



```
D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum7\kod>Sifrele sefer
Anahtari girin:
4
GQRQF

H:\cskitap\kitap\Bolum7\kod>Sifrele GQRQF
Anahtari girin:
4
sefer

H:\cskitap\kitap\Bolum7\kod>Sifrele GQRQF
Anahtari girin:
5
rdgds

H:\cskitap\kitap\Bolum7\kod>
```

İleriki konularda bu algoritmayı biraz daha geliştirip dosya şifreleme için kullanacağız.

## Sınıflar, Yapılar ve Numaralandırmalar

- **Sınıflar**
- **Sınıf Bildirimi**
  - Sınıf Nesneleri Tanımlama
  - Birden Fazla Sınıf Nesnesi Tanımlama
  - Sınıflara Metot Ekleme
  - this Anahtar Sözcüğü
  - set ve get Anahtar Sözcükleri
  - Özelliklerde Erişim Belirleyiciler
  - Sınıflara Yeni Metot Eklemek
  - Yapıcı Metotlar (Constructors)
    - Varsayılan Yapıcı Metot (Default Constructor)
    - Kopyalayıcı Yapıcı Metot (Copy Constructor)
  - Yıkıcı Metotlar (Destructors) ve Dispose() Metodu
- **Statik Üye Elemanlar**
  - Statik Metotlar
  - Statik Değişkenler
  - Statik Yapıcı Metotlar
- **Statik Sınıflar**
- **const ve readonly Elemanlar**
- **Singleton (Tek) Nesneler**
- **Operatör Aşırı Yükleme (Operator Overloading)**
  - Kompleks Sınıfı
  - Aritmetik Operatörlerinin Aşırı Yüklenmesi
  - İlişkisel Operatörlerin Aşırı Yüklenmesi
  - true ve false Operatörlerinin Aşırı Yüklenmesi
  - Mantıksal Operatörlerinin Aşırı Yüklenmesi
  - Dönüşüm Operatörünün Aşırı Yüklenmesi
  - Operatörlerin Aşırı Yüklenmesine Genel Bakış
- **İndeksleyiciler (Indexers)**
  - Tek Boyutlu İndeksleyici
  - Çok Boyutlu İndeksleyici
- **Yapılar (Structs)**
- **Numaralandırmalar (Enumeration)**
  - System.Enum Sınıfı

Bu bölüm nesne yönelimli programlamaya (*object oriented programming*) giriş niteliğindedir. Çünkü nesne yönelimli programlama tekninin temel yapı taşıları olan sınıfları inceleyeceğiz. Sınıfları incelemeye başlamadan önce Nesne Yönelimli programlamanın ne demek olduğunu kısaca açıklayalım. Çözülmesi istenen problemi çeşitli parçalara ayırıp her bir parça arasındaki ilişkiye gerçeginde uygun bir şekilde belirleme teknigine nesne yönelimi denir. Bu parçalar arasındaki ilişkiye kullanıp büyük çaplı programlar geliştirme teknigine de nesne yönelimli programlama denir. Nesne yönelimli programlamada adından da anlaşıldığı gibi her şey nesnelere dayanmaktadır. Bir dilin %100 nesneye dayalı olması her şeyin bir nesne olmasını gerektirir. C# dilinde her şey bir nesne olduğu için %100 nesne yönelimlidir.

Nesnelerin kaynak kodlarındaki karşılığı tahmin ettiğiniz gibi sınıflardır (*class*). Bu bölümde nesne yönelimli programlama tekniginde kullanılan veri yapısı modeli olan sınıfları yakından inceleyeceğiz. Nesne yönelimli programlama, ileride ayrıca detaylı bir şekilde ele alınacaktır.

Sınıflardan sonra, diğer bir veri modeli olan yapıları (*struct*) inceleyeceğiz. Yapılar, C# dilinde değer tipi olan verilerdir. Bu yüzden değer tipleri ile referans tipleri arasındaki farkı daha iyi anlamız için yapıları iyi kavramanız gereklidir. Ayrıca yapılar ve sınıflar arasındaki farkların neler olduğunu, yapı nesnelerine neden ihtiyaç duyuyor gibi konuları da bu kısımda inceleyeceğiz.

Son olarak çok kullanılan bir veri yapısı olan numaralandırmaları (*enumeration*) göreceğiz.

## Sınıflar

Sınıflar, C# dilinin ve nesne yönelimli programlama tekninin en önemli veri yapısidır. Sınıflar programciye bir veri modeli sunar. Bu veri modeli kullanılarak çeşitli nesneler oluşturulur. Aslında kitabı bu bölümün kadar sınıfları yoğun bir şekilde kullandık. Ancak henüz sınıfların çoğu olanaklarından haberdar değiliz. Örneğin dizi konusunda, dizilerin aslında System.Array dediğimiz bir sınıf türünden nesne olduklarını söylemiştim. Ayrıca bu sınıfın çeşitli özelliklerini ve iş yapan metodlarını kullanmıştım. Örneğin, Array sınıfının Length özelliği bize dizinin eleman sayısını veriyordu, aynı şekilde Array sınıfının Sort() metodu ilgili dizinin elemanlarının belirli bir kurala göre sıralamasını sağlıyordu. Şimdi bu hazır sınıfları kullanmanın yanında kendimiz özel sınıflar bildirip istediğimiz yerde kullanabileceğiz.

Sınıfların üye elemanları değişkenler (özellik) ya da metodlardır. Bazı özel metodlar yapanıcı ve yıkıcı metodlar olarak adlandırılır. Özel metodları anlatmaya başlamadan önce en basit bir sınıf bildiriminin nasıl yapıldığını ve sınıfların temel özelliklerini inceleyeceğiz.

Sınıf bildirimleri *class* anahtar sözcüğü kullanılarak yapılır. *class* anahtar sözcüğünden sonra sınıfın ismi gelir. Sınıf isminden sonra açılan ve kapanan parantezler arasına sınıfın üye elemanları yerleştirilir. Sınıfın üye elemanları metodlar ya da özelliklerdir.

Özellik dediğimiz de aslında şimdije kadar tanımlamış olduğumuz değişkenlerden başka bir şey değildir.

İki metodu ve iki özelliği olan bir sınıfın en genel bildirim şekli aşağıdaki gibi yapılır.

```
class Sınıf_İsmi
{
    <erişim_belirleyici> <veri türü><özellik 1>;
    <erişim_belirleyici> <veri türü><özellik 2>;

    <erişim_belirleyici><geri dönüş tipi>metot1(parametreler);
    {
        Metot Gövdesi
    }
    <erişim_belirleyici><geri dönüş tipi>metot2(parametreler);
    {
        Metot Gövdesi
    }
}
```

Şu ana kadar hiç görmediğimiz bir yapı değil bu. Ancak şimdije kadar erişim belirleyicilerini (access specifiers) hiç kullanmadık. Her metot ve özellik için, bildirim yapmadan önce erişim belirleyici türünü bildirmemiz gereklidir. Erişim belirleyiciler bir metoda ve özelliğe nerelerden erişileceğini tanımlar.

C# dilinde 5 tane erişim belirleyici vardır. Bunlar; public, private, protected, internal ve ‘protected internal’ anahtar sözcükleri ile belirlenir. public erişim belirleyicisi sınıfın bir özelliğine ya da metoduna istenildiği yerden erişilmesini sağlar. Örneğin, Array sınıfının Sort() metodu public olarak tanımlanmıştır. Eğer public olarak tanımlanmamış olsayıdı bizim bu metoda erişmemiz mümkün olmazdı. public üye elemanlar genellikle bir sınıfın dışarıya sunduğu arayüzü temsil eder. Ancak bazı durumlarda private olarak tanımlanmış üye elemanları da bu arayüze dahil edilebilir. Bunun için sınıfın içinde özel metodlar tanımlanır. private olarak tanımlanmış üye elemanlara dışarıdan erişmek mümkün değildir. private üye elemanlara ancak tanımladığı sınıfın içerisindeki üye elemanlar erişebilir. private olarak tanımlanan üye elemanları genellikle sınıfın metodlarını gerçeklerken birtakım ara değerleri tutmak ve işe yarayan metodları kullanabilmek için tanımlanırlar. Diğer bir erişim belirleyicisi olan protected ise sınıfı kullanan için private gibidir. İleride göreceğimiz kalıtım konusunda protected erişim belirleyicisinin özel anlamını göreceğiz. Simdilik protected erişim belirleyicisini kullanmayacağız. ‘protected internal’ ve internal erişim belirleyicisinin de henüz görmediğimiz konularla alakalı olduğu için bu konuda anlatılmayacaktır.

Bu bölümde sadece public ve private erişim belirleyicilerini kullanacağız. private erişim belirleyicisi ile işaretlenen üye elemanlara programımız içinde ulaşamayacağız. public olan elemanları ise tanımlandıkları sınıfların dışında rahatlıkla kullanabileceğiz.

public ve private belirleyicilerinin farkını görmek açısından ilerde çeşitli örnekler yapacağız.

## Sınıf Bildirimi

Sınıflar, daha önce de denildiği gibi, `class` anahtar sözcüğü kullanılarak bildirilir. Sınıfın bütün üye elemanları sınıf bildiriminin yapıldığı parantezler içerisinde yapılır. C++ dilinde ise sınıf bildiriminin dışında sınıfa ait bir metodun gövdesi yazılabilir. Sınıfların üye elemanı olan özellikler temel veri türleri olabileceği gibi kendi tanımladığımız sınıflar da olabilir. Ancak şimdilik sınıflarımızı sadece temel veri türlerini kullanarak oluşturacağız.

Daha önceden dediğimiz gibi sınıflar, birer veri yapısıdır. Bir sınıfı gerçek hayatı herhangi bir nesneye yönelik modelleyebiliriz. Mesela bir kredi kartı hesabını düşünün. Bir kredi kartı hesabının çeşitli özellikleri vardır, mesela hesabın numarası, hesabın kredi limiti, günlük limiti, hesabın sahibi ile ilgili özellikler kredi kartının sayabileceğimiz çeşitli özellikleridir. Bir bankacılık sistemi ile ilgili program geliştirmek yorsak kredi kartını temsil eden bir veri modeli geliştirmek bir örnek olabilir.

Şimdi bir kredi kartının belirli özelliklerinden olan hesap numarası, limit ve kart sahibi gibi 3 temel bilgiyi içeren bir sınıf tasarlayalım. Sınıfımızı tasarlama başlamadan önce sınıfın amacına uygun bir isim belirlememiz gereklidir. Kredi kartı ile ilgili verileri tutan bir sınıfın ismini Araba olarak vermek herhalde çok saçma olacaktır. Sınıfımızın ismini "KrediHesabı" olarak seçelim. (Siz daha güzel bir isim bulursanız onu kullanın.)

Düşündüğümüz sınıfın şimdilik sadece özellikleri (değişkenleri) mevcut. Konunun ilerleyen kısımlarında "KrediHesabı" sınıfına çeşitli metotlar da ekleyeceğiz. Şimdi bu sınıfı nasıl bildireceğimizi görelim.

```
class KrediHesabi
{
    public ulong HesapNo;
    public double Limit;
    public string KartSahibi;
}
```

KrediHesabi sınıfının bildiriminde sadece değişkenler yani özellikler var. Bu özelliklerin her biri en uygun veri türleri cinsinden bildirilmiştir. `HesapNo` `ulong` türünden, `Limit` özelliği `double` türünden ve `KartSahibi` özelliği `string` türünden bildirilmiştir. Dikkat ederseniz bütün elemanlar `public` olarak bildirilmiştir. `Public` olarak bildirildiğine göre KrediHesabi türünden tanımladığımız nesneler üzerinden bu özelliklerin tümüne erişebiliriz. Eğer herhangi bir erişim belirleyicisi kullanmamış olsaydık varsayılan olarak `private` olurdu.



Sınıf bildirimleri için bellekte yer tahsis edilmez. Sınıf bildirimleri sonradan tanımlayacağımız nesneler için derleyiciye nesnenin neye benzeyeceğini gösterir.

## Sınıf Nesneleri Tanımlama

Sınıf türünden nesnelerin tanımlanması daha önceki konularda da gördüğümüz gibi `new` anahtar sözcüğü kullanılarak yapılır. Sınıf nesnelerini bir değişken gibi düşünelim. O halde sınıf türünden bir nesnenin bildirimi aşağıdaki şekilde yapılabilir.

```
KrediHesabi hesap1;
```

Bu bildirim ile `KrediHesabi` türünden `hesap1` değişkeni bildiriliyor; fakat, henüz bellekte üye elemanları tutmak için yer tahsis edilmedi. Üye elemanlar için bellekte alan tahsisatı yapmak için `new` anahtar sözcüğü kullanılır. `new` anahtar sözcüğü ile yeni bir `KrediHesabi` nesnesinin tanımlanması aşağıdaki gibi yapılır.

```
KrediHesabi a = new KrediHesabi();
```

Bu işlemi aşağıdaki gibi bildirimin ve tanımlamanın farklı satırlarda olacak şekilde de gerçekleştirebiliriz.

```
KrediHesabi hesap1;
```

```
KrediHesabi a = new KrediHesabi();
```

Şimdi `KrediHesabi` sınıfını bir program içinde kullanalım ve bu şekildeki bir tanımlamada üye elemanlarının durumunu inceleyelim.

Aşağıdaki programda `KrediHesabi` sınıfı, `Main()` metodunun içinde bulunduğu sınıfın dışında tanımlanmıştır. Bu iki sınıfta aynı kaynak dosyası içinde olduğu için `csc` derleyicisi ile şu ana kadar derlediğimiz şekilde derleyeceğiz. Derleme işlemi sırasında, tanımlanan hesap isimli nesnenin üye elemanlarına atama yapmadığımıza dair bir uyarı alırız. Bu uyarının olmaması için ileride anlatacağımız çeşitli özel metotlar yazacağız.

Şimdi aşağıdaki programı inceleyelim:

```
using System;
class KrediHesabi
{
    public ulong HesapNo;
    public double Limit;
    public string KartSahibi;
}

class AnaSınıf
{
    static void Main()
    {
        KrediHesabi hesap = new KrediHesabi();

        Console.WriteLine(hesap.HesapNo);
        Console.WriteLine(hesap.Limit);
        Console.WriteLine(hesap.KartSahibi);
    }
}
```

Main() metodu her zaman olduğu gibi yine bir sınıf içinde yazılmıştır. Ancak bu sefer bu sınıfın dışında KrediHesabi adlı bir sınıf daha bildirdik. Bu sınıf kullanılarak "hesap" isimli bir nesne tanımlanmıştır. Sınıflar, new anahtar sözcüğü ile tanımlandığında sınıfın içindeki bütün üyeleri varsayılan değere atanır. Yani bu programı derleyip çalıştığımızda ekran altalta iki tane 0 yazacaktır. KartSahibi'ne ise null değeri atandığı için ekran da görünmez.

Şimdi Main() metodunu aşağıdaki gibi değiştirdiğimizde sınıfın üye elemanlarını nasıl değiştirdiğimizi inceleyelim.

```
static void Main(string[] args)
{
    KrediHesabi hesap = new KrediHesabi();
    hesap.HesapNo = 5657846985;
    hesap.Limit = 500000000;
    hesap.KartSahibi = "Sefer Algan";

    Console.WriteLine(hesap.HesapNo);
    Console.WriteLine(hesap.Limit);
    Console.WriteLine(hesap.KartSahibi);
}
```

Bu programı derleyip çalıştığımızda ekran'a

```
5657846985
500000000
Sefer Algan
```

yazacaktır.

Yukarıdaki programdan da görüldüğü üzere, bir nesnenin üyelerine ? operatörü ile ulaşabiliyoruz. Eğer bu üye elemanlar public olarak tanımlanmışsa üye elemanın değerini de istediğimiz gibi değiştirebiliyoruz.

Şimdi KrediHesabi sınıfının Limit özelliğini private olarak tanımlayalım. HesapNo elemanını herhangi bir erişim belirleyicisi ile bildirmeyelim. KartSahibi ise yine public olarak kalsın. Bu durumda hangi elemanlara ulaşabildiğimizi inceleyelim. Sınıfımızın yeni görünümü aşağıdaki gibi olmalıdır.

```
class KrediHesabi
{
    ulong HesapNo;
    private double Limit;
    public string KartSahibi;
}
```

Yeni bir nesne tanımladığımızda bu nesne üzerinden sadece KartSahibi özelliğine erişebiliriz. HesapNo ve Limit private olduğu için bu elemanlara erişmek mümkün de-

ğildir. ? operatörü ile private olan Limit elemanına hesap.Limit = 500000000; şeklinde ulaşmaya çalıştığımızda; 'KrediHesabi.Limit' is inaccessible due to its protection level' yani 'KrediHesabi.Limit' elemanına koruma düzeyinden dolayı erişilemez' hatası alırız.

Herhangi bir erişim belirleyicisi ile işaretlenmemiş elemanlar, private olarak kabul edilir.

private olan elemanlara herhangi bir değer atayamayacağımız gibi elemanları sadece okuma amaçlı olarak da kullanamayız. Örneğin

```
Console.WriteLine(hesap.Limit);
```

deyimini kullandığımızda yukarıdaki hatanın aynısını alırız.

## Birden Fazla Sınıf Nesnesi Tanımlama

Programımız içinde aynı sınıf türünden birden fazla nesneyi tanımlayabileceğimizi tahmin etmişsinizdir. Ancak tanımladığımız nesneleri birbirlerine atarken dikkat etmemiz gereken birtakım önemli noktalar vardır. Önceki bölümlerde değer ve referans tipleri arasındaki temel farkları incelemiştik. Değer tipleri birbirlerine atanırken bitsel bir kopyalama işleminin olduğunu söylemiştık. Ve atama işleminden sonra her bir nesnenin farklı bellek bölgesindeki verileri temsil ettiğini söylemişük. Ancak referans tiplerinin birbirlerine atanması sırasında bu geçerli değildir. Referans türleri değer türlerinden farklı işlem görürler, referans tipi olan iki nesneyi birbirlerine atadığımızda nesnelerin eleman değerleri tek tek kopyalanmaz. Atanan değerler dinamik bellek bölgesindeki nesnelere referans olan adreslerdir. Yani referans türünden iki nesneyi atama operatörü ile eşitlersek iki nesne de aynı bellek bölgesindeki verileri temsil eder, bir nesne üzerinde yaptığı değişiklik diğer nesneyi de etkiler.

Bu kavramları daha iyi anlayabilmek için KrediHesabi türünden iki tane nesne tanımlayıp iki nesneyi atama operatörü ile birbirlerine eşitledikten sonra nesnelerin elemanlarını ekran'a yazdırıralım.

Aşağıdaki programı inceleyin ve program çalıştırınca ekran'a ne yazacağını tahmin edin.

```
using System;
class KrediHesabi
{
    public ulong HesapNo;
    private double Limit;
    public string KartSahibi;
}
class AnaSınıf
```

```

{
    static void Main(string[] args)
    {
        KrediHesabi hesap1 = new KrediHesabi();
        KrediHesabi hesap2;

        hesap1.HesapNo = 5698457458;

        Console.WriteLine(hesap1.HesapNo);

        hesap2 = hesap1;

        Console.WriteLine(hesap2.HesapNo);

        hesap2.HesapNo = 1111111111;

        Console.WriteLine(hesap1.HesapNo);
    }
}

```

Yukarıdaki programda hesap1 nesnesi için bellekte yer ayrılmıştır. Çünkü **new** anahtar sözcüğü kullanılmamıştır. hesap2 için sadece stack bellek bölgesinde bir referans tutabilecek kadar alan ayrılmıştır.

```
hesap2 = hesap1;
```

İfadesinden sonra hesap2 değişkeninin tuttuğu referans hesap1'in değişkeninin tuttuğu referans ile aynı olmaktadır. Yani her iki değişken de heap alanında aynı bellek bölgesini göstermektedir. Dolayısıyla bu nesnelerin birinde yapılan değişiklik diğerini de etkileyecektir. Nitekim;

```
hesap2.HesapNo = 1111111111;
```

deyiği ile sadece hesap2'nin değil aynı zamanda hesap1'in de HesapNo elemanı değiştirilmiştir. Yukarıdaki programı çalıştırarak bunu rahatlıkla görebilirsiniz.

