tabanından veri çekmek

Öncelikle bir OleDbDataAdapter nesnesi oluşturmalıyız:

```
string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=dosya.mdb";
OleDbConnection baglanti=new OleDbConnection(kaynak);
baglanti.Open();
string sorgu="SELECT * FROM TabloAdı";
OleDbDataAdapter odda=new OleDbDataAdapter(sorgu,baglanti);
```

Sonra OleDbDataAdapter sınıfının Fill() metoduyla ilgili OleDbDataAdapter nesnesinin tuttuğu verileri bir DataSet nesnesine aktarmalıyız.

```
DataSet ds=new DataSet();
odda.Fill(ds); //Bu satırla odda nesnesindeki veriler ds nesnesine
atandı.
```

İstersek OleDbDataAdapter nesnesinden DataSet nesnesine bir tablo aktarırken tablonun adını belirleyebiliriz.

```
DataSet ds=new DataSet();
odda.Fill(ds,"TabloAdı");
```

DataSet sınıfının Tables özelliği ilgili DataSet nesnesindeki tabloları bir DataTable koleksiyonu olarak döndürür.

```
DataSet ds=new DataSet();
odda.Fill(ds,"TabloAdı");
DataTable dt=ds.Tables["TabloAdı"];
//veya
DataTable dt2=ds.Tables[0];
//veya
DataTableCollection dtc=ds.Tables; //DataTableCollection bir
koleksiyon sınıfıdır ve System.Data isim alanındadır.
```

DataTable sınıfının Columns özelliği ilgili DataTable nesnesindeki sütunları bir DataColumn koleksiyonu olarak döndürür.

```
DataTable dt=ds.Tables["TabloAdı"];
DataColumn dc=dt.Columns["SütunAdı"];
//veya
DataColumn dc2=dt.Columns[2];
//veya
DataColumnCollection dcc=dt.Columns; //DataColumnCollection koleksiyon
sınıfı System.Data isim alanındadır.
```

DataTable sınıfının Rows özelliği ilgili DataTable nesnesindeki satırları bir DataRow koleksiyonu olarak döndürür.

```
DataTable dt=ds.Tables["TabloAdı"];
DataRow dr=dt.Rows[1];
//veya
```

```
 DataRowCollection drc=dt.Rows; //DataRowCollection koleksiyon sınıfı
System.Data isim alanındadır.
```

DataRow sınıfının ItemArray özelliği ilgili satırdaki verileri bir object dizisi olarak tutar. Yani

```
 object[] o=dr.ItemArray;
```

Ayrıca bir DataRow nesnesindeki farklı sütunlara indeksleyici ile de erişilebilir. Örnek

```
 object o1=dr[0]; //veya
 object o2=dr["SütunAdı"];
```

Yine bu indeksleyiciler de object nesneleri döndürür.
NOT: DataTableCollection, DataRowCollection ve DataColumnCollection sınıflarının Count özellikleri bir datasetteki tablo sayısının ve bir tablodaki satır ve sütun sayısının bulunması amacıyla kullanılabilir.

Bir sütunun özelliklerini değiştirmek ve edinmek

DataColumn sınıfının çeşitli static olmayan özellikleri vardır:
bool AllowDBNull Sütunun boş değer kabul edip etmeyeceği belirtilir.
bool AutoIncrement Eklenen yeni kayıtlarda ilgili sütundaki verinin otomatik artıp artmayacağı belirtilir.
long AutoIncrementSeed Otomatik artacak değerin başlangıç değeri
long AutoIncrementStep Otomatik artımın kaçar kaçar olacağı belirtilir.
string Caption Sütunun ismi.
Type DataType Sütunun veri tipi.
object DefaultValue Sütundaki her hücrenin varsayılan değeri. (boş bırakıldığında)
int MaxLength Sütundaki her hücredeki verinin maksimum karakter sayısı.
int Ordinal Sütunun tabloda kaçıncı sırada olduğunu verir. (salt okunur)
bool Unique Sütunda bir verinin tekrarlanıp tekrarlanamayacağı. Tekrarlanabiliyorsa false, tekrarlanamıyorsa true.
DataTable Table Sütunun hangi tabloya ait olduğu (salt okunur)