Bu işlemlerin aynısını bir de değer tipleri için yapalım;

```

using System;
class AnaSınıf
{
    static void Main()
    {
        int a;
        int b;

        a = 20;
        b = a;
    }
}

```

```

Console.WriteLine(b);

b = 10;
Console.WriteLine(a);
Console.WriteLine(b);
}
}

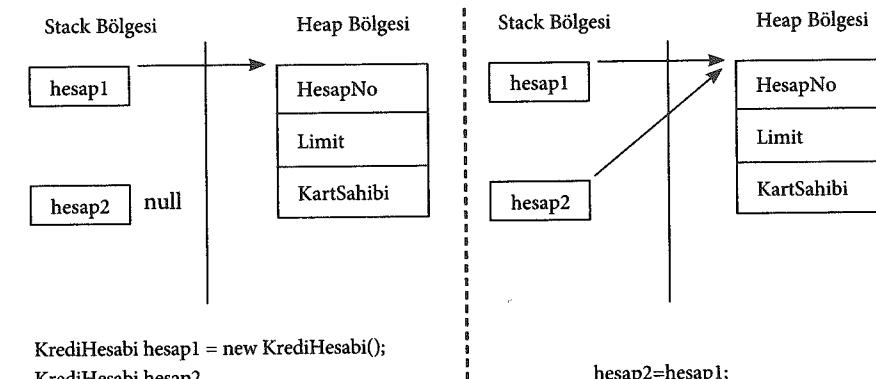
```

Yukarıdaki programı derleyip çalıştırıldığımızda ekrana

```
20
20
10
```

yazdığını görürsünüz. **b=a;** deyiminden sonra b nesnesini değiştirmek a nesnesini değiştirmemiştir.

Referans tiplerinin birbirlerine atanması ile ilgili bütün anlatılanların özetini aşağıdaki şemada görebilirsiniz.



## Sınıflara Metot Ekleme

Bölümün başında da denildiği gibi, değişkenler ve metodlar bir sınıfı oluşturan temel elemanlardır. Bu kısımda sınıflarımıza metodlar ekleyip, birtakım faydalı işler yapacağız. Metotlar konusunu bir önceki bölümde işlediğimiz için metot bildirimini ile ilgili temel bilgiler burada ayrıca anlatılmayacaktır. Metotlar ile ilgili temel bilgileri hatırlamak için bir önceki bölümü tekrar gözden geçirebilirsiniz.

Bu kısımda farklı bir sınıf tasarlayacağız. Bu sınıf, geometrik bir şekil olan dikdörtgeni temsil edecektir. Okul yıllarınızdan hatırlarsanız bir dikdörtgenin eni ve boyu vardır, bu en ve boy birbirlerinden farklı değerlerdedir. Eğer dikdörtgenin eni ve boyu eşitse bu özel bir Dikdörtgen olur; buna da kare denir.

Şimdi Dortgen adlı bir sınıf tasarlayacağız. Sınıfımızın en ve boy adlı iki tane üye değişkeni olacak. Bir tane de iş yapan metot ekleyeceğiz. Bu metot dörtgenin alanını hesaplayıp bu değer ile geri dönecektir. Üye elemanlarına dışarıdan erişebilmemiz için bütün elemanlar public olarak bildirilecektir.

Dortgen sınıfının yapısı aşağıdaki gibidir.

```
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;
    }
}
```

Dortgen sınıfının her iki metodu da sınıfın içinde bildirilmiştir. Eğer **EnBoyBelirle()** metodunu bildirmemiş olsaydık, Dortgen sınıfından bir nesne yarattığımızda enini ve boyunu tek tek aşağıdaki gibi belirlememiz gerekecekti.

```
Dortgen d = new Dortgen();
d.En = 10;
d.Boy = 20;
```

**EnBoyBelirle()** metodu ise bizim için bu işi biraz daha kolaylaştırmaktadır. Bu metodu aşağıdaki gibi kullanarak oluşturduğumuz Dortgen nesnesinin En ve Boy özelliklerini belirleyebiliriz.

```
Dortgen d = new Dortgen();
d.EnBoyBelirle(10,20);
```

Şimdi gerçek bir uygulamada farklı Dortgen nesneleri oluşturarak sonuçları ekrana yazdırıralım. Bu örnekte ayrıca **Alan()** metodunu da kullanarak Dortgen nesnelerinin alanlarını ekrana yazdıracağız.

Bu programı yazmaya başlamadan önce Dortgen sınıfına kullanışlı bir metot daha ekleyelim. Bu metot ile Dortgen nesneleri ile ilgili bilgileri ekrana yazdırmak isteyelim. Örneğin bir Dortgen nesnesinin bilgilerini aşağıdaki gibi ekrana yazsın:

\*\*\*\*\*

```
En : 20
Boy : 50
Alan : 1000
*****
```

**Yaz()** isimli metot aşağıdaki gibi yazılabılır. Yaz metodunun geri dönüş değeri ve herhangi bir parametresinin olmadığına dikkat edin.

```
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("*****");
}
```

**Yaz()** metodu sınıfın içinde olduğu için sınıfın diğer üye elemanlarını direkt kullanabiliyoruz. Örneğin,

```
Console.WriteLine("Alan :{ 0,5} ",Alan());
```

satırında **Alan()** metodu herhangi bir nesne üzerinden çağrılmamıştır. Bu metot hangi Dortgen nesnesi için çağrılsa o dortgenin alanı hesaplanır.

Şimdi oluşturduğumuz Dortgen sınıfının kullanımına bir örnek verelim:

```
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 AnaSınıf
{
    static void Main()
    {
        Dortgen d1 = new Dortgen();
        d1.EnBoyBelirle(20,50);
        d1.Yaz();

        Dortgen d2 = new Dortgen();
        d2.EnBoyBelirle(0,0);
        d2.Yaz();
    }
}

```

Bu programı derleyip çalıştırıldığımızda aşağıdaki ekran görüntüsünü elde ederiz.

```

D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum8\kod>Dortgen
*****
En : 20
Boy : 50
Alan : 1000
*****
En : 0
Boy : 0
Alan : 0
*****
H:\cskitap\kitap\Bolum8\kod>

```

Henüz sınıflarla ilgili anlatmadığımız çok daha güzel özellikler olmasına rağmen sınıfları kullanarak programlarımızın gittikçe profesyonelleştiğini görmek insana gerçekten zevk veriyor.

Tasarladığınız sınıfları ayrı dosyalar halinde de saklayabilirsiniz. Örneğin Dortgen sınıfını Dortgen.cs dosyasında AnaSınıf'ını da program.cs dosyasında saklayın. Derleme işlemi komut satırında her bir dosya aşağıdaki gibi ayrı ayrı belirtilerek yapılır.

```
csc Dortgen.cs program.cs
```

Bu işlemin başarılı olması için bütün dosyalara

```
using System;
```

deyiminin eklenmesi gerektiğini unutmayın.

Şimdi yukarıdaki örnek programa tekrar geri dönelim. EnBoyBelirle() metodu negatif sayılarla çağrıduğunda hiçbir hata vermeden çalışır. Ama gerçekten biliyoruz ki bir dörtgenin eni ya da boyu negatif bir sayı olamaz. Bu yüzden EnBoyBelirle() metodu bu şekildeki bir kullanımı engelleyecek biçimde yeniden yazmamız gereklidir.

EnBoyBelirle() metodunun parametrelerini kontrol edip, eğer bu parametrelerden herhangi biri negatif bir sayı ise işlemlerimizi gerçekleştirmemeliyiz. Peki, işlemleri gerçekleştirmeyip de ne yapacağız? Simdilik bu sorunun cevabını en iyi şekilde verebilmemiz mümkün değildir. Çünkü henüz istisnai durum yakalama dediğimiz mekanizmayı işledemek. Ancak bu konuyu işleyene kadar bir negatif parametre geldiğinde En ve Boy değerlerini sıfır eşitleyebiliriz. Bunu bu şekilde önlemış olmak programlarda birçok hata meydana getirmektedir. Çünkü herhangi bir uyarı verilmeden En ve Boy özelliklerini sıfır atanacaktır.

EnBoyBelirle() metodunun yeni ve gelişmiş versiyonu aşağıdaki gibidir.

```

public void EnBoyBelirle(int en,int boy)
{
    if(en < 0 || boy < 0)
    {
        En = 0;
        Boy = 0;
    }
    else
    {
        En = en;
        Boy = boy;
    }
}

```

## this Anahtar Sözcüğü

Nesneler üzerinden metodlar çağrılrken aslında biz farkına varmadan metodlara gizlice ilgili nesnelerin referansları geçirilir. Örneğin, **this** anahtar sözcüğünün anlamını daha iyi kavrayabilmek için bir soyutlama yapalım,

```
Dortgen d1 = new Dortgen();
d1.EnBoyBelirle();
```

yerine aşağıdaki gibi bir kullanım da olabilir:

```
Dortgen d1 = new Dortgen();
d1.EnBoyBelirle(d1' in referansı);
```

Eğer **this** anahtar sözcüğü olmasaydı ikinci şekildeki gibi bir kullanıma mecbur kaldacaktık. İkinci şekildeki gibi bir kullanımın kullanışsız olduğunu söyleyebiliriz. Bu yüzden C# dilini tasarlayanlar d1 nesnesinin referansını gizlice **EnBoyBelirle()** me-

toduna aktarmada kullanılacak bir sistem geliştirmişlerdir. İşte **this** anahtar sözcüğü bu referansı temsil etmektedir. **this** anahtar sözcüğü çok fazla kullanılmasa da bazı durumlarda faydalı olabilir; örneğin, **EnBoyBelirle()** metodunun aşağıdaki gibi olduğunu düşünelim:

```
public void EnBoyBelirle(int En, int Boy)
{
    if(en < 0 || boy <0)
    {
        En = 0;
        Boy = 0;
    }
    else
    {
        En = En;
        Boy = Boy;
    }
}
```

**EnBoyBelirle()** metodunun parametreleri olan En ve Boy değişkenleri **Dortgen** sınıfının üye elemanları ile aynı isme sahiptir. Metot gövdesinde kullandığımız En ve Boy değişkenleri **EnBoyBelirle()** metodunun parametreleridir. Çünkü bir üst faaliyet alanında olan **Dortgen** sınıfının En ve Boy değişkenleri **EnBoyBelirle()** metodunun parametreleri tarafından görünmez hale gelmiştir.

**EnBoyBelirle()** metodunu bu haliyle derleyip çalıştırduğumızda En ve Boy değişkenlerine 0 değerinin atandığını görürüz. Çünkü sınıfın üye elemanlarına herhangi bir değer atanmamıştır. Dolayısıyla sınıfın bütün elemanları varsayılan değer olan 0'da kalmıştır.

Böyle bir durumda **this** anahtar sözcüğünden faydalılabılır. **this** anahtar sözcüğü ilgili nesnenin referansını belirttiğine göre **EnBoyBelirle()** metodunu aşağıdaki gibi kullanarak mantıksal bir hatayı önlemiş oluruz.

```
public void EnBoyBelirle(int En, int Boy)
{
    if(en < 0 || boy <0)
    {
        En = 0;
        Boy = 0;
    }
    else
    {
        this.En = En;
        this.Boy = Boy;
    }
}
```

**this** anahtar sözcüğü ile **EnBoyBelirle()** metodunun En ve Boy parametreleri, o an üzerinde çalışılan **Dortgen** nesnesinin En ve Boy özelliklerine atanıyor.

**this** anahtar sözcüğünün kullanımına olanak tanımamak okunabilirlik açısından önemlidir.



## set ve get Anahtar Sözcükleri

**Dortgen** sınıfında En ve Boy elemanlarının neden public olarak bildirildiğini merak etmişsinizdir. En ve Boy elemanlarına public olduklarından dolayı direkt erişebilmemiz mümkündü. Ancak buna rağmen **EnBoyBelirle()** gibi bir metot ile elemanları değiştirmeyi tercih ettilik. Bunun en önemli sebeplerinden biri elemanlara direkt ulaşımı engellemektir. Çünkü herhangi bir metot yazmadan girilen değerin belirli bir aralıkta olduğunu kontrol edemeyiz. Bu yüzden **EnBoyBelirle()** gibi bir metot kullandık. Bazen de bu elemanları değiştirmek yerine bu elemanlarda tutulan değerleri de kullanmak isteyebiliriz. O zaman yapmamız gereken her bir eleman için elemanın değerine geri dönen bir metot bildirmek. Örneğin bir **Dortgen** nesnesinin En özelliğini veren metodu aşağıdaki gibi bildiririz.

```
public int dEn()
{
    return En;
}
```

Aynı şekilde bir **Dortgen** nesnesinin "En" özelliğini değiştiren bir metot da yazılabilir. Bunun için de aşağıdaki metot kullanılabilir.

```
public void dBoy(int en)
{
    if (en<0)
        En=0;
    else
        En=en;
}
```

Gördüğünüz gibi **dBoy()** metodu ile En özelliğine koşullu olarak değerler atanıyor.

Bildirdiğimiz metotların anlamlı olması için **Dortgen** sınıfının En ve Boy elemanlarının private olması gereklidir, yani sadece **Dortgen** sınıfı içindeki üye metotlar tarafından kullanılabilir olması gereklidir.

C# dilinde her üye özellik için iki ayrı metot bildirmek yerine, **get** ve **set** anahtar sözcüklerini kullanarak da sanki iki ayrı metot bildirmiş gibi oluruz. **get** anahtar sözcüğü **dEn()** metodunun yaptığı gibi üye elemanın değerine geri döner. **set** anahtar sözcüğü ise üye özelliğinin belli bir değere atanması için kullanılır.

Şimdi **get** ve **set** anahtar sözcüklerinin kullanımına bir örnek verelim. Bu örnekte **Dortgen** sınıfının yeni bir versiyonu yazılmaktır. Öncelikle En ve Boy özellikleri private olarak değiştirilecektir. Bu elemanlara erişmek için ise **get** ve **set** erişimleri kullanılacak.



get ve set anahtar sözcükleri ile bir özellik tanımlanır. Ancak bu özelliklere erişmek için belirli kod bloklarını çalışma şansımız mevcuttur. Metotlar ve özellikler arasındaki tek fark özellikler çağırırken, fonksiyon çağrılmak için kullandığımız parantezleri kullanmamamızdır.

private olarak tanımlanan En ve Boy elemanlarının Dortgen sınıfı için bir arayüz oluşturmayıcağı için ikisinin de önüne member (üye) sözcüğünü temsil eden "m" harfini koyacağiz. Böylece orijinal isimler get ve set erişimlerini kullanacağımız özelliklerde kalacaktır.

Dortgen sınıfının yeni arayüzü aşağıdaki gibidir.

```
class Dortgen
{
    private int mEn;
    private int mBoy;

    public int En
    {
        get
        {
            return mEn;
        }

        set
        {
            if (value <0)
                mEn=0;
            else
                mEn=value;
        }
    }

    public int Boy
    {
        get
        {
            return mBoy;
        }

        set
        {
            if (value <0)
                mBoy=0;
            else
                mBoy=value;
        }
    }
}
```

```
public int Alan()
{
    int Alan = En*Boy;
    return Alan;
}

public void EnBoyBelirle(int en,int boy)
{
    if(en < 0 || boy <0)
    {
        En = 0;
        Boy = 0;
    }
    else
    {
        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("*****");
}
```

Yukarıdaki Dortgen sınıfında karşılaştığımız yeni bir anahtar sözcük var: **value**. Bu anahtar sözcük, özelliğe atanacak nesnenin değerini ifade eder. Örneğin,

```
Dortgen d = new Dortgen();
d.En = 50;
```

ifadesinde **value** anahtar sözcüğü 50 değerini temsil etmektedir. Buradaki **value** değerinin önceden belirlenmiş herhangi bir türü yoktur. Veri elemanı hangi türden ise **value** değeri de o türden olur.

```
d.En = 50;
```

**deyimi** ile En özelliğinin set bloklarındaki kodlar çalıştırılır. Yani  $50>0$  olduğu için d nesnesinin mEn üye elemanına 50 değeri atanır. Aynı şekilde,

```
int a = d.En;
```

**deyimi** ile En özelliğinin get blokları arasındaki kod çalıştırılır. get blokları mEn değişkenine geri döndüğü için, a değişkenine o anki mEn'in değeri atanır. Gördüğünüz

gibi bu ifade ile aslında d nesnesinin mEn değişkenine erişiyoruz. "En" özelliği, bizim için mEn değişkenine erişmek için bir aracı gibi çalışmaktadır.

**get** ve **set** anahtar sözcüklerini kullandığımız En ve Boy özellikleri sadece bir erişim aracıdır. Veriler aslında mEn ve mBoy değişkenlerinde tutulur. En ve Boy özelliklerini bu değişkenlere çeşitli şekillerde erişmek için kullanırız.

**get** ve **set**'in en yaygın kullanıldığı alanlardan biri, bir özelliğin değiştiği anda belirli olaylar zincirinin çalışmasını sağlamaktır. Örneğin windows pencerelerinde göreceğimiz pencere.Height = 250;

deyimi pencerenin boyutunu çalışma zamanında 250 pixel olarak değiştirir. Aslında değiştirdiğimiz sadece bir değişkenin değeri olmasına rağmen pencerenin boyutundaki artışı da gözlemleriz. Bunu sağlayan Height özelliğinin set bloklarındaki kodlardır.



Bir değişkenin değerini değiştirdiğimizde çalışmasını istediğimiz kodları set blokları arasına yazarız.

Dortgen sınıfında da buna benzer bir kullanım vardır. Örneğin aşağıdaki programı yapın ve derleyin. (Tabi programı derlemeden önce Dortgen sınıfının bildirimini kaynak koda eklemeniz gereklidir)

```
class AnaSınıf
{
    static void Main()
    {
        Dortgen d1 = new Dortgen();
        d1.EnBoyBelirle(20, 50);
        d1.En = -50;
        Console.WriteLine(d1.En);
    }
}
```

Programı çalıştırığınızda ekrana 0 yazdığını göreceksiniz. Oysa bize d1.En değişkenine -50 değeri atamışık. Gördüğünüz gibi set blokları sayesinde En değişkenine 0'dan küçük değerlerin atanmasına engel olduk. Bu işlemleri get ve set blokları olmadan ya da mEn ya da mBoy nesnelerine geri dönen bir metot yazmadan yapamazdık.

get ve set bloklarından yalnızca birini tanımlamamız da mümkün olabilmektedir. Örneğin sadece get bloğu olan bir özellik sadece okunabilir bir özelliktir. Sadece set bloğu olan özellikler ise pratikte pek kullanışlı olmasa da yine de bu şekilde bildirimler yapabiliriz.

## Özelliklerde Erişim Belirleyiciler

Aynen metodlarda olduğu gibi **özellikler** (*property*) ile birlikte de erişim belirleyiciler (*access modifiers*) kullanılabilmektedir. Özelliklerde erişim belirleyicileri iki şekilde kullanılabilir. Birinci yöntem, özellik bildirim satırında aşağıdaki gibi bildirmektir.

```
private int m_uzunluk
public int Uzunluk
{
    get
    {
        return m_uzunluk;
    }
    set
    {
        m_uzunluk = value;
    }
}
```

Bu durumda hem **set** hem de **get** blokları **public** olduğundan dışarıdan erişilebilir durumdadır. Yani **get** ve **set** otomatik olarak **public** duruma gelmiştir.

C# 2.0 ile birlikte gelen yeniliklerden biri de **get** ve **set** bloklarının farklı erişim belirleyicileri ile belirleyebilmemizdir. Bazı durumlarda set bloğunun sınıfa özel olmasını isteyebiliriz. Bu durumda hem özelliklerin faydasından yoksun kalmamak hem de kurallarımızı tam işletebilmek için **get** ve **set** bloklarını farklı erişim belirleyicileriyle bildirebiliriz. Örneğin **get** bloğunu **public**, **set** bloğunu **private** yapabiliriz. Bu özellik sadece C# 2.0'da kullanılabilmektedir.

**Önemli Not:** Özelliklerin farklı erişim belirleyicilerle bildirilmesi sadece C# 2.0 ve sonrasında geçerlidir. C# 1.0 ve C# 1.1 de geçerli bir bildirim şekli değildir.



Bu durumda özellik bildirimini aşağıdaki gibi yapabiliriz;

```
public int Uzunluk
{
    get
    {
        return m_uzunluk;
    }
    private set
    {
        m_uzunluk = value;
    }
}
```

Yukarıdaki bildirimden sonra set bloğuna sadece sınıfın içinden erişilebilir. Yani herhangi bir nesnenin Uzunluk isimli özelliğine dışarıdan herhangi bir değer atayamayız. Dikkat ederseniz get bloğunda herhangi bir erişim belirleyicisi bildirilmemiş. Bu, get bloğunun Uzunluk özelliğinde tanımlanan şekliyle otomatik olarak public olacağını göstermektedir. Bu durumda get bloğuna ayrıca public yazmak gereksiz olacağının zaten yasaklanmış durumdadır.

Özelliklerin erişim belirleyicileri hakkında bilinmesi gereken en önemli nokta, özellik bildiriminde bildirilen erişim belirleyicisinin get veya set bloğunda tanımlanan erişim belirleyicisinden daha yüksek erişim yetkisine sahip olması gerektidir. Örneğin, özellik private olarak bildirilmişse get ya da set blokları public olamaz. Aynı şekilde, özellik public olarak bildirilmişse set veya get yine public olamaz. Kısaca özellik public ise set ya da get blokları **private**, **protected**, **internal** ya da **protected internal** olabilir. (public ve private dışındaki diğer erişim belirleyiciler kitabın ilerleyen bölgümlerinde anlatılacaktır.)