Veri tabanını güncellemek

Bunu bir örnek üzerinde anlatmayı uygun buluyorum. Öncelikle kaynak kodumuzla aynı klasörde dosya.mdb isimli bir Access dosyası oluşturun. Bu dosyada tablo isimli bir tablo oluşturun. Tablo şöyle olsun:

id	ad	soyad
1	bekir	oflaz
2	mehmet	kaplan

id alanı Sayı (Uzun Tamsayı) diğer alanlar ise Metin tipinde olsun. Tabloda birincil anahtar olmasın. Şimdi programımızı yazmaya başlayabiliriz.

```
 using System.Data.OleDb;
 using System.Data;
 class vt
 {
    static void Main()
    {
       string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=dosya.mdb";
       OleDbConnection baglanti=new OleDbConnection(kaynak);
```

```
        string sorgu="SELECT * FROM tablo";
        OleDbDataAdapter odda=new OleDbDataAdapter(sorgu,baglanti);
        DataSet ds=new DataSet();
        odda.Fill(ds,"tablo");
        DataRow r=ds.Tables["tablo"].NewRow();
        r["id"]=21;
        r["ad"]="hatice";
        r["soyad"]="özdoğan";
        odda.InsertCommand=new OleDbCommand("INSERT INTO tablo
(id,ad,soyad) values (?,?,?)",baglanti);
        odda.UpdateCommand=new OleDbCommand("UPDATE tablo SET id=?,
ad=?, soyad=? WHERE id=?",baglanti);
        odda.DeleteCommand=new OleDbCommand("DELETE FROM tablo WHERE
id=?",baglanti);

odda.InsertCommand.Parameters.Add("@id",OleDbType.Integer,2,"id");

odda.InsertCommand.Parameters.Add("@ad",OleDbType.VarChar,10,"ad");

odda.InsertCommand.Parameters.Add("@soyad",OleDbType.VarChar,10,"soyad");

odda.UpdateCommand.Parameters.Add("@id",OleDbType.Integer,2,"id");

odda.UpdateCommand.Parameters.Add("@ad",OleDbType.VarChar,10,"ad");

odda.UpdateCommand.Parameters.Add("@soyad",OleDbType.VarChar,10,"soyad");

odda.UpdateCommand.Parameters.Add("@oldid",OleDbType.Integer,2,"id").SourceVersion=DataRowVersion.Original;

odda.DeleteCommand.Parameters.Add("@id",OleDbType.Integer,2,"id").SourceVersion=DataRowVersion.Original;
        ds.Tables["tablo"].Rows.Add(r);
        ds.Tables["tablo"].Rows[0]["id"]=98;
        ds.Tables["tablo"].Rows[1].Delete();
        odda.Update(ds,"tablo");
    }
 }
```

Bu program veri tabanında ekleme, değiştirme ve silme yaptı. Access dosyasındaki tablomuzun yeni hâlinin şöyle olması gerekiyor:

id	ad	soyad
98	bekir	oflaz
21	hatice	özdoğan

Programdaki önemli satırları teker teker inceleyecek olursak;

```
 DataRow r=ds.Tables["Uyeler"].NewRow();
```

Burada ds DataSet nesnesindeki tablo tablosundaki düzene uygun bir DataRow nesnesi oluşturduk.

```
 odda.InsertCommand=new OleDbCommand("INSERT INTO tablo (id,ad,soyad)
values (?,?,?)",baglanti);
 odda.UpdateCommand=new OleDbCommand("UPDATE tablo SET id=?, ad=?,
soyad=? WHERE id=?",baglanti);
 odda.DeleteCommand=new OleDbCommand("DELETE FROM tablo WHERE
id=?",baglanti);
```

Bu üç satırda ise OleDbDataAdapter sınıfının ilk değer atanmamış ve tipi OleDbCommand olan üç özelliğine değer atadık. SQL cümlelerindeki ? işareti oraya bir parametre geleceğini belirtiyor. Burada OleDbDataAdapter sınıfının Update() metodu kullanıldığında aslında hangi komutların çalıştırılacağını belirledik. Bu satırlardan bazılarını yazmayabilirdik. Bu durumda yalnızca değer atanan özelliklerin komutları çalışırdı.

```
 odda.InsertCommand.Parameters.Add("@id",OleDbType.Integer,2,"id");
 odda.InsertCommand.Parameters.Add("@ad",OleDbType.VarChar,10,"ad");

odda.InsertCommand.Parameters.Add("@soyad",OleDbType.VarChar,10,"soyad");
 odda.UpdateCommand.Parameters.Add("@id",OleDbType.Integer,2,"id");
 odda.UpdateCommand.Parameters.Add("@ad",OleDbType.VarChar,10,"ad");

odda.UpdateCommand.Parameters.Add("@soyad",OleDbType.VarChar,10,"soyad");

odda.UpdateCommand.Parameters.Add("@oldid",OleDbType.Integer,2,"id").SourceVersion=DataRowVersion.Original;

odda.DeleteCommand.Parameters.Add("@id",OleDbType.Integer,2,"id").SourceVersion=DataRowVersion.Original;
```

Burada ? işaretleri yerine ne geleceğini belirledik. Bunların eklenme sırası önemlidir. İlk ? yerine ilk eklenen gelir. Veri tiplerini System.Data.OleDb isim alanındaki OleDbType enumuyla belirledik. DataRowVersion enumu ise System.Data isim alanındadır.

```
 ds.Tables["tablo"].Rows.Add(r);
 ds.Tables["tablo"].Rows[0]["id"]=98;
 ds.Tables["tablo"].Rows[1].Delete();
```

Bu üç satırda ise sırasıyla önceki oluşturduğumuz r satırını ds datasetindeki tablo tablosuna ekledik. Sonra 0. kaydın id sütunundaki değeri 98 yaptık. Sonra 1. kaydı sildik.

```
 odda.Update(ds,"tablo");
```

Son olarak yaptığımız offline değişikliklerin veri tabanında da yapılmasını sağladık. Offline çalışmada bağlantının Open() ve Close() metotlarıyla açılıp kapanmasına gerek yoktur.

Veri sağlayıcı bağımsız erişim

Veri tabanlarına erişim veri sağlayıcılarla olur. Örneğin OLEDB bir veri sağlayıcıdır. OLEDB ile hemen hemen bütün veri tabanlarına bağlanabiliriz. Ancak MSSQL Server veya Oracle Server'ın kendine has veri sağlayıcıları da vardır. Bunlar veri tabanına göre optimize edildiği için veri tabanına erişimde ciddi bir hız kazancı sağlarlar. .Net Framework kütüphanesinde OLEDB protokolü için sınıflar yer almakla birlikte bu veri tabanlarına erişim için özelleşmiş sınıflar da mevcuttur. Şimdi karşımıza iki seçenek çıkıyor. Ya OLEDB ile hızdan ödün verip veri tabanı bağımsız erişim sağlayacağız ya da her veri tabanı için özelleşmiş sınıfları kullanacağız. Verilerimizin tutulduğu veri tabanı değiştiğinde bizim de programımızın kaynak kodunu tekrar değiştirip tekrar derleyip tekrar dağıtmamız gerekecek. Peki hem hızdan ödün vermeden hem de veri sağlayıcı bağımsız erişim mümkün olabilir mi? Evet, mümkün olabilir. Bunu C# geliştiricileri fabrika sınıf modeliyle başarmışlardır. .Net kütüphanesindeki fabrika sınıfları hakkında bilgiyi aşağıdaki programı yazarak bulabilirsiniz:

```
using System;
using System.Data;
using System.Data.Common;
class c
{
   static void Main()
   {
      DataTable saglayicilar=DbProviderFactories.GetFactoryClasses();
      foreach(DataRow satir in saglayicilar.Rows)
      {
         for(int i=0;i<saglayicilar.Columns.Count-1;i++)
            Console.WriteLine(satir[i].ToString());
         Console.WriteLine("".PadRight(15,'-'));
      }
   }
}
```