## Sınıflara Yeni Metot EklemeK

Bir kere sınıflarımızın yapısını iyi tasarladıkta sonra istediğimiz kadar yeni metot ekleyebiliriz. Örneğin Dortgen sınıfına eklenebilecek metodlardan ikisi Dortgen'in kare olup olmadığını kontrol eden ve dortgenin köşegen uzunluğunu bulan metodlar olabilir. Her iki metodun da parametre almasına gerek yok çünkü iş yaparken mEn ve mBoy değişkenlerinden faydalananacaklar. Birinci metodun geri dönüş değeri bool, ikinci metodun geri dönüş değeri ise double türünden olmalı. Bu metodları aşağıdaki gibi oluşturabiliriz.

```
public bool Karemi()
{
    if (mBoy == mEn)
        return true;
    else
        return false;
}

public double KosegenBul()
{
    double Kosegen;
    Kosegen = Math.Sqrt(mBoy*mBoy + mEn*mEn);

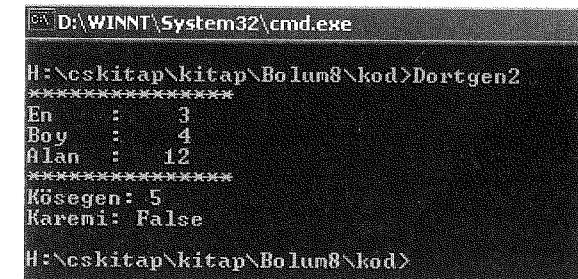
    return Kosegen;
}
```

Bu metodları aşağıdaki Main() metodunu yazarak test edebilir, bu metodları istediğiniz gibi geliştirebilirsiniz. Örneğin Dikdortgen'in içine çizilebilecek en büyük dairenin alanını veren metod gibi...

```
using System;

//Burada Dortgen sınıfı olmalı

class AnaSınıf
{
    static void Main()
    {
        Dortgen d1 = new Dortgen();
        d1.EnBoyBelirle(3, 4);
        d1.Yaz();
        Console.WriteLine("Kösegen: ");
        Console.WriteLine(d1.KosegenBul());
        Console.WriteLine("Karemi: ");
        Console.WriteLine(d1.Karemi());
    }
}
```



Yukarıdaki programı en son tasarladığımız Dortgen sınıfını kullanarak derleyip çalıştırduğumda aşağıdaki ekran görüntüsünü elde ederiz.

## Yapıcı Metotlar (Constructors)

Şu ana kadar gördüğümüz bütün örneklerde bir sınıf nesnesinin üye elemanlarından olan özelliklere değerler verirken ya kendimiz EnBoyBelirle() gibi metodlar tanımladık ya da ? operatörü ile eğer eleman public ise direkt ulaştık. Ancak bu şekilde ilk değer vermeler genellikle kullanılmaz, nesne yönelimli programlama tekniğinin getirdiği birtakım avantajlardan faydalanjırlar.

Bir nesnenin dinamik olarak yaratıldığı anda otomatik olarak çalıştırılan metodlar vardır. Bu metodlar sayesinde bir nesnenin üye elemanlarına ilk değerler verilir ya da sınıf nesnesi için gerekli kaynak düzenlemeleri yapılır. (örneğin dosyalarla ilgili bir sınıf ise dosyaların açılması gibi)

Kitabın 2. bölümünden beri referans tipi değişkenler oluşturulduğunda bütün elemanlarına varsayılan değer atandığını söylemiştık: (nümerik değişkenler için 0, refe-

rans tipler için null, bool türü için false). İşte bu işlemi yapan aslında otomatik olarak çalıştırılan bir yapıcısı metottur. Buradan da anlaşılabilir, her sınıfın biz tanımlasak da tanımlamasak da kesin bir yapıcısı metodu vardır. İşte bu metoda varsayılan yapıcısı metot (**default constructor**) denilmektedir.

Şimdi örnek bir sınıf üzerinden yapıcısı metodlarının nasıl bildirildiğine bakalım. Yapıcısı metodların diğer metodlardan iki farkı vardır. Birincisi, yapıcısı metodların geri dönüş değeri yoktur yani geri dönüş değeri kavramı yoktur; ikincisi, yapıcısı metodların ismi sınıf ismi ile aynı olmak zorundadır.



**Yapıcı metodların geri dönüş değerinin olmaması geri dönüş tipinin void olduğu anlamına gelmemektedir.**

Şimdi Zaman adlı bir sınıf için bir yapıcısı metod oluşturalım. Zaman sınıfı içinde saat, dakika ve saniye bilgisini tutan bir veri yapısı olacaktır. Zaman sınıfının bu 3 üye değişkeni public olarak tanımlanacaktır. Ancak siz get ve set blokları tanımlayarak üyelerin private olmasını sağlayabilirsiniz.

```
class Zaman
{
    public int Saat;
    public int Dakika;
    public int Saniye;

    public Zaman(int h, int m, int s)
    {
        Saat = h;
        Dakika = m;
        Saniye = s;
    }
}
```

Gördüğünüz gibi yapıcısı metodların normal metodlardan çok fazla farkı yoktur. Geri dönüş değeri kavramı olmadığı için return ile herhangi bir değer döndürülmedi. Ayrıca yapıcısı metodun parametreleri ile gelen değişkenler ilgili üye elemanlarına atandı. Şimdi de bu yapıcısı metodun otomatik olarak nasıl çalıştığını bakalım. Yapıcısı metodda değişkenlere değerler aktarılırken kontrol yapmak zorundayız. Örneğin Dakika değeri 60'dan büyüğe Saat değişkenini 1 artırmalıyız; aynı şekilde Saniye değeri de 60'dan büyüğe Dakika değerini 1 artırmalıyız. Bu yüzden yapıcısı metod yeniden yazılacaktır. Üye elemanları ekrana yazdırma için bir de Yaz() metodunu olacaktır. Zaman sınıfının yeni versiyonu aşağıdaki gibidir.

```
using System;
class Zaman
{
```

```
public int Saat;
public int Dakika;
public int Saniye;

public Zaman(int h, int m, int s)
{
    Saniye = s % 60;
    int yeniDakika = m + s / 60;
    Dakika = yeniDakika % 60;
    Saat = h + yeniDakika / 60;
}

public void Yaz()
{
    Console.WriteLine("Saat : {0}", Saat);
    Console.WriteLine("Dakika: {0}", Dakika);
    Console.WriteLine("Saniye: {0}", Saniye);
}

class Yapıcılar
{
    static void Main()
    {
        Zaman z = new Zaman(5, 59, 60);
        z.Yaz();
    }
}
```

Programı derleyip çalıştırduğumuzda aşağıdaki ekran görüntüsünü alırız.

```
D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum8\kod>Zaman
Saat : 5
Dakika: 0
Saniye: 60
H:\cskitap\kitap\Bolum8\kod>
```

Zaman z = new Zaman(5, 59, 60);

satırı ile otomatik olarak

Zaman(int h, int m, int s)

yapıcı metodu çağrılmaktadır. Bu sayede 5, 59 ve 60 değerleri metod içinde işlenerek z nesnesinin elemanları değiştiriliyor.

Yapıcı metodlar dışarıdan çağrıldığı için public olarak tanımlanmalıdır. Eğer yapıcısı metodlar private olarak tanımlanırsa sadece sınıfın üye elemanları tarafından erişilir, bu durumda aşağıdaki kullanım geçersiz olur:

Zaman z = new Zaman(5, 59, 60);

## Varsayılan Yapıçı Metot (Default Constructor)

Varsayılan yapıçı metot sınıf nesneleri oluşturulduğu anda biz herhangi bir yapıçı metot tanımlamamış olmamıza rağmen otomatik olarak çağrırlılar. Yani her sınıf nesnesinin en az bir yapıçı metodu vardır. Varsayılan bu yapıcının herhangi bir parametresi yoktur. Varsayılan bu metot nesnelerin üye elemanlarına varsayılan değerler atar; örneğin aşağıdaki programda z nesnesinin Saat, Dakika ve Saniye özelliklerine varsayılan yapıçı metot yardımıyla 0 değerleri atanmıştır.

```
using System;
class Zaman
{
    public int Saat;
    public int Dakika;
    public int Saniye;

    public void Yaz()
    {
        Console.WriteLine("Saat : { 0 }",Saat);
        Console.WriteLine("Dakika: { 0 }",Dakika);
        Console.WriteLine("Saniye: { 0 }",Saniye);
    }
}

class Yapıclar
{
    static void Main()
    {
        Zaman z = new Zaman();
        z.Yaz();

        Console.Read();
    }
}
```

Bu programı derleyip çalıştırıldığımızda ekrana alt alta üç tane 0 yazdığını görürsünüz.

Varsayılan yapıçı metodların herhangi bir parametre almadığını dikkat edin. Bu yüzden z nesnesi aşağıdaki gibi tanımlanmıştır.

```
Zaman z = new Zaman();
```

Nesneleri bu şekilde tanımlamak zorunda olduğumuz için bütün nesneler için varsayılan yapıçı metot çalıştırılmaktadır.

Yapıcı metodları biz kendi sınıflarımız içinden tekrar bildirebiliriz. Bu durumda varsayılan yapıçı metot artık çalıştırılmaz. Örneğin aşağıdaki programda varsayılan yapıçı imzası ile aynı olan bir metot bildirilerek üye elemanlara 0'dan başka değerler atanmıştır.

```
using System;
class Zaman
{
    public int Saat;
    public int Dakika;
    public int Saniye;

    public Zaman()
    {
        Saat = 10;
        Dakika = 10;
        Saniye = 10;
    }

    public void Yaz()
    {
        Console.WriteLine("Saat : { 0 }",Saat);
        Console.WriteLine("Dakika: { 0 }",Dakika);
        Console.WriteLine("Saniye: { 0 }",Saniye);
    }
}

class Yapıclar
{
    static void Main()
    {
        Zaman z = new Zaman();
        z.Yaz();

        Console.Read();
    }
}
```

Yapıcı metodlar da aşırı yüklenebilir. Örneğin aşağıdaki yapıçı metodları tanımlamak mümkün değildir. Metotlardan hangisinin seçileceğine karar verme ise doğal olarak sınıfın imzaları ile ilgilidir.

```
public Zaman(int h,int m,int s)
public Zaman(int h,int m)
public Zaman(int h)
```

Bu yapıçı metodlardan birincisi ile diğer metodları **this** anahtar sözcüğünü kullanarak oluşturabiliriz. **this** anahtar sözcüğünün bu şekildeki kullanımı aşağıdaki Zaman sınıfında gösterilmiştir.

```
class Zaman
{
    public int Saat;
    public int Dakika;
```

```

public int Saniye;

public Zaman(int h,int m,int s)
{
    Saat = h;
    Dakika = m;
    Saniye = s;
}

public Zaman(int h,int m):this(h,m,0)
{
}

public Zaman(int h):this(h,0,0)
{
}

public void Yaz()
{
    Console.WriteLine("Saat : { 0 }",Saat);
    Console.WriteLine("Dakika: { 0 }",Dakika);
    Console.WriteLine("Saniye: { 0 }",Saniye);
}

```

Bu örnekteki son iki yapıçı metodun birinci metodunu kullandığına dikkat edin.

## Kopyalayıcı Yapıçı Metot (Copy Constructor)

Bir Zaman nesnesini yine bir Zaman nesnesini kullanarak oluşturmak isteyebiliriz. Bu durumda Kopyalama yapıçı metotlarını kullanabiliriz. Bu metodların tek bir parametresi vardır. Bu parametreler de sınıf türünden nesnelerdir. Örnek bir kopyalama yapıçı metodu aşağıdaki şekilde oluşturulabilir.

```

public Zaman(Zaman yeni)
{
    Saat = yeni.Saat;
    Dakika = yeni.Dakika;
    Saniye = yeni.Saniye;
}

```

Bu yapıçı metodu ise aşağıdaki tanımlama ile çağırılır.

```

Zaman z1 = new Zaman(5,59,68);
Zaman z2 = new Zaman(z1);

```

C++ dilinde bir sınıf nesnesini başka bir sınıf nesnesini aktarırken özel bir kopyalayıcı metodu çağrılmaktadır. Ancak C#'ta iki sınıf nesnesi birbirlerine aktarılırken sadece referanslar aktarılır. Yani nesnelerin heap bellek alanındaki değerleri kopyalanmaz. Tipik olarak bu işlem atama operatörünü kullandığımızda karşımıza çıkar.

Aşağıdaki örneği dikkatle inceleyin.

```

using System;
//Zaman sınıfı burada olacak.

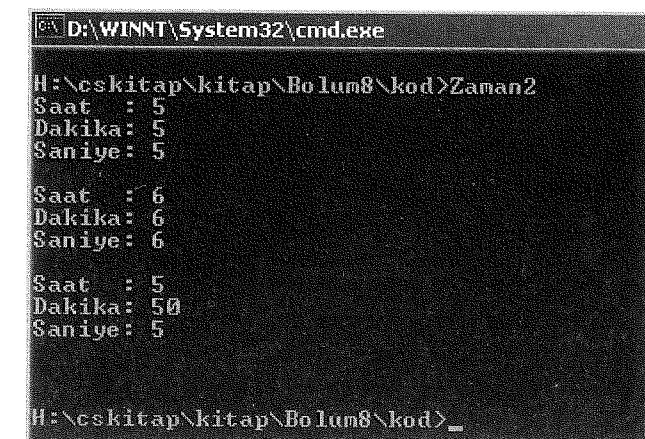
class Yapicilar
{
    static void Main()
    {
        Zaman z1 = new Zaman(5,5,5);
        z1.Yaz();
        Zaman z2 = new Zaman(6,6,6);
        z2.Yaz();

        z2=z1;

        z2.Dakika = 50;
        z1.Yaz();
        Console.Read();
    }
}

```

Yukarıdaki programın önce çıktısına bakalım (Programı derlemeden önce Zaman sınıfının ilgili yere eklemeyi unutmayın).



İlk başta z1 ve z2 nesneleri birbirlerinden bağımsız olarak oluşturulup ekrana değerleri yazdırılıyor. Ancak

```

z2=z1;

```

deyminden sonra z2 nesnesinde yaptığımız değişikliklerin aynısı z1 nesnesinde de olmuştur. Bu yüzden z2 nesnesinin üye elemanlarına bu satırından sonra erişemiyoruz.

# Yıkıcı Metotlar (Destructors) ve Dispose()

## Metodu

Nesnelere erişimin mümkün olmadığı yerlerde nesnelerin artık bellekte kalmasının bir anlamı yoktur. Bu yüzden C++ ile programlama yapan kişilerin en çok dert verdiği durumlardan biri olan ayrılmış belleği iade etmek gerekektir. C++ dilinde bir sınıfın içinde bir nesne için dinamik olarak bellek alanı tahsis edilmişse nesnenin ömrü dolduğunda bu alanın tekrar geri iade edilmesi gereklidir, zira heap bellek bölgesinden tahsis edilen alanlar geri iade edilmezse bellek sizıntıları (**memory leak**) olabilir. C++ programcılar bu iade işlemini kendileri yapmaktadır, ancak C# programlarının bu zahmeti çekmesine gerek yok. Gereksiz Nesne Toplama (**Garbage Collection**) dediğimiz mekanizma gereksiz nesnelerin tuttuğu referans bölgelerini zamanı ve yeri gelince iade eder. C++’ta bir nesnenin faaliyet alanı bittiğinde otomatik olarak nesneler yok edilir. Ancak dinamik olarak ayrılmış alanlar yok edilmez. Bu yüzden nesnelerin yok edilmelerinden hemen önce çalışan metotlar yazılır. Bu metotlara yıkıcı metotlar (**destructors**) denilmektedir. C#’ta bu işlemler tamamen farklıdır; gereksiz nesne toplayıcısı (**Garbage Collector**) bizim yerimize gereksiz alanları iade eder, ancak bu işlemin nesnelerin faaliyet alanlarının bittiği noktada olacağı garanti altına alınmamıştır. Bu yüzden bir nesnenin kapladığı bellek alanlarının ne zaman iade edileceği kesin olarak bilinemez. Fakat C#’ta bildirilen yıkıcı metotlarının bu iade işleminden hemen önce çalıştırılacağı kesindir.

C# dilinde bir nesnenin kaynakları iki şekilde iade edilebilir. Birincisi **Dispose()** metodunu kullanmak. **Dispose()** metoduna şimdilik değinmeyeceğiz. Çünkü **Dispose()** metodunu işlemek için **IDisposable** arayüzü bilmemiz gereklidir. İkinci yöntem ise yıkıcı metotlar bildirmektir.

Yıkıcı metotlar C++ dilindeki gibi bildirilir. Yıkıcı metotların isimleri de yapıçı metotlar gibi sınıf ile aynı olmalıdır. İsimlerinin başına ~ karakteri gelir. Yıkıcı metotların herhangi bir parametresi veya geri dönüş değeri yoktur. Aynı zamanda public ya da private gibi erişim belirteçleri ile işaretlenmezler. Bu bilgiler ışığında bir sınıfın yıkıcı metodu aşağıdaki gibi bildirilebilir.

```
class Dest
{
    ~Dest()
    {
        Console.WriteLine("Yıkıcı metot çağrıldı");
    }
}
```

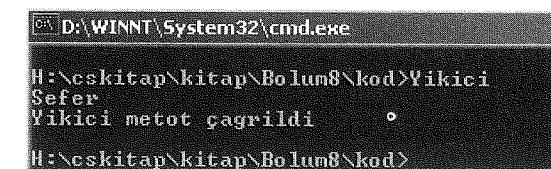
Şimdi de aşağıdaki programı çalıştırıp yıkıcı metodun ne zaman çağrıdığını görelim.

```
using System;
class Dest
{
```

```
~Dest()
{
    Console.WriteLine("Yıkıcı metot çağrıldı");
}

class AnaSınıf
{
    static void Main()
    {
        Dest d = new Dest();
        Console.WriteLine("Sefer");
    }
}
```

Önce programın çıktısına bakalım:



Eğer yukarıdaki program C++ ile yazılmış olsaydı ekran görüntüsü böyle olmazdı. Önce yıkıcı metot çağrıldı sonra ekrana "Sefer" yazılırdı.

```
d nesnesinin tanımlandığı
{
    Dest d = new Dest();
}
```

Bloğunun sonunda d nesnesine erişmek mümkün değildir. Ancak bloğun dışına gelindiğinde ilk yapılan yıkıcı metodu çağrılmamıştır. Önce ekrana "Sefer" yazıldı sonra gereksiz nesne toplama (**garbage collection**) mekanizması devreye girdi. Tabi nesne bellekten silinmeden önce yıkıcı metodumuzun çağrılabileceği garanti olduğu için ekrana "Yıkıcı metot çağrıldı" yazılmıştır.

Gereksiz nesne toplayıcısı (**garbage collector**) bizim isteğimiz dışında çalışmaktadır. Ancak .NET sınıf kütüphanesindeki bazı sınıflar (System.GC gibi) yardımıyla istediğimiz anda GC (**garbage collection**) mekanizmasını devreye sokabiliriz. Bu durum pek tavsiye edilmediği için bu kitapta bu konuya yer verilmeyecektir.

C#’ta genellikle yıkıcı metotlar nesnenin kapladığı alanı geri iade etmek için kullanılmaz. Daha çok sınıfın global düzeyde olan üye değişkenleri ile ilgili işlemler yapmak için kullanılır. Örneğin bir oyun programı düşünün. Oyunun belli bir noktasında oyuncı

daki oyuncu sayısını tutacak bir değişken olsun. Her oyuncu yaratıldığında bu değişken 1 artırılır, her oyuncunun işlevi bittiğinde ise yıkıcı metot yardımıyla oyuncu sayısı bir azaltılır. Statik üye elemanlarını anlatırken bu kullanım şekline bir örnek verilecektir.



Bir sınıfın sadece bir tane yıkıcı metodu olabilir.

## Statik Üye Elemanlar

C# dilindeki bütün metotlara sınıflar üzerinden erişiriz. Çoğu durumda da erişim için bu sınıflardan nesneler oluştururuz. Ancak bazı durumlarda metotları kullanmak için nesne oluşturmamız gereksiz olabilir. Bu durumda statik metotlar tanımlanır. Statik metotlar olabileceği gibi statik üye değişkenler ve statik yapıçı metotlar da olabilir. Bir üye elemanın statik olduğunu bildirmek için bildirimden önce **static** anahtar sözcüğü eklenir.

Statik elemanlar bir sınıfın global düzeydeki elemanlarıdır. Yani statik üyeleri kullanmak için herhangi bir nesne tanımlamamıza gerek yoktur. Şimdi sırasıyla statik üye elemanları inceleyelim.

### Statik Metotlar

Metotlar konusunda Math sınıfının çeşitli metotlarını görmüştük. Bu metotları herhangi bir nesne oluşturmadan kullanabildik. Örneğin bir sayının karekökünü almak için `Math.Sqrt(sayı);`

ifadesini kullanmamız yeterlidir. Karekök alma işlemi genel bir işlem olduğundan bu tür işlemleri yapmak için bir nesne tanımlamak gerçekten çok saçma olurdu. Neyse ki C# dilini tasarlayan insanlar bunu düşünmüşler. Statik metotlara sınıfın adı ile ulaşılır. Örneğin aşağıdaki **Topla()** metodu statik olarak tanımlanmış ve Main() metodu içinden çağrılmıştır.

```
using System;
class Cebir
{
    public static int Topla(params int[] dizi)
    {
        int toplam = 0;

        for(int i = 0 ; i < dizi.Length; ++i)
            toplam += dizi[i];

        return toplam;
    }
}
```

```
class AnaSınıf
{
    static void Main()
    {
        int i;
        i=Cebir.Topla(5, 6, 8);
        Console.WriteLine(i);
    }
}
```

Gördüğünüz gibi Cebir sınıfından herhangi bir nesne tanımlanmadan **Topla()** metoduna `Cebir.Topla(5, 6, 8);` şeklinde ulaşılmıştır. (**Topla** metodunun değişken sayıda parametre aldığına dikkat edin)

Peki, bir statik metoda nesne üzerinden erişmemiz mümkün mü? Hayır, mümkün değil. Örneğin, aşağıdaki c nesnesi üzerinden **Topla()** metoduna erişmeye çalışmak hatalıdır.

```
Cebir c = new Cebir();
c.Topla(5, 4);
```

Sınıflarımızı tasarlarken nesneler ile doğrudan iş yapmayan metotları statik olarak bildirmemiz gereklidir.

Şu ana kadar yaptığımız bütün örneklerde Main() metodunun statik olduğunu dikkat etmişsinizdir. Bunun sebebi Main() metodunun çalışması için herhangi bir sınıf nesnesine ihtiyaç duymadan çalışmasını sağlamaktır.

**static** anahtar sözcüğünü kullanırken erişim belirleyicisinden önce ya da sonra koymamız bir fark yaratmaz, örneğin aşağıdaki her iki bildirim de aynı anlama gelmektedir.

```
static public int Topla(params int[] dizi)
public static int Topla(params int[] dizi)
```

Bir statik metot içinden sınıfın diğer statik metotları çağrılabılır. Ancak normal bir üye metot çağrılamaz. Çünkü normal metotlar (statik olmayan) nesneler üzerinde işlem yaparlar. Dolayısıyla nesnelerin adresleri gizlice metoda **this** referansı ile gönderilir. Ancak statik metotlar sınıfın global metotları olduğu için **this** referansları yoktur. Bu yüzden statik bir metot içinden normal bir metot çağrılmak geçersizdir. Aşağıdaki programda statik metot içinden diğer bir statik metot çağrılmıştır.

```
class Statik
{
    public static void Metot1()
    {
        Console.WriteLine("Metot1");
    }
}
```



```

public static void Metot2()
{
    Metot1();
    Console.WriteLine("Metot2");
}
}

```

bu bildirim geçerli iken aşağıdaki sınıf bildirimi geçersizdir.

```

class Statik
{
    public void Metot1()
    {
        Console.WriteLine("Metot1");
    }

    public static void Metot2()
    {
        Metot1();
        Console.WriteLine("Metot2");
    }
}

```

Gördüğü gibi statik olmayan elemanlara ulaşabilmek için bir nesne olmalıdır. Eğer bir nesne var ise statik olmayan elemanlara ulaşılabilir.

## Statik Değişkenler

Statik değişkenler de statik metodlar gibi bir sınıf türünü ilgilendiren değişkenlerdir. Herhangi bir nesne ile statik değişkenlere ulaşmamız mümkün değildir. Statik değişkenlere ancak sınıfın adı ile ulaşabiliyoruz.

Statik değişkenler genellikle bir sınıfı global düzeyde ilgilendiren özellikler için kullanılır. Mesela bir oyun programında çalışma zamanının herhangi bir noktasında oyuncu sayısını tutan bir değişken statik olmalıdır.

Statik değişkenleri bildirmek için de **static** anahtar sözcüğü kullanılır. Bu anahtar sözcük, erişim belirleyicisinden önce de sonra da kullanılabilir. Örnek bir statik değişken bildirimi aşağıdaki gibi yapılabilir.

```
public static int Toplam;
```

ya da

```
static public int Toplam;
```

Statik değişkenler tanımlandıklarında varsayılan değere atanırlar. Örneğin aşağıdaki programda statik değişkenleri ekrana yazdırduğumda ekrana 0 ve false yazacaktır.

```
using System;
```

```

class Statik
{
    public static int a;
    public static bool b;
}

class AnaSınıf
{
    static void Main()
    {
        Console.WriteLine(Statik.a);
        Console.WriteLine(Statik.b);
    }
}

```

Statik üyelerle ancak statik bir metot içerisinde erişebiliriz. Örneğin aşağıdaki sınıf bildirimi geçersizdir.

```

class Statik
{
    public int d;

    static public void den()
    {
        d = 5;
    }
}

```

Bu sınıfı içeren bir program çalıştırıldığında şu hata verilir :

**"An object reference is required for the nonstatic field, method, or property 'Statik.d'"**

yani

"Statik sınıfındaki statik olmayan alanlar, metodlar ya da Statik.d özelliğine erişmek için bir nesne referansı gerekmektedir."

Her sınıf nesnesi oluşturulduğunda her statik üye değişken için bellekte ayrı yer tahsis edilmez.



## Statik Yapıçı Metotlar

Normal metodlar gibi yapıclar da statik olabilir. Statik yapıçı metodlar bir sınıfın statik değişkenleri ile ilgili işlemler yapmadır. Bir nesne ilk defa yaratıldığında statik üye değişkenini değiştirmek için genellikle statik yapıçı metodlar tanımlanır. Statik olan bir metod ile statik olmayan bir değişkeni değiştirmek nasıl mümkün değilse statik olan yapıclarda statik olmayan değişkenleri değiştiremez.

Statik metodların bildirilmesi normal metodların bildirimi ile aynıdır. Sadece bildirimin başına **static** anahtar sözcüğü eklenir.

Şimdi statik bir yapıçı metoda örnek verelim: Bu programda Oyuncu sınıfının iki tane yapıçı metodu var; bu metodlardan birisi statik, diğeri ise değil. Bir nesne oluşturulduğunda hangi yapıcıların çalıştığını görebilmek için programı derleyip çalıştırın.

```
using System;
class Oyuncu
{
    public Oyuncu()
    {
        Console.WriteLine("Statik olmayan yapıçı");
    }
    static Oyuncu()
    {
        Console.WriteLine("Statik yapıçı");
    }
}

class AnaSınıf
{
    static void Main()
    {
        Oyuncu o = new Oyuncu();
        Oyuncu x = new Oyuncu();
    }
}
```

Bu programın önce ekran çıktısına bakalım:

```
H:\cskitap\kitap\Bolum8\kod>Statik
Statik yapıcı
Statik olmayan yapıçı
Statik olmayan yapıçı
H:\cskitap\kitap\Bolum8\kod>
```

Gördüğü gibi o nesnesi oluşturulduğunda hem statik yapıçı metot hem de statik olmayan metot çağrılmıştır (önce statik yapıçı metot çağrılmış). Ancak x nesnesi oluşturulduğunda sadece statik olmayan yapıçı metot çağrılmıştır. Bir nesnenin varlığını kullanarak çağrıabileceğimiz tek metot statik yapıçı metottur.

Statik yapıçı metodlar herhangi bir parametre alamazlar. Yani parametrelili bir statik yapıçı metot tanımlanamaz. Statik yapıçı metodların erişim belirleyicileri de yoktur.

 Bir nesneyi hangi yapıçı metot ile oluşturursak oluşturalım statik yapıçı metot mutlaka ilk nesne tanımlandığında çalışırır.

Statik yapıçı metodlar ile genellikle statik değişkenler ile ilgili işlemler yapılır. Ama bu zorunlu değildir. Örneğin aşağıdaki kaynak kodda her Oyuncu nesnesi oluşturulduğunda statik Toplam değişkeni 1 artırılıyor ve her Oyuncu değişkeni işlevini bitirdiğinde yıkıcı işlev yardımıyla Toplam değişkeni 1 azaltılıyor.

```
using System;
class Oyuncu
{
    public static int Toplam;

    public Oyuncu()
    {
        Oyuncu.Toplam++;
    }

    static Oyuncu()
    {
        Toplam = 0;
    }

    ~Oyuncu()
    {
        Console.WriteLine("Bir oyuncu gitti...");
        Toplam--;
    }
}

class AnaSınıf
{
    static void Main()
    {
        Oyuncu o = new Oyuncu();
        Console.WriteLine("Toplam oyuncu=" + Oyuncu.Toplam);
        Oyuncu x = new Oyuncu();
        Console.WriteLine("Toplam oyuncu=" + Oyuncu.Toplam);
    }
}
```

Bu programın ekran görüntüsü aşağıdaki gibidir.

```
H:\cskitap\kitap\Bolum8\kod>Statik2
Toplam oyuncu=1
Toplam oyuncu=2
Bir oyuncu gitti...
Bir oyuncu gitti...
H:\cskitap\kitap\Bolum8\kod>
```

## Static Sınıflar

C# 2.0 ile birlikte C# diline static sınıflar kavramı gelmiştir. Aslında statik sınıflar dile eklenmiş artı bir özellikten ziyade yapmak istemediklerimizi zorlayıcı bir etken olarak sunmak için vardır. Gerçek hayatı birçok projede bazı sınıfların içindeki bütün üye elemanlar static olarak tanımlanabiliyor. Bu tür durumlara genellikle birbirinden bağımsız bir şekilde iş yapan metodların tanımlandığı sınıflarda karşılaşmaktadır. Örneğin XML tabanlı bir dosyadan ya da metin tabanlı bir dosyadan uygulama ayarlarının verilen bir anahtara göre yüklenmesi işlemlerini yapan metod toplulukları genelde static tanımlanır. Bu durumda o sınıf türünden nesne yaratmak, nesne referansı tanımlama ya da o sınıfı kalıtım yolu ile (kalıtım sonrası konularda detaylı olarak ele alınacaktır) genişletmek genelde rastlanan birşey değildir. Bu yüzden C# dilini tasarılayanlar bu tür durumlarda sınıf ile yapılabilecekleri kısıtlamak (buyle performans sağlamak) ve sadece belirlenen özelliklerde (static) üye elemanlarının tanımlanabileceği static sınıf kavramını geliştirmiştir.



Önemli Not: Statik metodlar C# 2.0'dan itibaren dile eklenmiştir.

İçinde sadece statik üye elemanları barındıran sınıflar static sınıf olarak tanımlanmalıdır. Bu bir zorunluluk değildir ancak iyi bir kodlama teknigidir. Bir sınıf static olarak bildirildiği andan itibaren o sınıf türünden bir nesne yaratılamaz. Örneğin aşağıdaki örnekte olduğu gibi **MatematikselIslemler** isimli sınıfın içinde static olmayan hiçbir üye eleman yoktur. (const elemanların otomatik olarak static olduğunu hatırlatalım) Bu yüzden MatematikselIslemler isimli sınıf kullanarak bir nesne yaratlığımızda o nesne üzerinden anlamlı bir metod ya da özellik çağrıramayacağımız açıktır. Bu durumda nesne yaratmamız anlamsız hale geliyor. Biz de iyi programcılar olarak bu sınıfı static yapıp nesne yaratılmasını engelliyoruz.

```
using System;
static class MatematikselIslemler
{
    public static int Topla(int a,int b)
    {
        return a+b;
    }

    public static int Carp(int a,int b)
    {
        return a*b;
    }

    public const double PI = 3.14;
}
```

```
class Program
{
    static void Main(string[] args)
    {
        // Sınıf static olduğundan aşağıdaki nesne tanımlaması geçersidir.
        MatematikselIslemler m = new MatematikselIslemler();
    }
}
```

Statik sınıflar içinde static olmayan metod ya da özellik tanımlanamayacağı gibi aynı şekilde static yapıcı metodlar da tanımlanamaz.

Static sınıflar türünden aşağıdaki gibi nesne yaratamayız.

```
MatematikselIslemler m = new MatematikselIslemler();
```

Aynı zamanda static sınıf türünden aşağıdaki gibi referanslar tanımlayamayız.

```
MatematikselIslemler m;
```

Dolayısıyla bir metodun parametreleri ya da geri dönüş değeri kesinlikle static olan bir sınıf türünden olamaz.

Sonraki Konular için Not: İleride göreceğimiz türetme kavramında da görebeksiniz ki static sınıflar aynı zamanda türetmeyi de desteklemez. Bir static sınıfın başka bir sınıf türetilemeyeceği gibi bir static sınıfın başka bir static sınıf da türetilemez.

Göründüğü gibi static sınıflar dile eklenmiş kısıtlayıcı bir etkendir. Kod okunabilirliği açısından da önemli bir bildirim yöntemidir. Programcı herhangi bir sınıfın static olduğunu gördüğü anda içindeki bütün üye elemanlarının static olduğunu anlayacaktır.

Birçoğunuza aklına gelecek bir soru: Madem, static olarak bildirilen bir sınıf sadece static elemanlar içerebiliyor o halde neden üye elemanları tek tek static olarak bildirip gereksiz kod kalabahlığı yapıyoruz? Evet, bu konuda çeşitli tartışmaları internette bulabilirsiniz, bu bir tasarım kararı olduğundan söylenecek çok fazla bir şey yok. Ancak bir sonraki dil yeniliğinde bu özelliği görmemiz içten bile değil.

## using Deyimi ile Statik üyelerde Direkt Erişim

C# 6.0 ile birlikte yine kod yazım kolaylığı sağlayan ancak bana göre okunabilirliği azaltan bir özellik gelmiştir. Bu özelliğe göre artık using deyiği ile ilgili sınıf kod bloğunun başında bildirildiğinde o sınıf içerisinde bulunan bütün static üye elemanlara kod içerisinde direkt erişilebilir. Buna göre aşağıdaki kullanım tamamen geçerlidir.

```
using System;
```



```
using System.Console;

namespace deneme
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("console.WriteLine yazmadan erişildi");
        }
    }
}
```

Yani artık bir sınıf içerisindeki üye özellik static ise o üye elemana sınıf ismini yazmadan erişmek mümkünür. Bunun için tek şart ilgili sınıfın isim alanı açısından erişilebilir olmasıdır.



Her ne kadar bu özellik kolaylık gibi gözükse de okunabilirlik açısından kullanılmamayı tavsiye ediyorum. Çünkü uzun kod bloklarını okuduğunuzda ilgili metodun hangi sınıfta olduğunu anlamınız mümkün olmayacağındır.



**Önemli Not :** using deyiği kullanılarak bir sınıfın static üyelerine direkt erişim özelliği yalnızca C# 6.0 (2015 yılında gelen) ile geçerlidir.

## const ve readonly Elemanlar

Değerinin değişmeyeceği düşünülen bir eleman sabit olarak bildirilebilir. Sabit değişkenler const anahtar sözcüğü ile bildirilir. Örneğin Math sınıfının E ve PI üye değişkenleri const olarak bildirilmiştir. Const olarak bildirilen değişkenlere mutlaka ilk değer verilmelidir. İlk değer verilmemiş olan sabit değişkenleri bildirmek hatalıdır. Bir sabit değişken aşağıdaki gibi bildirilir:

```
public const double PI=3.14;
```

Sabit ifadelerin değerleri derleme aşamasında mutlaka hesaplanabilmelidir. Buna göre aşağıdaki sabit bildirimi geçerlidir;

```
int x = 5;
const int y = x + 2;
```

Aşağıdaki const ifadesi ise derleme aşamasında belli olmadığı için geçersizdir.

```
int x = Convert.ToInt32(Console.ReadLine());
const int y =x;
```

y=x; ifadesinin geçersiz olmasının sebebi x değişkeninin çalışma zamanında belirlenmesidir.

Sabit değişkenleri değiştirmeye çalışmak derleme zamanında hataya yol açar. Örneğin aşağıdaki kullanım geçersizdir:

```
const int y = 5;
y = 6;
```

Statik üyelerde olduğu gibi sabit ifadeler de bir sınıfın global düzeydeki elemanlardır. Yani bir nesne üzerinden const ifadelerine erişilemez. Statik değişkenlerde olduğu gibi sabit değişkenlere de sınıfın ismiyle erişilebilir. Aşağıdaki programda bu durumlar belirtilmiştir.

```
using System;
class Cebir
{
    public const double PI=3.14;
}

class Const
{
    static void Main()
    {
        //Geçerli
        Console.WriteLine(Cebir.PI);

        Cebir c = new Cebir();
        //Geçersiz
        Console.WriteLine(c.PI);
    }
}
```

Sabit ifadeleri referans tipinden olamaz. Çünkü referans tiplerinin değerleri çalışma zamanında ayarlanır. Örneğin aşağıdaki kullanım geçersizdir,

```
class Sınıf
{
}

class Cebir
{
    const Sınıf a = new Sınıf();
}
```

Ancak bu duruma bir istisna vardır. String türü referans türü olmasına rağmen eğer string türünden bir nesneyi new anahtar sözcüğünü kullanmadan tanımlarsak değişkeni const olarak tanımlayabiliriz. Aşağıdaki tanımlama tamamen geçerlidir.

```
public const string a = "Sefer Algın";
```

Referans tipleri için sabitleri tanımlamada - C++ dilinde olmayan - **readonly** anahtar sözcüğü kullanılır. Ancak **readonly** (sadece okunur) olarak tanımlanan değişkenler global düzeyde olacak diye bir şey yoktur. Bütün elemanları **readonly** olarak düzenleyebiliriz.

statik elemanları **readonly** olarak tanımladığımızda **const** anahtar sözcüğünün referans tipleri için bir versiyonunu yapmış oluruz. Örneğin,

```
using System;
class Cebir
{
    static readonly Deneme a = new Deneme();
}
```

```
class Deneme
{ }
```

kullanımı tamamen geçerlidir. Üstelik statik eleman olduğu için a değişkeni global düzeydedir.

**readonly** olarak tanımlanan değişkenler adından da anlaşıldığı üzere sadece okunabilir değişkenlerdir. Bu nedenle, çalışma zamanında ya da derleme aşamasında değerleri değiştirilemez.

 static **readonly** ve **const** olarak tanımlanmış değişkenlerin tek farkı ilk değer verilme zamanlarıdır. **const** değişkenlere derleme zamanında **static readonly** değişkenlere ise çalışma zamanında ilk değer verilir.

## Singleton (Tek) Nesneler

Şu ana kadar öğrendiklerimizi pekiştirmek amacıyla gerçek projelerde karşımıza çıktı kabilecek örnek bir problemi çözmeye çalışalım.

**Singleton** deseni bir programın yaşam süresince belirli bir nesneden sadece bir örneğinin (instance) olmasını garantiyor. Aynı zamanda bu desen, yaratılan tek nesneye ilgili sınıfın dışından global düzeyde mutlaka erişilmesini hedefler. Örneğin bir veritabanı uygulaması geliştirdiğinizi düşünelim. Her programcı mutlaka belli bir anda sadece bir bağlantı nesnesinin olmasını isteyecektir. Böylece her gerekiğinde yeni bir bağlantı nesnesi yaratmaktansa varolan bağlantı nesnesi kullanılarak sistem kaynaklarının daha efektif bir şekilde harcanması sağlanır. Bu örnekleri daha da artırmak mümkündür. Siz ne zaman belli bir anda ilgili sınıfın sadece bir örneğine ihtiyaç duyarsanız bu deseni kullanabilirsiniz.

Peki, bu işlemi nasıl yapacağız? Nasıl olacak da bir sınıfın sadece bir nesne yaratılması garanti altına alınacak? Aslında biraz düşünürseniz cevabını hemen bulabilirsiniz! Çözüm gerçekten de basit: statik üye elemanlarını kullanarak.