DbProviderFactories sınıfı System.Data.Common isim alanındadır. DbProviderFactories sınıfının static GetFactoryClasses() metodu, içinde .Net Framework kütüphanesindeki fabrika sınıflar hakkında bilgi olan bir tablo döndürür. Tablodaki her satır farklı bir fabrika sınıf içindir. Tablodaki her sütun ise ilgili fabrika sınıf hakkında farklı kategorideki bilgiler içindir. Tablodaki son sütun bizim için biraz fazla karmaşık bilgiler içerdiği için bu sütunu ekrana yazma gereksinimi görmedim. Bu programın bendeki çıktısı şöyle oldu:

```
Odbc Data Provider
.Net Framework Data Provider for Odbc
System.Data.Odbc
---------------
OleDb Data Provider
.Net Framework Data Provider for OleDb
System.Data.OleDb
---------------
OracleClient Data Provider
.Net Framework Data Provider for Oracle
System.Data.OracleClient
---------------
SqlClient Data Provider
```

```
.Net Framework Data Provider for SqlServer
System.Data.SqlClient
---------------
```

Bu ekran sizde farklı olabilir. Herhangi bir fabrika nesnesi oluşturmak için:

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

Yani bir fabrika oluşturmak için az önceki tabloda karşımıza çıkan üçüncü sütundaki yazıyı kullanıyoruz. DbProviderFactory sınıfı System.Data.Common isim alanındadır. Konunun geri kalan kısmını bir örnek üzerinde anlatmayı uygun buluyorum:

```
 using System.Data.Common;
 using System;
 class Fabrikalar
 {
    static void Main()
    {
       DbProviderFactory
fabrika=DbProviderFactories.GetFactory("System.Data.SqlClient");
       DbConnection baglanti=fabrika.CreateConnection();

baglanti.ConnectionString="server=Sunucu;uid=Kullanici;pwd=sifre;database=VeriTabani";
       baglanti.Open();
       DbCommand komut=fabrika.CreateCommand();
       komut.Connection=baglanti;
       komut.CommandText="SELECT * FROM TABLO";
       DbDataReader okuyucu=komut.ExecuteReader();
       while(okuyucu.Read())
          Console.WriteLine(okuyucu[1].ToString()+"
"+okuyucu[2].ToString());
    }
 }
```

Buradaki DbConnection, DbCommand ve DbDataReader sınıfları yine System.Data.Common isim alanındadır. Şimdiye kadar gördüğümüz veri tabanıyla ilgili sınıfların çoğunun System.Data.Common isim alanında erişim sağlayıcı bağımsız versiyonları vardır. Yine bu sınıfların arayüzleri aynıdır. Burada yaptığımız aslında veri tabanına özgü sınıflar kullanmak yerine hangi veri tabanına bağlanılacağını bir stringle belirlemekti. Böylelikle öncelikle kodlarla veri tabanından veri tabanının hangi veri tabanı olduğu bilgisi alınır. Sonra GetFactory() metodunun parametresine uygun string alınır. İşte bu sayede de veri sağlayıcıdan bağımsız işlemler yapılmış olur. Çünkü hangi veri tabanına bağlanılacağı derleme zamanında değil, programın karşılaşağı durumlara göre çalışma zamanında belirlenir.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Form tabanlı uygulamalar

C#'ta görsel formlarla ilgili işlemler System.Windows.Forms isim alanındaki sınıflarla yapılır. Bu isim alanı System.Windows.Forms.dll assemblysindedir.

Form sınıfı

Form sınıfı programımıza bir form getirmeye yarar. Örnek program:

```
using System.Windows.Forms;
class FormOrnek
{
   static void Main()
   {
      Form f=new Form();
      f.Show();
   }
}
```

Bu programda önce bir komut satırı açılır, sonra bir form açılır, sonra form kapanır. Eğer sisteminiz çok hızlıysa formu görmemeniz bile mümkündür. Ancak program çalışacaktır. Şimdi bir de bu programı

```
csc /t:winexe form.cs
```

komutuyla derleyin. Bu durumda ilgili exe dosyasını çift tıklayıp açtığımızda artık komut satırı penceresi gözükmeyecektir. Yani `/t:winexe` argümanının tek fonksiyonu budur. Şimdi program kodumuzu şöyle değiştirelim:

```
using System.Windows.Forms;
class FormOrnek
{
   static void Main()
   {
      Form f=new Form();
      Application.Run(f);
   }
}
```

Bu sefer Form sınıfının Show() metodunu değil, Application sınıfının static Run() metodunu kullandık ve metodun parametresine Form nesnemizi verdik. Bu sefer form biz sağ üst köşedeki kapatma düğmesine basana kadar ekranda kalacaktır. Şimdi programızı şöyle değiştirelim:

```
using System.Windows.Forms;
class FormOrnek:Form
{
   static void Main()
   {
      Application.Run(new FormOrnek());
   }
}
```

Bu programda öncekilerin aksine Form sınıfına bağlanmadık. Kendi sınıfımızı Form sınıfından türettik. Bu sayede Form sınıfının birçok static üye elemanına yalnızca üye elemanın adını yazarak erişebiliriz. Ancak burada sınıfımızı Form sınıfından türetmedeki asıl amaç programımızın aslında bir form olmasıdır. Yani mantıksal bütünlüğü korumak için sınıfımızı Form sınıfından türettik.

NOT: `Sinif a=new Sinif()` satırında nesneyi oluşturan asıl kısım `new Sinif()` kısmıdır. Diğer kısım yalnızca oluşturulan bu nesnenin a referansına atanmasını sağlar.

Button sınıfı

Daha önce formun mantıksal olarak programımızın kendisini temsil ettiğini söylemiştik. Butonlar ve diğer kontroller ise yalnızca programımızdaki bileşenleri temsil ederler. Şimdi Button sınıfıyla ilgili bir örnek yapalım:

```
using System.Windows.Forms;
using System.Drawing;
class Form1:Form
{
   private Button buton;
   public Form1()
   {
      buton=new Button();
      buton.Text="Butonun üzerindeki yazı";
      buton.Location=new Point(50,50);
      buton.Cursor=Cursors.No;
      buton.Size=new Size(150,50);
      this.Text="Formun başlığı";
      this.Controls.Add(buton);
   }
   static void Main()
   {
      Application.Run(new Form1());
   }
}
```

Size ve Point System.Drawing isim alanında bulunan iki sınıftır ve bir kontrolün konum ve boyut bilgisini tutarlar. Button sınıfının Location özelliği Point tipinden, Size özelliği ise Size tipindendir. Button sınıfının Text özelliği butonun üzerindeki yazıyı, Cursor özelliği fare ilgili butonun üzerine geldiğinde alacağı şekli belirtmemizi sağlar. Cursors enumu System.Windows.Forms isim alanındadır. Daha önceden de gördüğümüz gibi this anahtar sözcüğü

bir yere hangi nesne üzerinden erişilmişse o nesneyi tutar. Form1 sınıfının yapıcı metoduna `new Form1()` yazarak eriştiğimize göre bu yere yeni bir Form1 nesnesi üzerinden erişilmiştir. Form sınıfının Text özelliği formun başlık çubuğundaki başlığını belirtir. Form sınıfındaki Controls özelliğinin tipi yalnızca kontrolleri tutabilen bir koleksiyondur. Bu koleksiyona nesne eklersek aynı zamanda forma da kontrol eklemiş oluruz.

NOT: Nesne yönelimli programlamanın en temel yaklaşımı bir problemi olabildiğince çok parçaya bölüp her bir parçanın yalnızca kendisiyle ilgili kodlar içermesidir. Bu sayede kaynak kodun karmaşıklığı azalacak ve kodun yönetimi kolaylaşacaktır. Biz bu örneğimizde bütün kodları Main metodunun içine koyabilirdik. Ancak nesne yönelim tekniğinin temel yaklaşımına aykırı olurdu. Üstelik bu sayede kodumuzda mantıksal bir bütünlük de yakaladık. Örneğin buton, formumuzda bulunan bir bileşendir. Bunu Main ya da yapıcı metoda koymak yerine Form1 sınıfının bir özelliği olarak yerleştirdik. Benzer şekilde yapıcı metoda yalnızca formla ilgili kodlar yerleştirdik. Programımızın temel kısmı olan Main() metodumuz ise yalnızca ilgili metotları çağırmaktan başka işe yaramadı. Gelişmiş uygulamalarda Main() metodu olabildiğince az kod içerir. Main() metodu yalnızca diğer metotların koordinasyonuyla ilgilenir.