Singleton tasarım desenine geçmeden önce sınıflar ve nesneler ile ilgili temel bilgilimizi hatırlayalım. Hatırlayacağınız üzere bir sınıfın yeni bir nesne oluşturmak için **yapıcı metot (constructor)** kullanılır. Yapıçı metotlar C# dilinde **new** anahtar sözcüğü kullanılarak aşağıdaki gibi çağrılabilmektedir.

```
Sınıf nesne = new Sınıf();
```

Bu şekilde yeni bir nesne oluşturmak için **new** anahtar sözcüğünün temsil ettiği yapıçı metoduna dışarıdan erişimin olması gereklidir. Yani yapıçı metodun **public** olarak bildirilmiş olması gereklidir. Ancak Singleton desenine göre belirli bir anda sadece bir nesne olabileceği için **new** anahtar sözcüğünün ilgili sınıf için yasaklanması gereklidir yani yapıçı metodun **protected** ya da **private** olarak bildirilmesi gereklidir. Eğer bir metodun varsayılan **yapıcı metodu (default constructor- parametresiz yapıçı metot)** **public** olarak bildirilmemişse ilgili sınıf türünden herhangi bir nesnenin sınıfın dışında tanımlanması mümkün değildir. Ancak bizim isteğimiz yalnızca bir nesnenin yaratılması olduğuna göre ilgili sınıfın içinde bir yerde nesnenin oluşturulması gereklidir. Bunu elbette statik bir **özellik (property)** ya da statik bir metodla yapacağız. Bu statik metod sınıfın kendi içinde yaratılan nesneyi geri dönüş değeri olarak bize gönderecektir. Peki, bu nesne nerede ve ne zaman yaratılacaktır? Bu nesne statik metodun ya da özelliğin içinde yaratılıp yine sınıfın private olan elemanına atanır. Tekil olarak yaratılan bu nesne her istendiğinde eğer nesne zaten yaratılmışsa bu private olan elemanın referansına geri dönmek ya da nesneyi yaratıp bu private değişkene atamak gerekmektedir. Sanırım bu deseni nasıl uygulayabileceğimizi kafanızda biraz canlandırdınız. O halde daha fazla uzatmadan desenimizi uygulamaya geçirelim.

```
public class SingletonNesne
{
    private static SingletonNesne nesne = new SingletonNesne();

    private SingletonNesne ()
    {
    }

    public static SingletonNesne Nesne
    {
        get
        {
            return nesne;
        }
    }
}
```

Yukarıdaki sınıf bildiriminden sonra **SingletonNesne** örneği aşağıdaki gibi elde edilebilir.

```
SingletonNesne nesne = SingletonNesne.Nesne;
```

Böylece yukarıdaki deyimi her kullandığınızda size geri dönen nesne, sınıfın belleğe ilk yüklenliğinde yaratılan nesne olduğu garanti altına alınmış oldu. Dikkat etmeniz gereken diğer bir nokta ise nesneyi geri döndüren özelliğin yalnızca `get` bloğunun olmasıdır. Böylece bir kez yaratılan nesne harici bir kaynak tarafından hiçbir şekilde değiştirilemeyecektir.

Yukarıdaki her `SingletonNesne` versiyonunda da biz yaratılan nesneyi

```
SingletonDeseni nesne = SingletonDeseni.Nesne;
```

şeklinde istediğimizde nesne zaten yaratılmış durumda olmaktadır. Oysa bu sınıfı daha efektif bir hale getirerek yaratılacak nesnenin ancak biz onu istediğimizde yaratılmasını sağlayabiliriz. Bu durumu uygulayan Singleton desenini aşağıdaki gibi yazabiliriz.

```
public class SingletonNesne
{
    private static SingletonNesne nesne;

    private SingletonNesne()
    {

    }

    public static SingletonNesne Nesne()
    {
        if (nesne == null)
            nesne = new SingletonNesne();

        return nesne;
    }
}
```

Gördüğünüz üzere nesne ilk olarak sınıf belleğe yüklenliğinde değil de o nesneyi ilk defa kullanmak istediğimizde yaratılıyor. İlgili nesneyi her istediğimizde yeni bir nesnenin yaratılmaması için de

```
if(nesne == null)
```

şeklinde bir koşul altında nesnenin yaratıldığına dikkat edin.

Her şeye rağmen yukarıdaki 2 versiyonda da bazı durumlar için tek bir nesnenin olmasını garanti etmemiş olabiliriz. Eğer **çok kanallı** (*multi-thread*) bir uygulama geliştiriyoysanız farklı kanalların aynı nesneyi tekrar yaratması olasıdır. Ancak eğer çok kanallı çalışmıyorsanız (*çokunlukla tek thread ile çalışırız*) yukarıdaki sade ama öz olan versiyonlardan birini kullanabilirsiniz. Ama eğer çok kanallı programlama modeli söz konusu ise ne yazık ki farklı kanalların aynı nesneden tekrar yaratmasını engellemek için ekstra kontroller yapmanız gerekmektedir.

Thread güvenli yeni nesne, istediğimiz yapılar kullanılarak, aşağıdaki gibi tasarlabilir.

```
public class SingletonNesne
{
    private static SingletonNesne nesne = new SingletonNesne();
    private static SingletonNesne()
    {
    }

    private SingletonNesne()
    {
    }

    public static SingletonNesne Nesne
    {
        get
        {
            return nesne;
        }
    }
}
```

Yukarıdaki versiyonun diğerlerinden tek farkı yapıcı metodun da statik olmasıdır. C# dilinde statik yapıcı metotlar bir uygulamada ancak ve ancak bir nesne yaratıldığında ya da statik bir üye eleman referans edildiğinde bir defaya mahsus olmak üzere çalıştırılır. Yani yukarıdaki versiyonda **farklı kanalların** (*thread*) birden fazla Singleton nesnesi yaratması imkansızdır. Çünkü static üye elemanlar ancak ve ancak bir defa çalıştırılır.

Multi-Thread uygulama geliştirme konusu bu kitabın kapsamı dışındadır. Multi-Thread uygulamalarla ilgili detaylı bilgiler [www.csharpnedir.com](http://www.csharpnedir.com) sitesindeki makalelerde bulabilirsiniz.



## Operatör Aşırı Yükleme (Operator Overloading)

Nesne yönelimli programlama teknığının en etkili özelliklerinden biri operatörlere sınıflarımız için yeni anlamlar yüklemektir. Sınıflarımız için tasarlayabileceğimiz en etkili üye elemanları belki de operatör metotlarıdır. Operatör metotları bir nesnenin ifadeler içinde operatörlerle kullanıldığı zaman davranışını belirler. Örneğin `int` türünden nesneler için `+` operatörü iki sayının toplamı demektir. Aynı şekilde `*` operatörü de iki sayının çarpımı demektir. Temel veri türleri için operatörler zaten belirlenmiştir. Bu yüzden bunlara müdahale edemeyiz. Ancak kendi tasarladığımız sınıflar için operatör-

lere yeni anlamlar yükleyebiliriz. Örneğin, matematikteki kompleks sayıları hatırlayın. Bir kompleks sayının gerçek ve sanal olmak üzere iki bölümü vardır. C#'ta kompleks sayıları temsil eden bir veri türü bulunmamaktadır. Ancak bunu kendimiz oluşturabiliriz. Kompleks sınıfında bir sınıfı tasarladığımızı düşünelim. İki kompleks sayı nesnesini topladığımızda sonuç yine bir kompleks sayı nesnesi olmalıdır. Ancak derleme sırasında ne yapacağını bilmemektedir.

```
Kompleks a = new Kompleks();
Kompleks b = new Kompleks();
Kompleks c = a + b;
```

+ operatörünü bu şekilde kendi tanımladığımız sınıflarda kullandığımızda, + operatörü ne yapacağını anlamaz. Halbuki bu şekildeki bir kullanım gerçekten gerekli. Çünkü iki kompleks sayının toplanması son derece doğaldır. İşte bu tür kullanımlara müsaade etmek için operatör metotları tanımlanıyor. Örneğin yukarıdaki kullanımımı geçerli kılmak için operator+ metodu bildirilmelidir. Aynı şekilde iki kompleks sayıyı çarpmak için operator\* metodu bildirilmelidir. Aslında operatör metotlarının sonuçlarını daha önceki örneklerimizde kısmen gördük. Örneğin, string sınıfı türünden nesneleri + operatörü ile topladığımızda iki yazı arka arkaya eklenir.

string türünün kullanımına bir göz atalım;

```
string a = "csharp";
string b = "nedir.com";
string c = a + b;
```

Burada yapılan işlemleri anladığınızda operatör yüklemenin çok basit olduğunu göreceksiniz. Derleme işlemi

```
string c = a + b;
```

satırına geldiğinde derleyici, string sınıfının operator+ metodunu arar; eğer operator+ metodu mevcut değilse kod derlenmez. operator+ metodunun bulunması durumunda bir string parametresi olan metot aranır. Bu metot bulunduğuanda a nesnesi için operator+ metodu çağrılır. Yani gizlice aşağıdaki metot çağrıları yapılır:

```
String.operator+(a, b);
```

Yukarıdaki kullanımından da anlaşılacağı gibi operatör metotları statik olmalıdır. Çünkü operator+ metoduna herhangi bir nesne üzerinden değil de sınıfın ismi ile erişilmiştir.

Gördüğünüz gibi operatör metotları ile yapabileceğimiz her şeyi aslında klasik metotlar ile de yapabiliriz. Ancak operatör metotları sınıflarımızı aritmetik işlemler içeren ifadelerde rahatlıkla kullanmamızı sağlıyor.

Operatör metotlarının sonuçlarını daha yakından görmek için matematikteki kompleks sayıları simüle eden bir sınıf tasarlayacağız. Bu sınıfın tasarımını bitirdiğimizde artık eli-

mizde bütün operatörlere cevap veren ve temel işlemleri yapabilen bir sınıf olacak.

Öncelikle kompleks.cs isimli bir dosya açın, bu dosyada sadece Kompleks sınıfının bildirimleri yer alınsın. İkinci bir dosya da program.cs isimli olsun ve Main() metodunun olduğu sınıfı da buraya yazın. Her bir operatör metodunu tanımladığımızda derleme işlemini şu şekilde yapın:

```
csc kompleks.cs program.cs
```

Şimdi kompleks sayı sınıfının ana yapısını belirlemeden önce operator metotları ile ilgili uymamız gereken genel bilgileri maddeler halinde yazalım.

1. Operatör metotları static olarak tanımlanmalıdır.
2. Operator metotlarının isimleri için **operator** anahtar sözcüğü kullanılır. Örneğin operator+, operator\*, operator<= gibi.
3. Bütün operatör metotlarının mutlaka en az bir parametresi olmalıdır. İki parametresi olan operator metotları olabildiği gibi tek parametresi olan operatör metotları da olabilir.
4. Operatör metotları da klasik metotlar gibi aşırı yüklenebilir. Mesela operator+ metodunun birkaç farklı versiyonunu yazabiliriz.
5. Tekil (unary) operatör metotlarında parametre mutlaka ilgili sınıf türünden olmalıdır. Binary (iki operand alan) operatör metotlarında ise en fazla parametre ilgili sınıf türünden olmalıdır.
6. Operator metotları **ref** ve **out** anahtar sözcüklerini kullanmamalıdır.

## Kompleks Sınıfı

Kompleks sınıfının mGerçek ve mSanal isimli iki tane private olan üye değişkeni olacaktır. Kompleks sayıları simüle etmek için bu iki değişken yeterlidir. Bu değişkenler private oldukları için get ve set blokları da tanımlanacaktır. Şimdi kompleks.cs dosyasını açın ve aşağıdaki sınıf bildirimini yapın.

Kompleks sınıfının biri varsayılan yapıcı olmak üzere 3 tane yapıcı işlevi olacaktır. Bu metotların neden olduğunu tahmin edin.

Ayrıca kompleks sayıları zaman zaman ekrana yazdırma için bir de **Yazdir()** metodu bildirilmelidir.

```
using System;
class Kompleks
{
    private double mGercek;
```

```

private double mSanal;

public double Gercek
{
    get{ return mGercek;}
    set{ mGercek = value;}
}

public double Sanal
{
    get{ return mSanal;}
    set{ mSanal = value;}
}

public Kompleks(double a, double b)
{
    mGercek = a;
    mSanal = b;
}

public Kompleks()
{
    mGercek = 0;
    mSanal = 0;
}

public Kompleks(Kompleks a)
{
    mGercek = a.Gercek;
    mSanal = a.Sanal;
}

public void Yazdir()
{
    if (mSanal > 0)
        Console.WriteLine("{ 0} + j{ 1} ",mGercek,mSanal);
    else
        Console.WriteLine("{ 0} - j{ 1} ",mGercek,-mSanal);
}
}

```

Kompleks sınıfının yapıçı metodlarından dolayı aşağıdaki gibi nesneler oluşturabiliriz.

```
Kompleks a = new Kompleks(-5,9);
```

```

Kompleks b = new Kompleks();
Kompleks c = new Kompleks(a);

```

## Aritmetik Operatörlerinin Aşırı Yüklenmesi

Bu kısımda +,\*,- ve / operatör metodlarının çeşitli versiyonlarını yazacağız. Önce + operatöründen başlayalım. İki kompleks sayının toplamı gerçek ve sanal kısımlarının ayrı ayrı toplamından ibarettir. operator+ metodu bu işlemi yapıp sonucu yeni bir Komplex sayı nesnesi olarak geri döndürmelidir.

Aşağıdaki operator+ metodu

```
Kompleks3 = Kompleks1 + Kompleks2
```

işlemi yapmamıza yarayacaktır.

Şimdi, Kompleks sınıfına aşağıdaki operator+ metodunu ekleyin.

```

public static Kompleks operator+(Kompleks a, Kompleks b)
{
    double GercekToplam = a.Gercek + b.Gercek;
    double SanalToplam = a.Sanal + b.Sanal;
    return new Kompleks(GercekToplam,SanalToplam);
}

```

Bu metodu ekledikten sonra aşağıdaki programı derleyip çalıştırın:

```

using System;
class OperatorYukleme
{
    static void Main()
    {
        Kompleks a = new Kompleks(-5,9);
        Kompleks b = new Kompleks(1,2);
        Kompleks c = a + b;
        c.Yazdir();
    }
}

```

Göreksiniz ki ekrana  $-4 + j11$  yazacak.

İki Kompleks sayıyı toplamayı başardık. Peki bir kompleks sayı ile bir tamsayıyı toplamak istersek ne yapacağız? Bu durumda operator+ metodunun yeni bir versyonunu yazmalıyız, bu versiyonu da aşağıdaki ifadeyi legal kılacak şekilde oluşturmalıyız.

```
Kompleks2 = Kompleks1 + Tamsayı
```

Buradaki Tamsayı herhangi bir türden olabileceğि için bütün türler için ayrı birer metot yazmak gereklidir. Biz buradaki Tamsayıyı double türünden olacak şekilde düşüneceğiz. Böylece double türünden küçük tipler içinde aslında metodu yazmış oluruz.

Bir Kompleks sayı ile double türünden sayının toplanmasını sağlayan operator+ metodu aşağıdaki gibidir.

```
public static Kompleks operator+(Kompleks a, double b)
{
    double GercekToplam = a.Gercek + b;
    return new Kompleks(GercekToplam, a.Sanal);
}
```

Bu metot ile aşağıdaki ifadelerin geçerli olmasını sağlamış olduk.

```
Kompleks a = new Kompleks(-5, 9);
```

```
Kompleks c = a + 5;
```

```
Kompleks d = a + 5d;
```

```
Kompleks e = a + 5.6f;
```

```
byte b = 5;
```

```
Kompleks c = a + b;
```

Ancak bu metot ile

```
Kompleks c = 5 + a;
```

ifadesi geçerli kılınmaz. Bu ifadenin geçerli olması içinse ayrı bir operator+ metodu bildirilmelidir. Bu yeni metotta değişen tek şey parametrelerin sırasıdır. Bu metodu aşağıdaki gibi tasarlayabiliriz.

```
public static Kompleks operator+(double b, Kompleks a)
{
    return a + b;
}
```

Görülügü gibi operator+ metodunun bu versiyonunda parametreler yer değiştirilecek bir önceki operator+ metodu çağrılmıştır.

operator- metodu ile ilgili yapacaklarımıza operator+ ile hemen hemen aynıdır. Tek fark metodların gövdesinde toplama yerine çıkarma yapılmasıdır. operator- metodunun 3 versiyonu da aşağıda toplu olarak verilmiştir.

```
public static Kompleks operator-(Kompleks a, Kompleks b)
{
    double GercekFark = a.Gercek - b.Gercek;
    double SanalFark = a.Sanal - b.Sanal;
    return new Kompleks(GercekFark, SanalFark);
}
```

```
public static Kompleks operator-(Kompleks a, double b)
{
    double GercekFark = a.Gercek - b;
    return new Kompleks(GercekFark, a.Sanal);
}
```

```
public static Kompleks operator-(double b, Kompleks a)
```

```
{
    double GercekFark = b - a.Gercek;
    return new Kompleks(GercekFark, a.Sanal);
}
```

operator- metodu ile ilgili dikkat çeken en önemli özellik işlemin değişme özelliğinin olmamasıdır. Bu yüzden 5-Kompleks ile Kompleks – 5 ifadelerini hesaplayan metotları birbirlerine bağlı olarak geliştiremedik.

Şimdi de \* ve / operatörlerini aşırı yükleyerek operator\* ve operator/ metodlarını oluşturalım. Bu metotlar operator- ve operator+ metodları ile hemen hemen aynı şekilde tanımlanır tek değişen işlemlerin farklı olmasıdır.

Burada sadece

```
operator*(Kompleks a, Kompleks b)
```

ve

```
operator/(Kompleks a, Kompleks b)
```

metotları yapılacaktır. Diğer iki versiyonun yapılması okura bırakılmıştır.

Metotları yazmadan önce iki kompleks sayının çarpılması ve bölümnesini hatırlatalım. Örneğin;

$$\begin{aligned} (3 + j5) * (5 - j2) \text{ ifadesinin değeri} \\ &= 3*5 - j3*2 + j5*5 - 2*5*j*j \\ &= 15 - j6 + j25 + 10 \quad (j*j = -1) \\ &= 25 + j19 \end{aligned}$$

olar.

Aynı şekilde

$$\begin{aligned} (5 + j10) / (3 + j4) \text{ ifadesinin değeri} \\ &= (5 + j10) * (3 - 4j) / (3 + 4j) * (3 - 4j) \\ &= (55 + j10) / 25 \\ &= 2.2 + j0.4 \end{aligned}$$

olar.

Şimdi de bu işlemleri yapacak operator\* ve operator/ metodlarını yazalım.

```
public static Kompleks operator*(Kompleks a, Kompleks b)
{
    double Sanal1 = a.Gercek * b.Sanal;
    double Sanal2 = a.Sanal * b.Gercek;
    double SanalCarpim = Sanal1 + Sanal2;

    double Gercek1 = a.Gercek * b.Gercek;
    double Gercek2 = a.Sanal * b.Sanal;
    double GercekCarpim = Gercek1 - Gercek2;
```

```

        return new Kompleks(GercekCarpim, SanalCarpim);
    }

Aynı şekilde operator/ metodu da aşağıdaki gibi yazılabilir.

public static Kompleks operator/(Kompleks a, Kompleks b)
{
    Kompleks bEslenik = new Kompleks(b.Gercek, -b.Sanal);
    Kompleks Pay = a * bEslenik;
    double Payda = b.Gercek * b.Gercek + b.Sanal * b.Sanal;

    double BolumGercek = Pay.Gercek / Payda;
    double BolumSanal = Pay.Sanal / Payda;

    return new Kompleks(BolumGercek, BolumSanal);
}

```

## İlişkisel Operatörlerin Aşırı Yüklenmesi

İlişkisel operatör metodları true ya da false değer ile geri dönerler. Bu yüzden bu tür operatörlerin metodlarını yazarken buna özen göstermemiz gereklidir. operator< metodunun gecriye int ya da string bir değer döndürmesi mantıklı değildir. Ama bu tür kullanımlar da geçerlidir. void dışında herhangi bir geri dönüş değeri ile ilişkisel operatör metodlarını bildirebiliriz. Operatörlerin anlamsal bütünlüğünü korumak için biz asla true ya da false dışında bir değerle ilişkisel operatör metodlarını geri döndürmeyeceğiz.

İlişkisel operatörler 6 tanedir. Bunlar !=, ==, <, >, <= ve >= operatörleridir. Bu operatörlerin aşırı yüklenmesi ile ilgili tek kural zıt anlamlı operatörlerin her ikisinin de aynı anda yüklenmiş olma zorunluluğudur. Yani operator== metodu bildirilmişse operator!= metodu da bildirilmelidir. Aynı koşul < ve > operatörleri ile <= ve >= operatörleri için de geçerlidir. Zıt anlamlı bu operatörlerden birini bildirmiş olmak diğerinin de tersini bildirmiş olmak anlamına gelir. Şimdi Kompleks sınıfı için operator!= ve operator== metodlarını bildirelim.

```

public static bool operator==(Kompleks a, Kompleks b)
{
    if(a.Sanal == b.Sanal && a.Gercek == b.Gercek)
        return true;
    else
        return false;
}

public static bool operator!=(Kompleks a, Kompleks b)
{
    return !(a == b);
}

```

Görüldüğü gibi operator!= metodunun geri dönüş değeri operator== metodunun geri dönüş değerinin tersi olacak şekilde düzenlenmiştir.

Bu iki metotla aşağıdaki kullanımlar geçerli hale gelmiştir.

```

Kompleks a = new Kompleks(5,10);
Kompleks b = new Kompleks(1,10);
bool c = a != b;
bool d = a == b;

```

Diger ilişkisel operatör metodlarının yazılması okuyucuya bırakılmıştır.