Olaylar ve formlar

Hatırlarsanız Temsilciler ve Olaylar konusunda olayların Windows formlarıyla içli dışlı olduğunu söylemiştik. Olayların en yaygın kullanım yeri Windows formlarıdır. Windows formlarıyla ilgili sınıfların çoğunun onlarca olayı vardır. Bu olayların tipi EventHandler temsilcisidir. Bu temsilci şu şekildedir.

```
void EvenHandler(object kaynak, EventArgs e)
```

Yani bu temsilci geriye bir değer döndürmeyen, iki parametre alan ve parametreleri de sırasıyla object ve EventArgs türünden olan metotları temsil eder. Dolayısıyla da Windows formlarıyla ilgili sınıfların olaylarına da yalnızca bu prototipte metotlar bağlanabilir. Hatırlarsanız olaylar konusunda hem temsilciyi (olay töneticisi), hem olayı, hem olaya bağlanan metodu biz yazmış, hatta olayı da biz gerçekleştirmiştik. Halbuki Windows formlarında biz yalnızca olaya bağlanan metotla ilgileniriz. Windows formlarında olay yöneticisi yukarıda prototipi yazılan EventHandler temsilcisidir. Yine bütün formlarla ilgili olaylar ilgili sınıfta bildirilmiştir. Olayın gerçekleştirilmesini ise kullanıcı yapar. İşletim sistemi herhangi bir olay gerçekleştiğinde (örneğin butonun tıklanması) olayın gerçekleştiği bilgisini ve bu olayla ilgili ek bilgileri programımıza verir. Eğer programımızda bu olayı işleyecek herhangi bir metot yoksa hiçbir şey yapılmaz. Varsa o metottaki komutlar çalıştırılır. Şimdi olayları hatırlama açısından bir örnek yapalım:

```
using System;
delegate void OlayYoneticisi(); //Olay yöneticisi bildirimi
class AnaProgram
{
   static void Main()
   {
      AnaProgram nesne=new AnaProgram();
      nesne.Olay+=new OlayYoneticisi(Metot); //Olay sonrası
işletilecek metotların eklenmesi
      nesne.Olay(); //Olayın gerçekleştirilmesi
   }
   //Olay sonrası işletilecek metot
   static void Metot()
   {
      Console.WriteLine("Butona tıklandı.");
   }
   event OlayYoneticisi Olay; //Olay bildirimi
```

```
}
```

Bu en basit olay mantığıydı. Şimdi bunu Windows formlarına uyarlayalım:

```
delegate void EventHandler(object o,EventArgs e);
class Button
{
   event EventHandler Click;
}
class KendiSinifimiz
{
   static void Main()
   {
      Button buton1=new Button();
      buton1.Click+=new EventHandler(Metot);
   }
   void Metot(object o,EventArgs e)
   {
      //Tıklandığında çalışacak
   }
}
class IsletimSistemi
{
   static void KullaniciTikladiginda()
   {
      buton1.Click;
   }
}
```

Kuşkusuz bu program çalışmayacaktır. Yalnızca Windows formlarındaki olayların nasıl işlediğini anlamanız için bu örneği verdim. IsletimSistemi sınıfının arka planda sürekli çalıştığını düşünebilirsiniz. Şimdi artık olaylarla ilgili gerçek örneğimize başlayabiliriz:

```
using System;
using System.Windows.Forms;
class Form1:Form
{
   private Button buton;
   public Form1()
   {
      buton=new Button();
      buton.Text="Tıkla";
      this.Controls.Add(buton);
      buton.Click+=new EventHandler(Tiklandiginda);
   }
   static void Main()
   {
      Application.Run(new Form1());
```

```
    }
    void Tiklandiginda(object o,EventArgs e)
    {
        MessageBox.Show("Butona tıkladınız");
    }
}
```

EventArgs System alanında ve MessageBox System.Windows.Forms isim alanında birer sınıflardır. MessageBox sınıfının static Show() metodu ekrana bir ileti kutusunun gelmesini sağlar. EventArgs ise olayla ilgili ek bilgileri tutan sınıftır. Diğer form kontrollerini bir sonraki konumuz olan Visual Studio.NET dersinde göreceğiz. Yani diğer kontrolleri formumuza kodlarla değil, Visual Studio.NET ile getireceğiz.

Bu kitabın diğer sayfaları

• C# hakkında temel bilgiler • İlk programımız • Değişkenler • Tür dönüşümü • Yorum ekleme • Operatörler • Akış kontrol mekanizmaları • Rastgele sayı üretme • Diziler • Metotlar	• Sınıflar • Operatör aşırı yükleme • İndeksleyiciler • Yapılar • Enum sabitleri • İsim alanları • System isim alanı • Temel I/O işlemleri • Temel string işlemleri • Kalıtım	• Arayüzler • Partial (kısmi) tipler • İstisnai durum yakalama mekanizması • Temsilciler • Olaylar • Önişlemci komutları • Göstericiler • Assembly kavramı • Yansıma	• Nitelikler • Örnekler • Şablon tipler • Koleksiyonlar • yield • Veri tabanı işlemleri • XML işlemleri • Form tabanlı uygulamalar • Visual Studio.NET • Çok kanallı uygulamalar	• Linux'ta C# kullanımı • Kaynakça • Giriş sayfası • **Ayrıca Bakınız:** ASP.NET

Madde Kaynakları ve Katkıda Bulunanlar

C Sharp hakkında temel bilgiler *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=34103 *Katkıda bulunanlar*: Anov, Bekiroflaz, 5 anonim düzenlemeler

İlk programımız *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=34446 *Katkıda bulunanlar*: Anov, Bekiroflaz, Cankaya72, Emreayan, Joseph, Srhat, Srkn43, 8 anonim düzenlemeler

Değişkenler *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=35202 *Katkıda bulunanlar*: Anov, Bekiroflaz, Joseph, Srhat, Srkn43, ŞizofrenDana, 20 anonim düzenlemeler

Tür dönüşümü *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=34597 *Katkıda bulunanlar*: Anov, Bekiroflaz, Zynp arkbga, ŞizofrenDana, 9 anonim düzenlemeler

Yorum ekleme *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=35203 *Katkıda bulunanlar*: Anov, Bekiroflaz, Jasper Deng, SLV100, Srhat, 6 anonim düzenlemeler

Operatörler *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=34596 *Katkıda bulunanlar*: Anov, Bekiroflaz, Srhat, Zynp arkbga, 7 anonim düzenlemeler

Akış kontrol mekanizmaları *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=35119 *Katkıda bulunanlar*: Anov, Bekiroflaz, Srhat, Taskinv, 14 anonim düzenlemeler

Rastgele sayı üretme *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=28434 *Katkıda bulunanlar*: Anov, Bekiroflaz, Srhat

Diziler *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=35142 *Katkıda bulunanlar*: Anov, Bekiroflaz, Joseph, Srhat, 12 anonim düzenlemeler

Metotlar *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=34868 *Katkıda bulunanlar*: Anov, Bekiroflaz, Srhat, 10 anonim düzenlemeler

Sınıflar *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=34598 *Katkıda bulunanlar*: Anov, Bayramakgul, Bekiroflaz, Srhat, Zynp arkbga, 11 anonim düzenlemeler

Operatör aşırı yükleme *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32107 *Katkıda bulunanlar*: Anov, Bekiroflaz, Srhat, 1 anonim düzenlemeler

İndeksleyiciler *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32108 *Katkıda bulunanlar*: Anov, Bekiroflaz, Srhat

Yapılar *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=30479 *Katkıda bulunanlar*: Anov, Bekiroflaz, Hayalet kny, Srhat