## true ve false Operatörlerinin Aşırı Yüklenmesi

true ve false operatörleri de aynı anda olmak şartıyla aşırı yüklenebilirler. true ve false değerleri if deyiminde, while ve for gibi döngülerde sıkılıkla kullanılır. Örneğin eğer bu operatörler aşırı yüklenirse koşul ifadelerini aşağıdaki gibi kurma imkanına kavuşuruz.

```

if(Kompleks)
    BiseylerYap();
else
    BaskaSeylerYap();

```

Bu tür operatör metodları genellikle ilişkisel operatörlerin ve mantıksal operatörlerin ifadelerde sıkılıkla kullanılmasının mantıklı olduğu durumlarda yüklenir. Örneğin Kompleks sınıfı için eğer en az bir bileşen sıfırdan farklı ise **operator true** için true değerine geri dönülür, her iki bileşen de sıfır ise false değerine geri dönülür. **operator false** metodu için ise bu işlemlerin tersi yapılır.

Bu operatörlerin tek bir operandı olduğu için metodların prototipi aşağıdaki gibi olmalıdır.

```

public static bool operator true(Kompleks a)
public static bool operator false(Kompleks a)

```

**operator true** ve **operator false** metodları aşağıdaki gibidir.

```

public static bool operator true(Kompleks a)
{
    if(a.Sanal != 0 || a.Gercek != 0)
        return true;
    else
        return false;
}

public static bool operator false(Kompleks a)
{
    if(a.Sanal == 0 && a.Gercek == 0)
        return true;
    else
        return false;
}

```

Bu operatörleri aşağıdaki program ile test edelim.

```

using System;
class OperatorYukleme
{
    static void Main()
    {
        Kompleks a = new Kompleks(5,10);
        Kompleks b = new Kompleks(0,0);

        if(a)
            Console.WriteLine("a sıfır değil");
        else
            Console.WriteLine("a sıfır");

        if(b)
            Console.WriteLine("b sıfır değil");
        else
            Console.WriteLine("b sıfır");
    }
}

```

```

D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum8\kod>Operatoryukleme
a sıfır değil
b sıfır
H:\cskitap\kitap\Bolum8\kod>

```



Bu programı derlediğinizde karşınıza çıkacak uyarıları dikkate almayın. Çünkü uyarı ile ilgili konuyu henüz görmedik.

## Mantıksal Operatörlerinin Aşırı Yüklenmesi

Mantıksal operatörleri yükleme konusunu iki kısımda incelememiz gereklidir. `&`, `|`, `!`, `&&` ve `||` operatörleri C#'taki mantıksal operatörleridir. Bunlardan `||` ve `&&` operatörlerinin aşırı yüklenebilmesi için birtakım ön şartlar vardır. Bu operatörlere kısa devre operatörleri de denildiğini hatırlayın.

Şimdi Kompleks sınıfı için `&`, `!` ve `|` operatör metotlarını yazalım. Bu operatörler iki operandlidir ve üretilen sonuç bool türünden olduğu için operator metotları da bu sonuca uyumlu bir şekilde hazırlanmalıdır. Bu operatör metotları aşağıdaki gibidir.

```

public static bool operator| (Kompleks a,Kompleks b)
{
    if((a.Sanal!=0 || a.Gercek!=0) | (b.Sanal!=0 || b.Gercek!=0))
        return true;
    else
        return false;
}

```

```

    }

    public static bool operator&(Kompleks a,Kompleks b)
    {
        if((a.Sanal==0 && a.Gercek==0) | (b.Sanal==0 && b.Gercek==0))
            return false;
        else
            return true;
    }

    public static bool operator!(Kompleks a)
    {
        if((a.Gercek != 0) | (a.Sanal != 0))
            return false;
        else
            return true;
    }
}

```

Yukarıdaki metodların sonuçlarını aşağıdaki test programını yazarak görün.

```

using System;
class OperatorYukleme
{
    static void Main()
    {
        Kompleks a = new Kompleks(5,1);
        Kompleks b = new Kompleks(0,1);
        Kompleks c = new Kompleks(0,0);

        if(a & b)
            Console.WriteLine("a & b doğru");
        else
            Console.WriteLine("a & b doğru değil");

        if(a | c)
            Console.WriteLine("a | b doğru");
        else
            Console.WriteLine("a | b doğru değil");

        if(!b)
            Console.WriteLine("!b doğru");
        else
            Console.WriteLine("!b doğru değil");
    }
}

```

Bu programın çıktısı aşağıdaki gibidir. (Programı derlerken Kompleks.cs dosyasını derleme işlemeye dahil etmeyi unutmayın.)

```
D:\WINNT\System32\cmd.exe
H:\neskitap\kitap\Bolum8\kod>Mantiksal
a & b dogru
a | b dogru
!b dogru degil
H:\neskitap\kitap\Bolum8\kod>
```

Şimdi de kısa devre operatörleri dediğimiz `&&` ve `||` operatörlerinin aşırı yüklenmesine geçelim. Normal olarak bu operatörleri aşırı yüklemek mümkün değildir. Ancak diğer mantıksal operatörlerinin aşırı yüklenmesi ile ilgili birtakım şartlar sağlandığı takdirde `&&` ve `||` operatörlerinin kullanımına olanak verilmiş olur. Bu şartların hepsi ifadelerde mantıksal bir bütünlüğü sağlamak için öngörülmüştür. Bu şartlar aşağıda sıralanmıştır.

1. `&` ve `|` operatörleri aşırı yüklenmiş olmalıdır.
2. `true` ve `false` operatörlerinin aşırı yüklenmiş olması gereklidir.
3. `operator&` ve `operator|` metodlarının parametreleri ilgili sınıfın türünden olmalıdır.
4. `operator&` ve `operator|` metodlarının geri dönüş değeri ilgili sınıfın türünden olmalıdır.

Şimdi Kompleks sınıfı için kısa devre operatörlerinin kullanımını mümkün kılacak şartları sağlayalım. Bu yüzden `operator|` ve `operator&` yeniden düzenlenecektir. `operator true` ve `operator false` metodları zaten daha önce tanımlanmıştır.

`&` ve `|` operatörlerinin aşırı yüklenmiş yeni metodlarını aşağıdaki gibi değiştirelim.

```
public static Kompleks operator&(Kompleks a, Kompleks b)
{
    if((a.Sanal==0 && a.Gercek==0) | (b.Sanal==0 && b.Gercek==0))
        return new Kompleks(0,0);
    else
        return new Kompleks(1,1);
}

public static Kompleks operator|(Kompleks a, Kompleks b)
{
    if((a.Sanal!=0 || a.Gercek!=0) | (b.Sanal!=0 || b.Gercek!=0))
        return new Kompleks(1,1);
    else
        return new Kompleks(0,0);
}
```

Yukarıdaki metodları içeren programın çıktısı bir önceki programın çıktısı ile aynı olacaktır (Test programının ayınısını bu yeni metodlar ile çalıştırın.). Dikkat ederseniz geri dönüş değerleri `true` ifadesini karşılamak için her iki elemanı da sıfırdan farklı olan yeni bir Kompleks sayı, `false` değerini ifade etmek içinse her iki elemanı da sıfır olan yeni bir Kompleks sayı olarak düzenlenmiştir.

## Dönüşüm Operatörünün Aşırı Yüklenmesi

Tasarladığımız sınıfların temel veri türleri ile ya da tanımladığımız diğer sınıflar ile uyumlu çalışabilmesini sağlamak için C#’ta tür dönüştürme işlemleri de aşırı yüklenebilir. Türler arasındaki dönüşümlerin karakteristik yapısını belirlemek için dönüşüm operatörleri sınıf için aşırı yüklenebilir. Tür dönüştürme metodları ile aşağıdaki gibi bir deyimin nasıl bir sonuc üreteceğini belirleyebiliriz.

```
Kompleks a = new Kompleks();
int i = a;
```

Normal şartlarda bu deyimleri içeren bir program derlenemez. Çünkü Kompleks türenen bir nesne int türenen bir değişkene atanmıştır. Bu iki tür arasında herhangi bir dönüşüm kuralı belirlenmediği için derleyici bu durumda ne yapacağına karar veremez. Hatırlarsanız yukarıdaki kullanım bilincsiz (`implicit`) tür dönüşüm kapsamına girmektedir. Aynı şekilde bilinçli (`explicit`) yapılan tür dönüşümlerinde de dönüşüm hakkında herhangi bir kural olmadığı için aşağıdaki kullanım da geçersizdir.

```
Kompleks a = new Kompleks();
int i = (int)a;
```

Bu kural elbette ki bütün veri türleri için geçerlidir. İşte bu tür kullanımları geçerli hale getirmek için tür dönüştürme operatör metodları bildirilir. Yukarıdaki iki kullanım birbirinden tamamen farklıdır. Yani her iki durum için de farklı metodlar tanımlanmalıdır.

Bu operatör metodlarının en genel ifadesi aşağıdaki gibidir.

```
public static implicit operator Hedef_Tür(Dönüştürülecek_Tür a)
{
    return Hedef_Tür;
}
```

```
public static explicit operator Hedef_Tür(Dönüştürülecek_Tür a)
{
    return Hedef_Tür;
}
```

Örneğin, `a` bir Kompleks sayı ise

```
int i = a;

deyiminin geçerli olması için
```

```
public static implicit operator int(Kompleks a)
{
    return 5;
}
```

gibi bir metot bildirilir. Geri dönüş değeri int türünden ve Kompleks sayıdan elde edebileceğimiz bir değer olacaktır. Aynı şekilde,

```
int i = (int) a;

deyiminin geçerli olması için ise

public static explicit operator int(Kompleks a)
{
    return 5;
}
```

metodunun bildirilmesi gereklidir.

Dönüşüm operatörleri ile ilgili önemli bir kısıtlama vardır. Yukarıdaki iki metodu bir sınıf için aynı anda bildirememiz. Yani, geri dönüş türü ve parametresi aynı olup da hem explicit hem de implicit metotlar bildirilemez. Ancak implicit operator metotları bildirildiğinde bilinçli yapılan tür dönüşümlerinde bu metotlar otomatik olarak çağrılır. Ancak explicit operatör metodlarında bu geçerli değildir. Yani explicit operator metodunun çağrılması durumunda dönüşüm olabilmesi için mutlaka tür dönüştürme operatörü kullanılmalıdır.

Şimdi implicit operator metodunu Kompleks sınıfı için bildirelim. Dönüştürülecek tür olarak double türünü seçiyoruz. Çünkü Kompleks sınıfının bütün üye değişkenlerini double türünden seçmişiktir. Tabi bu bir şart değil, sınıfınızın kendi içinde tutarlı bir davranış göstermesi için double türünü seçiyoruz.

İlk önce bir double türünden nesneye Kompleks sayı atandığında double türünden nesneye hangi değerin atanacağını belirlemek gerekmektedir. Kompleks sınıfı için tür dönüşümü çok fazla anlamlı gelmeyecek, ancak bu operator metodlarını kafanızda daha iyi canlandırmak için bu örnekleri vermekte faydalı olabilir. Sonuç olarak double türünden nesneye kompleks sayının gerçek sayı kısmının atanmasını istiyoruz.

Aşağıdaki **implicit operator int** metodunu Kompleks.cs dosyasına ekleyin ve ardından test işlemleri için bir sonraki programı yazın.

```
public static implicit operator double(Kompleks a)
{
    return a.Gercek;
}
```

Aşağıdaki programı yazarak double türünden ve diğer türden nesnelere atanmış komplex sayıların nasıl bir sonuç çıkardığını görün.

```
using System;
class TurDonusumu
{
    static void Main()
    {
        Kompleks kompleks = new Kompleks(5,1);

        double d = kompleks;

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

Programı derleyip çalıştırıldığınızda ekrana 5 yazdığını göreceksiniz. Bu metodu kullanarak kompleks sayıyı double türünün dışındaki bir türden nesneye atayamayız. Örneğin aşağıdaki kullanım geçersizdir.

```
int d = kompleks;
```

Bu atamanın geçerli olması için

```
public static implicit operator int(Kompleks a)

metodunun tanımlanması gereklidir.
```

**implicit operator int** metodunu bildirmiş olmamız aynı zamanda bilinçli (explicit) olarak tür dönüşümü yapmamıza da imkan vermektedir. Yani,

```
double d = (double)kompleks;
```

gibi bir kullanım tamamen geçerlidir.

Diğer bir önemli nokta da tür dönüştürme operatörü ile bilinçli (explicit) yapılan dönüşümlerin de herhangi bir veri türüne dönüşüm yapabilmemizdir. Örneğin aşağıdaki kullanım tamamen geçerlidir.

```
int d = (int)kompleks;
```

Gördüğünüz gibi implicit operator double metodunu bildirmiş olmamıza rağmen int türünden nesnelere kompleks bir sayıyı tür dönüştürme operatörü kullanarak atayabiliyoruz. Ancak bu durumda double türü int türüne dönüştürüldüğü için veri kaybının olma ihtimali vardır. Veri kayıplarının olmaması için tür dönüştürme operatörünü dikkatli kullanılmalıdır.

Şimdi de bu işlemi tersten yapalım. Yani bir kompleks sayıya başka türden bir nesneyi atamaya çalışalım. Yani aşağıdaki kullanımı geçerli hale getirelim.

```
Kompleks kompleks = new Kompleks(5,1);
kompleks = 8.5;
```

Yukarıda bildirdiğimiz bu **implicit operator int** metodu bu durumda çağrılamaz. Bu dönüşümün olabilmesi için (double türünün Kompleks türüne) **implicit operator Kompleks** metodunu bildirmemiz gereklidir. Bu metodun parametresi ise int türden olmalıdır. Diğer metottan farkı, gördüğünüz gibi geri dönüş değeri ile parametresinin yer değiştirmiştir.

Bu metodu aşağıdaki gibi yazabiliriz.

```
public static implicit operator Kompleks(double a)
{
    return new Kompleks(a, 0);
}
```

Bu metodu da Kompleks.cs dosyasına ekleyip aşağıdaki programı yazın.

```
using System;
class OperatorYukleme
{
    static void Main()
    {
        Kompleks kompleks = new Kompleks(5, 1);

        kompleks = 8.5;

        Console.WriteLine(kompleks.Gercek);
    }
}
```

Programı derleyip çalıştırığınızda ekrana 8.5 yazdığını göreceksiniz. Bu da komplex nesnesinin Gerçek üye elemanına doğru bir atama yaptığı göstermektedir.

double türü byte, int ve char türünden daha büyük bir tür olduğu için aşağıdaki atama ifadelerinin tamamının doğru olduğunu söyleyebiliriz.

```
kompleks = (byte)10;
kompleks = (int)10;
kompleks = (char)10;
```

Şimdi de **explicit operator** metodlarının bildirilmesini inceleyelim. Yukarıda da de-nildiği gibi **implicit operator** metodlarını tanımladığımızda bilinçli (**explicit**) olarak yaptığımız tür dönüşümleri geçerli kılınmaktadır. Ancak bazı durumlarda tür dönüşümlerinin sadece bilinçli olarak yapılmamasını isteyebiliriz. Bu durumda otomatik tür dönüşümü yapmak istendiğinde **explicit operator** metodları çağrılmayacaktır.

Şimdi

```
public static explicit operator double(Kompleks a)
```

metodunu **explicit** olacak şekilde yeniden düzenleyelim. (Unutmayın aynı tür için hem **explicit** hem de **implicit operator** metodları bildirilemez.)

```
public static explicit operator double(Kompleks a)
{
    return a.Gercek;
}
```

Gördüğünüz gibi diğer metottan tek fark **implicit** anahtar sözcüğünün yerine, **explicit** anahtar sözcüğünün gelmesidir.

Bu metodu bildirdikten sonra,

```
double a = kompleks;
ataması geçersiz iken
double a = (double)kompleks;
ataması geçerlidir.
```

Tür dönüşüm operator metodlarını bildirirken dikkat etmemiz gereken en önemli nokta, bilinçsiz dönüşümler sırasında veri kayıplarının olmasına imkan vermemektir. Eğer veri kaybı olma ihtimali söz konusu ise **explicit operator** metodlarını kullanmanızı tavsiye ederim.

## Operatörlerin Aşırı Yüklenmesine Genel Bakış

C#'taki birçok operatörün nasıl aşırı yüklediğini inceledik. Belki dikkatinizi çekmişdir, atama ve işlemli atama operatör metotlarını tanımlamadık. Bu operatörlerin aşırı yüklenmesi mümkün değildir. Ancak **operator+** metodu yüklenliğinde **+=** operatörü de bir anlamda yüklenmiş sayılır. Derleyici

```
a += b;
```

deyimini gördüğünde otomatik olarak **operator+** metodunu çağırır. Bu işlem atama operatörü olan bütün operatörler için geçerlidir. Örneğin **/=**, **-=**, **\*=** gibi.

Operatorleri yüklerken operatör öncelik sırasını değiştiremeyeceğimiz gibi temel veri türleri için operatörleri yeniden yükleyemeyiz.

C# dilinde aşırı yükleme yapamayacağımız birkaç operatör daha vardır. Bunlar **&&**, **||**, **[]**, **()**, **=**, **? .. ? :**, **->**, **is**, **sizeof**, **new** ve **typeof** operatörleridir.

Hatırlarsanız **&&** ve **||** operatörleri direkt aşırı yüklenmemelerine rağmen bazı koşulları sağlayarak bu operatörlerin çalışma koşullarını sağlayabiliyoruz.

[ ] operatörü aşırı yüklenememesine rağmen birazdan inceleyeceğimiz indeksleyiciler konusunda zaten buna gerek olmadığını göreceksiniz.

## İndeksleyiciler (Indexers)

İndeksleme (indexing) sözcüğü, [ ] operatöründen gelmektedir. Diziler konusunda [ ] operatörü ile bir dizinin elemanlarına ulaşığımızı söylemişik. Yine aynı bölümde bütün dizilerin System.Array türünden birer nesne olduğunu söylemişik. System.Array sınıfında tanımlanan indeksleyici sayesinde nesnelerin elemanlarına [ ] operatörünü kullanarak erişebiliyoruz. Eğer operator[ ] metodu olsaydı indeksleyiciler ile aynı işi yapacaklardı. İndeksleyiciler genellikle sınıfımızın üye elemanlarından biri dizi ya da benzeri bir türden ise kullanışlı olmaktadır. Ancak bu amacın dışında kullanmamız da tamamen serbest bırakılmıştır. Fakat biz iyi bir programcı olarak hiçbir zaman operatörleri gerçek anlamları dışında bir anlamla yüklememeliyiz. İndeksleyiciler de diziler gibi tek ya da daha çok boyutlu olabilmektedir. Şimdi örnek bir sınıf üzerinden indeksleyici tanımlamayı görelim.

### Tek Boyutlu İndeksleyici

En genel anlamda tek boyutlu bir indeksleyici aşağıdaki gibi tanımlanır.

```
Eleman_Tipi this[ indeks_tipi indeks]
{
    get
    {
        return Eleman_Tipi;
    }

    set
    {
        //işlemler
    }
}
```

Şimdi bu yapıyı yakından inceleyelim: Eleman\_Tipi, [ ] operatörü ile ulaşımak istenen nesnenin türünü belirtmektedir. indeks\_tipi ise [ ] operatörü ile kullanılacak indeks değerinin türünü belirtir. Dizilerde bu tür tipik olarak int türüdür, ancak bizim sınıflarımızda bu int olmak zorunda değildir. Ama int türü dışında bir tür kullanmak indeksleyicinin varoluş amacına ters düşeceği için genellikle int olarak alınır.

Aşağıdaki örnek sınıfta bir indeksleyicinin çalışma mantığını inceleyeceğiz. Bu örnekteki kullanım indeksleyicinin genel kullanımı dışındaki bir kullanımıdır. Ancak indeksleyiciler ile neler yapabileceğimizi görmek açısından bu örnek önemlidir.