Enum sabitleri *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32109 *Katkıda bulunanlar*: Anov, Bekiroflaz, Hayalet kny, 1 anonim düzenlemeler

İsim alanları *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=33716 *Katkıda bulunanlar*: Anov, Bekiroflaz, 2 anonim düzenlemeler

System isim alanı *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=33709 *Katkıda bulunanlar*: Anov, Bekiroflaz, Tamer229, Vito Genovese, 5 anonim düzenlemeler

O işlemleri *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=35236 *Katkıda bulunanlar*: Anov, Bekiroflaz, Yabancı, 9 anonim düzenlemeler

Temel string işlemleri *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=34250 *Katkıda bulunanlar*: Anov, Bekiroflaz, 10 anonim düzenlemeler

Kalıtım *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=35466 *Katkıda bulunanlar*: Anov, Bekiroflaz, 3 anonim düzenlemeler

Arayüzler *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=34424 *Katkıda bulunanlar*: Anov, Bekiroflaz, 4 anonim düzenlemeler

Partial (kısmi) tipler *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=31580 *Katkıda bulunanlar*: Anov, Bekiroflaz, 1 anonim düzenlemeler

İstisnai durum yakalama mekanizması *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32136 *Katkıda bulunanlar*: Anov, Bekiroflaz, 2 anonim düzenlemeler

Temsilciler *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32140 *Katkıda bulunanlar*: Anov, Bekiroflaz, Srhat, 1 anonim düzenlemeler

Olaylar *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32141 *Katkıda bulunanlar*: Anov, Bekiroflaz, 1 anonim düzenlemeler

Önişlemci komutları *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32142 *Katkıda bulunanlar*: Anov, Bekiroflaz, 3 anonim düzenlemeler

Göstericiler *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32143 *Katkıda bulunanlar*: Anov, Bekiroflaz, 2 anonim düzenlemeler

Assembly kavramı *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=31592 *Katkıda bulunanlar*: Anov, Bekiroflaz, 1 anonim düzenlemeler

Yansıma *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32144 *Katkıda bulunanlar*: Anov, Bekiroflaz, 1 anonim düzenlemeler

Nitelikler *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32145 *Katkıda bulunanlar*: Anov, Bekiroflaz, 2 anonim düzenlemeler

Örnekler *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=34453 *Katkıda bulunanlar*: Alperkar, Anov, Bekiroflaz, 8 anonim düzenlemeler

Şablon tipler *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32470 *Katkıda bulunanlar*: Altanyuksel, Anov, Bekiroflaz, 1 anonim düzenlemeler

Koleksiyonlar *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32380 *Katkıda bulunanlar*: Anov, Bekiroflaz, 1 anonim düzenlemeler

yield *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32149 *Katkıda bulunanlar*: Anov, Bekiroflaz, 1 anonim düzenlemeler

Veri tabanı işlemleri *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=33962 *Katkıda bulunanlar*: Anov, Bekiroflaz, 5 anonim düzenlemeler

Form tabanlı uygulamalar *Kaynak*: http://tr.wikibooks.org/w/index.php?oldid=32152 *Katkıda bulunanlar*: Anov, Bekiroflaz, 3 anonim düzenlemeler

Resim Kaynakları, Lisanslar ve Katkıda Bulunanlar

Resim:C sharp running.GIF *Kaynak*: http://tr.wikibooks.org/w/index.php?title=Dosya:C_sharp_running.GIF *Lisans*: Public Domain *Katkıda bulunanlar*: Bekiroflaz on Turkish Wikipedia

Resim:Variable definition in c sharp.gif *Kaynak*: http://tr.wikibooks.org/w/index.php?title=Dosya:Variable_definition_in_c_sharp.gif *Lisans*: Public Domain *Katkıda bulunanlar*: Bekiroflaz on Turkish Wikipedia

Resim:Type conversion in c sharp.gif *Kaynak*: http://tr.wikibooks.org/w/index.php?title=Dosya:Type_conversion_in_c_sharp.gif *Lisans*: Public Domain *Katkıda bulunanlar*: Bekiroflaz on Turkish Wikipedia

Lisans