```
using System;
class Indexer
{
```

```
public double sayi;

public double this[ double indeks]
{
    get
    {
        return indeks * indeks;
    }

    set
    {
        sayi = value;
    }
}

class indeksleyici
{
    static void Main()
    {
        Indexer i = new Indexer();

        Console.WriteLine("i[ 1.2 ]={ 0 } ", i[ 1.2 ] );

        i[ 5 ] = 5;

        Console.WriteLine(i.sayi);

        i[ 5 ] = 8;

        Console.WriteLine(i.sayi);

        Console.WriteLine("i[ 5 ]={ 0 } ", i[ 5 ] );
    }
}
```

Bu programın önce ekran çıktısına bakalım:

```
cmd | D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum8\kod>Indeksleyiciler
i[1.2]=1.44
5
8
i[5]=25
H:\cskitap\kitap\Bolum8\kod>
```

Gördüğünüz gibi

indexer[ sayı ]

biçimindeki bir kullanımda get blokları sayesinde geriye döndürülen değer, sayı\*sayı değeridir. set blokları ile ise [ ] operatörü ile gelen indeks numarasından bağımsız olarak Indexer sınıfının sayı elemanına değerler atanıyor. set bloğundaki value anahtar sözcüğünün kullanıldığına dikkat edin. Buradaki value değeri sınıfların özelliklerinde kullandığımız value değeri ile aynı işlevdedir. Bütün bu işlemleri normal bir metot ile de yapabildik.

Bu örnekte indeksleyicinin kullanımı çok gerekli olan bir şey değil. Indeksleyiciler çoğunlukla üye elemanlarından birinin dizi olduğu sınıflarda dizinin elemanlarına [ ] operatörü yardımıyla erişmek için kullanılır.

Aşağıdaki örnekte indeksleyicilerin gerçek amacına yönelik bir örnek verilmiştir.

```
using System;
class Indexer
{
    private int[] dizi;

    public Indexer(int DiziUzunlugu)
    {
        dizi = new int[DiziUzunlugu];
    }

    public int DiziBoyut
    {
        get
        {
            return dizi.Length;
        }
    }

    public int this[ int indeks ]
    {
        get
        {
            return dizi[indeks];
        }
        set
        {
            dizi[indeks] = value;
        }
    }
}

class indeksleyici
{
    static void Main()
    {
        Indexer x = new Indexer(5);

        for(int i=0; i < x.DiziBoyut; ++i)
            Console.WriteLine("i[{0}] = {1}", i, x[i]);
    }
}
```

Indexer sınıfının yapıcı metodu ile dizi'nin boyutu dinamik olarak belirleniyor. Dolayısıyla dizinin bütün elemanları varsayılan değere yani 0 değerine atanacaktır. In-

dexer türünden bir nesne oluşturup x[5] dediğimizde aslında Indexer sınıfındaki dizinin 5. elemanına ulaşıyoruz demektir. Dinamik olarak belirlenen dizinin boyutunu çalışma zamanında elde edebilmek için sadece get bloğu olan DiziBoyut isimli bir özellik tanımlanıyor.

Gördüğünüz gibi [ ] operatörü ile yine bir dizinin elemanlarına erişiliyor. Bu programı derleyip çalıştırıldığımızda aşağıdaki ekran görüntüsünü alırız.

```
H:\cskitap\kitap\Bolum8\kod>Indexer
i[0] = 0
i[1] = 0
i[2] = 0
i[3] = 0
i[4] = 0
H:\cskitap\kitap\Bolum8\kod>
```

İndeksleyicilerle sınıflarımıza nasıl bir dizi görüntüsü verdigimizi inceledik. Şimdi de indeksleyicilerle ilgili diğer önemli konuları inceleyelim.

İndeksleyicilerde metotlar gibi aşırı yüklenen indeksleyicilerin hangisinin seçileceği ise metodlarda olduğu gibi elemanların türlerine bakılarak yapılır. Bir sınıf için aşırı yüklenmiş dizilerin kullanımına çok az rastlanır. Zaten bu, pek de tercih edilen bir kullanım değildir. Aşağıdaki örnekte bir sınıfın farklı iki indeksleyicisini görebilirsiniz.

```
class Indexer
{
    private int[] dizi1;
    private double[] dizi2;

    public int this[ int indeks ]
    {
        get
        {
            return dizi1[indeks];
        }
        set
        {
            dizi1[indeks] = value;
        }
    }

    public double this[ double indeks ]
    {
        get
        {
            return dizi2[(int)indeks];
        }
    }
}
```

```

        set
        {
            dizi2[ (int)indeks] = value;
        }
    }
}

```

İndeksleyiciler aşırı yüklenerek sınıf içindeki birden fazla diziyeye [ ] operatörü ile ulaşılabilir.



Bir sınıfta indeksleyici tanımlamak için sınıfın herhangi bir üye elemanının dizi olması gerekmekz.

## Çok Boyutlu İndeksleyici

Çok boyutlu indeksleyicilerin kullanımı, tek boyutlu indeksleyiciler ile tamamen aynıdır. Tek fark ilgilenilen dizilerin boyutunun birden fazla olmasıdır.

İki boyutlu bir indeksleyiciye bir örnek verelim:

```

using System;
class Indexer2
{
    private int[,] dizi;

    public Indexer2(int Boyut1,int Boyut2)
    {
        dizi = new int[ Boyut1 ,Boyut2];
    }

    public int Boyut1
    {
        get
        {
            return dizi.GetLength(0);
        }
    }

    public int Boyut2
    {
        get
        {
            return dizi.GetLength(1);
        }
    }

    public int this[ int indeks1,int indeks2]
    {
        get
        {
            return dizi[ indeks1,indeks2];
        }
    }
}

```

```

        }
    }

    set
    {
        dizi[ indeks1,indeks2] = value;
    }
}

class indeksleyici
{
    static void Main()
    {
        Indexer2 x = new Indexer2(10,6);

        for(int i=0 ; i<x.Boyut1; ++i)
            for(int j=0 ; j<x.Boyut2; ++j)
                x[ i,j] = i + j;

        for(int i=0 ; i<x.Boyut1; ++i)
        {
            for(int j=0 ; j<x.Boyut2; ++j)
                Console.Write("{ 0,4} ",x[ i,j] );
            Console.WriteLine();
        }
    }
}

```

Boyut1 ve Boyut2'nin elde edilmesi ile ilgili oluşturduğumuz özelliklere dikkat edin. Bir önceki programda olduğu gibi bu özellikler yine sadece okunabilir olarak ayarlanmıştır.

İndeksleyiciyi tanımlarken boyut sayısı kadar indeks parametresi yazıldığını görüyorsunuz.

Bu programda bir öncekide yaptıklarımızın iki boyutlusunu yaptık sadece. Bu programı derleyip çalıştırduğumızda aşağıdaki ekran görüntüsünü elde ederiz.

0	1	2	3	4	5
1	2	3	4	5	6
2	3	4	5	6	7
3	4	5	6	7	8
4	5	6	7	8	9
5	6	7	8	9	10
6	7	8	9	10	11
7	8	9	10	11	12
8	9	10	11	12	13
9	10	11	12	13	14

## Yapılar (Structs)

C dilinde programlama yapanlar yapıları çok fazla kullanmaktadır. Yapılar C# di-

linde değer tipi olan verilerdir. Yapı bildirimleri sınıf bildirimini ile neredeyse aynıdır. **class** anahtar sözcüğü yerine **struct** anahtar sözcüğü konularak aynı kod parçaları yapıpaya çevrilebilir. Nesnelere referans yolu ile erişmek bazı durumlarda avantajlı olamayabilir. Örneğin birbiri ile ilişkili az sayıda değişkeni bir sınıf içinde tutmak bellek kullanımı açısından pek yararlı değildir. Referans tipleri için hem heap bölgesinde sınıfın üye elemanları için alan tahsis edilir hem de referans değerini tutmak için stack bölgesinde alan tahsis edilir. Heap bölgesindeki nesneye her ulaşmak istediğimizde ayrıca zaman geçeceği için az sayıda eleman içerecek ve sadece bir veri tipi olabilecek değişkenleri sınıf olarak tanımlamak yerine yapı olarak tanımlamak hız ve verimlilik açısından daha faydalıdır. Yapılar değer tipi oldukları için yapı türünden nesneler stack bellek bölgesinde saklanır.

Bir yapı bildirimini aşağıdaki gibi yapılabilir.

```
struct YAPI
{
    Elemanlar;
}
```

Yapılar genellikle birbirleriyle ilişkili değişkenleri bir yapıda toplamak için kullanılır. Yapılar nesne yönelimli programmanın bel kemiği olan kalıtımı desteklemez. Yani yapıları türemeyiz. Ancak yapı nesneleri de C#'taki diğer bütün nesneler gibi object'ten türemiştir. Şimdi Ogrenci adlı bir yapı bildirip çeşitli özelliklerini belirleyelim.

```
struct Ogrenci
{
    public string Ad;
    public string Soyad;
    public int no;
}
```

Gördüğünüz gibi yapı bildiriminin sınıf bildiriminden tek farkı, **class** yerine **struct** anahtar sözcüğünün kullanılmasıdır.

Yapıların kullanımı sınıfların kullanımı ile aynıdır. Bir yapı nesnesi tanımlamak için yine **new** anahtar sözcüğü kullanılabilir. **new** anahtar sözcüğü kullanıldığında yapının varsayılan yapıcı metodu ya da bizim tanımladığımız diğer yapıcı metotları çağırılır. **new** anahtar sözcüğünü kullanmadan da sınıflardan farklı olarak yapı nesnesi tanımlayabiliriz. Bu şekilde tanımlanan yapı nesnelerine ilk değer atanmayacağı için her bir üye elemana tek tek değer atanmalıdır. Yapı nesnelerinin üye elemanlarına sınıflarda olduğu gibi **:** operatörü ile erişilir.

Aşağıdaki iki örnekte Ogrenci yapısı türünden bir nesne **new** anahtar sözcüğü ile ve **new** anahtar sözcüğü olmadan ayrı ayrı tanımlanmıştır.

```
using System;
```

```
class Yapılar
{
    struct Ogrenci
    {
        string Ad;
        public string Soyad;
        public int no;
    }

    static void Main()
    {
        Ogrenci ali=new Ogrenci();
        Console.WriteLine(ali.no);
    }
}
```

Bu programı derleyip çalıştırığınızda ekrana 0 yazdığını göreceksiniz. Çünkü ali nesnesi oluşturulduğunda Ogrenci yapısının varsayılan yapıcı metodu ile bütün üye elemanlar varsayılan değere atanmıştır.

```
using System;
class Yapılar
{
    struct Ogrenci
    {
        string Ad;
        public string Soyad;
        public int no;
    }

    static void Main()
    {
        Ogrenci ali;
        Console.WriteLine(ali.no);
    }
}
```

Bu programda ise ali nesnesi tanımlanmış olmasına rağmen üye elemanlarına herhangi bir değer atanmamış olduğu için bu değişkenleri kullanamayız. Programı derlediğinizde:

“Use of possibly unassigned field ‘no’” yani “değeri atanmamış ‘no’ değişkenin kullanımı” hatası alırız.

Hatırlarsanız bu durum değer tipindeki temel veri türleri içinde geçerliydi. Yani aşağıdaki kullanım tamamen geçersizdir.

```
int i;
Console.WriteLine(i);
```

Buradan değer tipindeki veri türlerinin aslında birer yapı olduğunu çıkarabiliriz. Gerçekten de temel veri türlerinden olan int türü aslında System.Int32 yapısını temsil eder. Diğer değer tipi veri türleri için de bu geçerlidir.

Bir de aşağıdaki programı inceleyelim.

```
using System;
class Yapilar
{
    struct Ogrenci
    {
        public string Ad;
        public string Soyad;
        public int no;
    }

    static void Main()
    {
        Ogrenci can= new Ogrenci();

        Ogrenci ali = can;
        Console.WriteLine(ali.no);
    }
}
```

Bu programı derlediğinizde ekrana 0 yazdığını görürsünüz. Değer tiplerinde birbirine atanan nesnelerde bitsel bir kopyalama işlemi olmaktadır. Bu yüzden yukarıdaki ali nesnesi ile can nesnesi atama operatörü ile birbirlerine atanmasına rağmen her iki nesne de stack bölgesinde farklı alanlarda tutulurlar. Yani bu iki nesne birbirinden tamamen bağımsızdır. Birini değiştirmek diğerini değiştirmeyecektir. Hatırlarsanız sınıf nesnelerinde bu böyle değildi.

Yapılar metotlara parametre olarak aktarılırken de parametre değişkenleri bitsel kopyalamaya tabi tutulur. Örneğin aşağıdaki metotta kendisine parametre olarak gelen yapı değişkenini değiştirmemize rağmen orijinal değişkenin değeri değişmemiştir.

```
using System;
class Yapilar
{
    public struct Ogrenci
    {
        public string Ad;
        public string Soyad;
        public int no;
    }
```

```
}
public static void Metot(Ogrenci o)
{
    o.no = 5;
}

static void Main()
{
    Ogrenci can= new Ogrenci();
    can.no = 125;

    Console.WriteLine(can.no);

    Metot(can);

    Console.WriteLine(can.no);
}
```

Bu programı derlediğimizde ekrana iki defa 125 yazacaktır. Bu da **Metot()** içinde değiştirilen no değişkeninin etkisinin can yapı nesnesine yansımadığını gösterir.

Yapıların da sınıflar gibi birden fazla yapıçı metotları olabilir. Ancak varsayılan yapıçıyı kendimiz bildiremeyiz. Mesela aşağıdaki yapıçı metot bildirimi derleme zamanında hata verecektir.

```
public struct Ogrenci
{
    public string Ad;
    public string Soyad;
    public int no;

    public Ogrenci()
    {
    }
}
```

Yapılarda bildirilen bütün yapıcların parametre alma zorunluluğu vardır. Örneğin aşağıdaki yapıçı metot bildirimi geçerlidir.

```
public Ogrenci(string ad,string soyad,int No)
{
    Ad = ad;
    Soyad = "dsa";
    no = No;
}
```

Yapıcı metot bildirildiği zaman yapının bütün elemanlarına ilk değer verme zorunluluğu vardır. Buna göre aşağıda bildirilen yapıçı metot geçersizdir.

```
public struct Ogrenci
{
    string Ad;
    public string Soyad;
    public int no;

    public Ogrenci(string ad, string soyad)
    {
        Ad = ad;
        Soyad = "dsa";
    }
}
```

Bu yapı bildirimini içeren bir programı derlediğimizde;

"Field 'Yapilar.Ogrenci.no' must be fully assigned before control leaves the constructor" yani "Kontrol yapıçı metottan çıkmadan önce 'Yapilar.Ogrenci.No' değişkenine mutlaka ilk değer verilmelidir" hatası alırız. Buradan yapıçı metotlarının, yapıda bulunan değişken sayısının kadar parametresinin olması gerekiği düşünülmemelidir. Örneğin aşağıdaki metot bildirimini tamamen geçerlidir.

```
public Ogrenci(string ad, string soyad)
{
    Ad = ad;
    Soyad = "dsa";
    no = 0;
}
```

Yapı nesneleri, faaliyet alanları bittiğinde, otomatik olarak stack bölgesinde silinirler. Programcının ayrıca bilinçli bir şekilde nesneye stack'de ayrılan alanı iade etmesine gerek yoktur. Bu yüzden yapılarda yıkıcı metot bildirmek yasaklanmıştır.

Sınıflarda olduğu gibi yapılar içinde değişkenler için get /set blokları ve indeksleyiciler tanımlanabilir. Sınıflara ilişkin konuda ayrıntılı bir şekilde bu konuları incelemiş olduğumuz için burada tekrar değinmeyeceğiz.

Son olarak yapıları bazı durumlarda neden kullanmamız gerektiğini anlamak için yapıların bazı avantajlarını listeleyelim.

1. Stack bellek bölgesinde bir değişken için alan ayırmak aynı işi heap bölgesinde yapmaktan daha hızlıdır.

2. Sınıf nesneleri Garbage Collection mekanizması ile heap alanından silinmektedir. Bir sınıf nesnesi için Garbage Collection mekanizmasının ne zaman devreye gireceği

kesin olarak belli değildir. Yani bir nesnenin faaliyet alanı yıkıcı metotlar çağrılmayabilir. Bu da programlarımızda nelerin olup bittiğini tam olarak anlayamamamıza neden olur. Ancak yapı nesnelerinde bu böyle değildir. Yapılarla yıkıcı metotlar olmamasına rağmen yapı nesnesinin faaliyet alanı dışına çıktıığında nesne bellekten otomatik olarak silinir.

3. Stack bölgesinde bulunan değişkenlerin kopyalanması daha hızlıdır.

## Numaralandırmalar (Enumeration)

Numaralandırma kelimesi enumeration sözcüğünden gelmektedir. Numaralandırmalar, enum anahtar sözcüğü ile bildirildikleri için bu kısımda numaralandırma yerine enum sabitleri sözcüklerini kullanacağız. Enum sabitleri C# dilinde çeşitli semboller tamsayılar ile ifade edebilmek için geliştirilmiş bir veri yapısıdır. Yapı olarak çok basit ama işlev olarak çok kullanışlı bir veri tipidir. Hatırlarsanız metotlar bölümünde bir dizinin elemanlarını ekrana yazan aşağıdaki方法u yazmıştık. İkinci parametre dizinin elemanlarını ekrana ne şekilde yazacağımızı belirtiyordu. Bu belirtme işlemini int türünden bir tamsayı ile belirtiyorduk. Bu durumda bu methodu kullanan birisi hangi sayı ile ne şekilde yazılacağını çok çabuk unutabilir. Halbuki ekrana yazma biçimini sözcüklerle ifade edebilmiş olsaydık işimiz daha kolay olurdu. Örneğin ikinci parametre DIKEY ise elemanlar ekrana alt alta yazılacak, YATAY ise elemanlar yanyana yazılacak. İşte bunu yapabilmek için enum sabitlerinden faydalanağız. Enum sabitleri yazılıardan oluşan sembollerini kendi belirlediğimiz numara sistemine göre numaralandırır. Örneğin DIKEY için 0, YATAY için 1 gibi. Şimdi iki elemanlı bir enum sabitin nasıl bildirildiğini inceleyelim. enum sabitleri enum anahtar sözcüğü ile bildirilir. Bildirimleri sınıf ve yapı bildirimine benzemektedir. YATAY ve DIKEY isimli iki elemanı olan bir enum sabiti aşağıdaki gibi bildirilir.

```
enum BICIM : numara_türü{ DIKEY, YATAY }
```

Bu bildirimde enum sabitinin adı BICIM'dir. numara\_türü enum sabitlerinin aslında hangi türü temsil ettiğini gösterir. Yani BICIM.YATAY ve BICIM.DIKEY sabitlerinin hangi türden olduğunu gösterir. Parantez içindeki yazılı ise numaralandırmak istediğimiz sembollerdir. Bu parantez içine enum sabitinin türünün alabileceği değer kadar sembol yazabiliz. Örneğin eğer enum sabiti byte türündense en fazla 255 tane sembol tanımlayabiliriz. Pratikte 255'den fazla sembol tanımlamakla çok karşılaşmayacağımız için enum sabitlerinin türünü byte olarak bildirmek programın hızını az da olsa artırır. En azından her sembol için 3 byte'lık bir bellek alanı kazancımız olur. (int türünün 4, byte türünün 1 byte olduğunu hatırlayın). Eğer yeteri kadar bellek alanınızın olduğunu düşünüyorsanız enum sabitinin türünü varsayılan değerde bırakın. Günümüz bilgisayar sistemlerinde bellek alanları artık yeteri kadar fazla olduğu için böyle bir kaygımız zaten olmayacak.

Enum sabitlerinin türü varsayılan olarak int türüdür yani aşağıdaki gibi bir bildirimde bütün sabitler int türündendir.

```
enum BICIM{ DIKEY, YATAY }
```

Eğer enum sabitlerinin byte türünden olmasını istiyorsak aşağıdaki gibi bir bildirim yapmalıyız.

```
enum BICIM : byte{ DIKEY, YATAY }
```

Enum sabitlerinin türünü char türü dışında bütün tamsayılar ile bildirebiliriz. Bunlar;

byte, sbyte, short, ushort, int, uint, long, ve ulong türleridir.

Şimdi gelin DiziYaz() metodunu enum sabitlerini kullanarak daha güzel bir hale getirelim. Aşağıdaki programda BICIM adlı bir enum sabiti bildirilmiştir.

```
using System;
enum BICIM : byte{ DIKEY, YATAY }
class AnaSınıf
{
    static void DiziYaz(Array dizi,BICIM b)
    {
        foreach(Object i in dizi)
            if(b == BICIM.DIKEY)
                Console.WriteLine(i.ToString());
            else
                Console.Write(i.ToString()+" ");
    }

    static void Main()
    {
        int[ ] a = new int[ 5 ];
        DiziYaz(a,BICIM.DIKEY);
        DiziYaz(a,BICIM.YATAY);
    }
}
```

Yukarıda da gördüğünüz gibi, enum sabitlerinin sembollerine enum sabitinin ismini kullanarak `?` operatörü ile ulaşıyoruz.

enum sabitlerinde parantez içinde yazılan ilk simbol 0 sayısını temsil etmektedir. Sonra gelen her simbol ise değeri açıkça belirtilmemiş olana kadar +1 olarak artmaktadır. Yani YATAY simbolu 1 sayısı ile ifade edilirken DIKEY simbolü 0 sayısı ile ifade edilir. Buna göre bu programdaki

```
if(b == BICIM.DIKEY)
```

deyimini

```
if(b == 0)
```

olarak değiştirmemiz herhangi bir fark oluşturmayacaktır.

Enum sabitleri de birer veri tipi olduğu için metotlara parametre olarak yukarıdaki gibi geçirilebilir.

Sabitlere karşılık düşen sayıların varsayılan olarak sıfırdan başladığını söylemek. Ancak bunu engellemek bizim elimizde; örneğin DIKEY simbolünün 5 sayısını YATAY simbolünün de 9 sayısını temsil etmesini sağlamak için bildirimi aşağıdaki gibi yapmalıyız.

```
enum BICIM : byte{ DIKEY=5, YATAY=9}
```

Eğer bu bildirimi aşağıdaki gibi yapmış olsaydık YATAY simbolünü 6 sayısını temsil ederdi.

```
enum BICIM : byte{ DIKEY=5, YATAY}
```

Aynı şekilde aşağıdaki NOT enum sabitinde

```
enum NOT : byte{ PEKIYI=5, IYI ,ORTA=10, GECERSIZ,ZAYIF}
```

PEKİYI → 5

IYI → 6

ORTA → 10

GECERSIZ → 11

ZAYIF → 12

olarak numaralandırılır.

Geleneksel olarak enum sabitleri büyük harflerle belirtilir, ancak bu mecburi değildir. Değişken isimlendirme kuralına uyan her simbol enum sabiti olabilir. Örneğin 5DIKEY ya da PEKİYI şeklinde bir simbol kullanamayız.



enum sembollerine de diğer türlerde olduğu gibi tür dönüşümü uygulanabilir. Örneğin

```
Console.WriteLine((int)BICIM.DIKEY);
```

deyimi ile ekrana DIKEY simbolunun sayı karşılığı yazılır.

```
Console.WriteLine(BICIM.DIKEY);
```

deyimi ile ise ekrana DIKEY yazısı yazılır.

Aşağıdaki atama işlemi ise tahmin ettiğiniz gibi son derece geçerlidir.

```
int a = (int) BICIM.YATAY;
```

## System.Enum Sınıfı

C#'ta bildirilen bütün enum sabitleri aslında System.Enum isimli bir sınıfından türetilmiştir. Bu yüzden oluşturduğumuz enum sabitleri üzerinden System.Enum sınıfının metodlarından faydalabiliriz.

Örneğin bir enum sabitinin bütün sembollerine ulaşabilmek için System.Enum sınıfının GetNames() metodunu kullanabiliriz. Bu metodun prototipi aşağıdaki gibidir.

```
public static string[] GetNames(Type a)
```

Buna göre haftanın günlerini içeren bir enum sabitinin sembollerini aşağıdaki gibi ekrana yazdırabiliriz.

```
using System;
enum BICIM : byte
{
    PAZARTESİ,
    SALI,
    CARSAMBA,
    PERSEMBE,
    CUMA,
    CUMARTESİ,
    PAZAR
}

class Class1
{
    static void Main()
    {
        string[] Semboller = BICIM.GetNames(typeof(BICIM));
        foreach(string s in Semboller)
            Console.WriteLine(s);
    }
}
```

Programı derleyip çalıştırıldığımızda aşağıdaki ekran görüntüsünü elde ederiz.

```
D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum8\kod>Enums
PAZARTESİ
SALI
CARSAMBA
PERSEMBE
CUMA
CUMARTESİ
PAZAR
H:\cskitap\kitap\Bolum8\kod>
```

## İsim Alanları ve System İsim Alanı

- **İsim Alanı Nedir?**
- **İsim Alanı Bildirimi**
- **using Anahtar Sözcüğü**
- **using ile Takma İsim (Alias) Verme**
- **İç içe (nested) Geçmiş İsim Alanları**
- **External Alias (Harici Takma İsimler)**
- **:: Operatörü**
- **global Harici Takma İsmi**
- **System İsim Alanı**
  - Temel Tür Yapıları
  - Tarih ve Zaman İşlemleri
  - BitConverter Sınıfı
  - Convert Sınıfı
  - Buffer Sınıfı
  - GC (Garbage Collector) Sınıfı

Bu bölümde .NET sınıf kütüphanelerinin en önemli organizasyon metodu olan *isim alanlarını* (*namespace*) inceleyeceğiz. Ardından .NET sınıf kütüphanesindeki en önemli *isim alanı* olan *System isim alanında* bulunan çeşitli sınıfları yakından inceleyeceğiz.

### İsim Alanı Nedir?

İsim alanları, yazdığımız programlarda mantıksal organizasyonu sağlar. Kodların organizasyonu kavramı tamamen yazılım teknolojisinin hızlı bir şekilde büyümeye ilgilidir. Gün geçtikçe dünyanın yazılıma olan ihtiyacı da artmaktadır. Çünkü teknoloji ile yazılım dünyası paralel ilerlemektedir. Bu sayede artan talepler, yeni programlama tekniklerinin geliştirilmesine yol açmıştır. Özellikle büyük yazılım projelerinde onlarca programcının çalışması gerektiğini düşünürsek bunu daha iyi kavrızır. Hatta bazen yüzlerce kişinin çalıştığı projeler de olabilmektedir. Bu durumda her yüz programcının da birbirinden bağımsız bir şekilde çalışması için organizasyonun iyi olması gereklidir. Organizasyondan kastımız her programcının yazdığı kodların ortak bir plat-

formda toplandığı zaman derleme hatasına yol açılmamasıdır. Örneğin C gibi yapısal programlama dillerinde birden fazla kişinin çalıştığı projelerde değişken isimlerinin ya da fonksiyon isimlerinin çakışma ihtimali çok yüksektir. Bildiğiniz gibi aynı erişebilirlik alanında aynı isimli birden fazla tip bildirimini bulunamaz. Bu isim çakışmasını engellemek için C dilinde çok gelişmiş bir organizasyon yöntemi bulunmamaktadır. Bu yüzden kodların çoğu anlamsız birtakım isimler oluşuyor. Örneğin aynı işi yapacak bir fonksiyonu bir kişi **firma1EkranaYaz()** şeklinde yazarken diğer bir kişi **firma2EkranaYaz()** şeklinde yazabilir. Görüüğünüz gibi bu böyle arttıkça tür ve fonksiyon isimlerinin önüne anlamsız birtakım örnekler gelecek. Bu hem programcı için hem de kodu okuyan için zorluk demektir.

Bu isim karmaşıklığının önüne geçmek için hemen bütün modern dillerde programın mantıksal bir yapı içine girmesi için çeşitli mekanizmalar vardır. Örneğin C++ ve C#'taki isim alanları ya da Java dilindeki paket (**package**) mantığı bu kullanımlara örnektir.

İsim alanları sayesinde isim benzerlikleri bir sorun olmaktan çıkmaktadır. Örneğin .NET sınıf kütüphanesindeki Array sınıfının yanında kendimiz de Array isimli bir sınıf tasarlayabiliriz. C gibi yapısal programlama dillerinde bu şekilde aynı isimli yapılar tanımlanamaz. Bu karmaşıkları önleyen elbette ki isim alanlarının oluşturduğu mantıksal organizasyondur. İsim alanları sadece mantıksal bir organizasyon sağlamsaktır. C++ dilindeki **include** anahtar sözcüğünün kullanımı ile karıştırılmamalıdır. **include** ile dosyalar fiziksel olarak birbirlerine bağlanırken isim alanları ile sadece mantıksal bir işlem yapılır. Yani isim alanları sadece derleme aşaması ile ilgilidir.

Eğer isim alanları olmasaydı, bütün değişken ve metod isimleri global düzeyde olacağı için .NET sınıfı kütüphanesi dışındaki bir kütüphaneyi kullanmak istediğimizde, bu kütüphanedeki bütün tür isimlerin standart olanlardan farklı olması gerekecekti. Bu da müthiş bir isim karmaşığına yol açacaktı.

Şu ana kadar yazmış olduğumuz programlardan aslında isim alanlarına pek yabancı değiliz. Örneğin her programımızın başına eklediğimiz

```
using System;
```

deyiği ile System isim alanını kullanmak istediğimizi derleyiciye bildiriyorduk. System isim alanını **using** anahtar sözcüğü ile eklememiş olsaydık da programlarını bir şekilde derleyebildirdik. Ancak System isim alanında bulunan bir tür direk erişemezdik. Örneğin aşağıdaki program derlenmeyecektir.

```
class AnaSınıf
{
    static void Main()
    {
        Console.WriteLine("sefer");
    }
}
```

Bu programın derlenememesinin sebebi Console isimli sınıfın bulunamamasıdır. Programı aşağıdaki gibi değiştirirsek herhangi bir hata almadan program derlenecektir.

```
class AnaSınıf
{
    static void Main()
    {
        System.Console.WriteLine("sefer");
    }
}
```

Bu programda ise Console sınıfının gerçek yolunu bildiriyoruz. Yani derleyiciye Console sınıfının System isim alanında bulunduğuunu bildiriyoruz.

Temel veri türlerini System isim alanını eklemeden de kullanabiliriz. Çünkü bu temel veri türleri C# derleyicisi tarafından otomatik olarak System isim alanında yapılar ve türetilir. Örneğin aşağıdaki programda int türünden bir değişken tanımlayabiliğen int türünün karşılığı olan Int32 türünden bir nesne tanımlayamayız.

```
class AnaSınıf
{
    static void Main()
    {
        //Geçerli tanımlama
        int a = 5;

        //Geçersiz tanımlama
        Int32 b = 10;
    }
}
```

İkinci tanımlamanın geçerli olması için ya programın başına

```
using System;
deyiminin eklenmesi gereklidir ya da tanımlamayı
System.Int32 b = 10;
şeklinde değiştirmemiz gereklidir.
```

Şu ana kadar yazdığımız programları isim alanı içinde yazmadık. Bunun nedeni derleyicinin varsayılan bir isim alanı eklemesidir. Eğer sınıflarımızı ve diğer tür bildirimlerini herhangi bir isim alanında yapmazsa derleyici kodlarımız için otomatik bir isim alanı belirler. Kaynak kodda tanımladığımız bütün türler bu varsayılan isim alanında dayanmış gibi davranışır.

Örneğin aşağıdaki programda Deneme sınıfı herhangi bir isim alanında bildirilmemesine rağmen Main() metodu içinden erişilebilmektedir.

```
using System;
class Deneme
{
    public int x;
    public int y;
    public Deneme(int x,int y)
    {
        this.x = x;
        this.y = y;
    }
}

class AnaSınıf
{
    static void Main()
    {
        Deneme d = new Deneme(5, 6);
        Console.WriteLine(d.x);
    }
}
```

Yukarıdaki programı derlediğimizde herhangi bir hata ile karşılaşmayız. Bunun sebebi derleyicinin global düzeyde bildirilen bütün tipler için otomatik bir isim alanı tanımlamasıdır.

## İsim Alanı Bildirimleri

Şimdi de kendi isim alanımızı nasıl tanımlayabileceğimize bakalım. İsim alanları namespace anahtar sözcüğü kullanılarak bildirilir. Bu anahtar sözcükten sonra isim alanının ismi verilir. Ardından gelen parantezler içinde ise tip bildirimleri yapılır. Bu bilgiler ışığında bir isim alanı bildiriminin aşağıdaki gibi yapıldığını söyleyebiliriz.

```
namespace CSharpNedir
{
    ...
    Tür Bildirimleri
    ...
}
```

Bu bildirimden sonra Deneme sınıfını bir isim alanı içinde bildirip olabilecek değişikliklere bakalım. Aşağıda bir isim alanının nasıl bildirildiğini inceleyin ve Deneme sınıfı türünden bir nesnenin nasıl oluşturulduğuna dikkat edin.

```
using System;
namespace CsharpNedir
{
```

```
class Deneme
{
    public int x;
    public int y;

    public Deneme(int x,int y)
    {
        this.x = x;
        this.y = y;
    }
}

class AnaSınıf
{
    static void Main()
    {
        CsharpNedir.Deneme d = new CsharpNedir.Deneme(5, 6);
        Console.WriteLine(d.x);
    }
}
```

Bu programda Deneme sınıfı türünden bir nesne oluşturmak için `CsharpNedir.Deneme d = new CsharpNedir.Deneme(5, 6);` deyiminin kullanıldığına dikkat edin. Deneme isimli sınıf artık global düzeyde olmadığı için Deneme sınıfı türünden bir nesneyi aşağıdaki şekilde oluşturamayız:

```
Deneme d = new Deneme(5, 6);
```

Ancak bir sonraki kısımda göreceğimiz `using` anahtar sözcüğünü kullanarak bu tür bir erişimi mümkün kılacağız.

İsim alanları diğer bloklarda olduğu gibi bloğun kapanmasıyla sonlanmaz. İstediğimiz kadar aynı isimli isim alanı oluşturabiliriz. Bu da farklı dosyalarda bulunan türlerin aynı mantıksal düzeyde bulunmasını sağlar. Bunu görebilmek için iki ayrı sınıf yazıp farklı dosyalara kaydedeceğiz.

`Deneme1.cs` dosyasını aşağıdaki gibi oluşturun.

```
namespace CsharpNedir
{
    class Deneme1
    {
        public int x;
        public int y;

        public Deneme1(int x,int y)
        {
```

```

        this.x = x;
        this.y = y;
    }
}
}

```

Deneme2.cs dosyasını da aşağıdaki gibi oluşturun.

```

namespace CsharpNedir
{
    class Deneme2
    {
        public int x;
        public int y;

        public Deneme2(int x,int y)
        {
            this.x = x;
            this.y = y;
        }
    }
}

```

Şimdi de ana programımızın bulunacağı Program.cs isimli dosyayı aşağıdaki gibi oluşturun

```

using System;
class AnaSınıf
{
    static void Main()
    {
        CsharpNedir.Deneme1 d1 = new CsharpNedir.Deneme1(5, 6);
        Console.WriteLine(d1.x);

        CsharpNedir.Deneme2 d2 = new CsharpNedir.Deneme2(10, 9);
        Console.WriteLine(d2.x);
    }
}

```

Yazdığımız bu 3 dosyayı aynı klasöre taşıyıp konsol modundan cd komutları ile bu klasöre gelin ve komut derleyicisine,

**csc Deneme1.cs Deneme2.cs Program.cs**

yazın.

Gördüğünüz gibi derleme zamanında herhangi bir hata almadık. Aynı isimli birden fazla isim alanını bu şekilde farklı dosyalarda oluşturabilmemiz, isim alanlarının sadece mantıksal bir gruptan olduğunu gösterir.

İsim alanlarını bildirmenin en önemli avantajı farklı isim alanlarında aynı isimli türleri bildirmede görülür. Diyelim ki 2D (iki boyutlu) grafikleri içeren bir sınıf kütüphanesi geliştirmiyoruz ve bu sınıf kütüphanesi içinde “Nokta” adlı bir sınıfımız var. Bu isim alanını aşağıdaki gibi bildiririz:

```

namespace Grafik2D
{
    public class Nokta
    {
        .....
    }
}

```

Grafik2D isim alanında bulunan sınıfları programcılar hizmetine sunduktan sonra bu sefer, aynı firma olarak 3D grafikleri için bir sınıf tasarlamak istedik. Nokta isimli sınıf bu sefer de Grafik3D isim alanında bulunmalıdır.

```

namespace Grafik3D
{
    public class Nokta
    {
        .....
    }
}

```

Bu iki isim alanında bulunan Nokta isimli sınıflar birbirleriyle karıştırılmadan kullanılabilir.

Örneğin her iki nokta türünden bir nesne aşağıdaki gibi tanımlanabilir.

```

Grafik2D.Nokta nokta2D = new Grafik2D.Nokta();
Grafik3D.Nokta nokta3D = new Grafik3D.Nokta();

```

Bu yaptıklarımızı çalışabilir bir örnek üzerinden gösterelim. Aşağıdaki programda farklı isim alanlarında tanımlanan ve isimleri aynı olan sınıfları nasıl kullanabileceğimizi görebilirsiniz.

```

using System;
namespace Grafik3D
{
    public class Nokta
    {
        public Nokta()
        {
            Console.WriteLine("3D nokta");
        }
    }
}

```

```

namespace Grafik2D
{
    public class Nokta
    {
        public Nokta()
        {
            Console.WriteLine("2D nokta");
        }
    }
}
class AnaSınıf
{
    static void Main()
    {
        Grafik2D.Nokta nokta2D = new Grafik2D.Nokta();
        Grafik3D.Nokta nokta3D = new Grafik3D.Nokta();
    }
}

```

Programı derleyip çalıştırığınızda ekrana

2D Nokta  
3D Nokta

yazdığını göreceksiniz.

İsim alanları yalnızca tür bildirimlerini içermelidir. Herhangi bir değişken tanımlaması ya da bir değişken yeralamaz. Örneğin aşağıdaki isim alanı bildirimi geçersizdir.

```

namespace Deneme
{
    int a=10;
}

```

Aynı şekilde metot bildirimleri de bir tür bilgisi olmadığı için isim alanlarında metot bildirimi yapamayız. Buna göre aşağıdaki isim alanı bildirimi geçersizdir.

```

namespace Deneme
{
    public int Metot()
    {
    }
}

```

İsim alanlarında yalnızca sınıf (class), temsilci (delegate), numaralandırma (enum), arayüz (interface) ya da yapı (struct) bildirimi yapılabilir.

## using Anahtar Sözcüğü

.NET Framework sınıf kütüphanelerindeki sınıflardan bahsederken sınıfların isimlerini isim alanları ile belirtmiştık. Örneğin Array sınıfını System.Array şeklinde Conso-

le sınıfını System.Console şeklinde belirtmiştık. Aslında using ile hiçbir isim alanını kodumuza eklemeden de .NET sınıf kütüphanesi kullanabiliriz. Örneğin daha sonra göreceğimiz ArrayList sınıfı türünden bir nesne oluşturmak için

```

System.Collections.ArrayList = new System.Collections.ArrayList()
deyimini yazmamız gereklidir.
System.Collections.ArrayList

```

ifadesine bir sınıfın tam (**full name**) ismi denilmektedir. Her ArrayList nesnesi tanımlarken System.Collections öne ekini kullanmak gerçekten sıkıcı olabilir. Bu yüzden **using** anahtar sözcüğü ile System isim alanı altında bulunan Collections isim alanı programa dahil edilir. Bu işlemi şu ana kadar gördüğümüz yöntemle yaparız. Yani programın başına

using System.Collections;

deyimini ekleriz.

Aşağıdaki programda Grafik3D isim alanı **using** anahtar sözcüğü ile derleyiciye bildirilmiştir. Bundan sonra Grafik3D isim alanında bulunan türlere sınıfın tam adını belirtmeden direkt erişebiliriz.

```

using System;
using Grafik3D;
namespace Grafik3D
{
    public class Nokta
    {
        public int x;
        public int y;

        public Nokta(int x,int y)
        {
            this.x = x;
            this.y = y;
        }
    }

    class AnaSınıf
    {
        static void Main()
        {
            Nokta nokta = new Nokta(5,6);
            Console.WriteLine(nokta.x);
        }
    }
}

```

Yukarıdaki programda,

using Grafik3D;

deyimi ile Grafik3D isim alanında bulunan bütün türler kaynak kodumuzdaki varsa-  
ylan isim alanına eklenir.

using deyimleri herhangi bir nesne tanımlaması veya isim alanı bildirimi yapılmadan  
kaynak koda eklenmelidir. Örneğin aşağıdaki kullanım hatalıdır.

```
using System;
namespace Grafik3D
{
    public class Nokta
    {
        public int x;
        public int y;

        public Nokta(int x,int y)
        {
            this.x = x;
            this.y = y;
        }
    }

    using Grafik3D; // Geçersiz using deyimi kullanımı

    class AnaSınıf
    {
        static void Main()
        {
        }
    }
}
```

Aynı isim alanı birkaç defa using kullanılarak kaynak koda eklenirse derleyici tarafından hata olarak kabul edilmez. Bu durum özellikle iç içe (nested) geçmiş isim alanlarında görülür. Mesela aşağıdaki gibi bir kullanım derleme zamanında hataya yol açmaz.

```
using System;
using Grafik2D;
using Grafik3D;

using anahtar sözcüğünün diğer bir kullanımı ise belirlenen bir bloğun sonunda nes-  
nelerin Dispose() metodunu çağrırmaktır. using anahtar sözcüğünü bu şekilde kullanı-  
bilmek için IDisposable arayüzünün ilgili sınıf tarafından işlenmiş olması gereklidir.  
Henüz arayüzlerle ilgili konuyu işlememiş olmamıza rağmen using anahtar sözcüğü-  
nün bu şekilde kullanımına bir örnek vermek istiyorum. Aşağıdaki programda using  
bloğunun sonunda d nesnesinin Dispose() metodu çağrılacaktır.
```

Deneme sınıfının IDisposable arayüzünden türemiş olması kafanızı karıştırmasın. Siz sadece Main() metodunun içine dikkat edin.

```
using System;
class Deneme : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("Dispose() metodu çağrıldı");
    }
}

//Bizim için önemli olan bundan sonrasıdır.

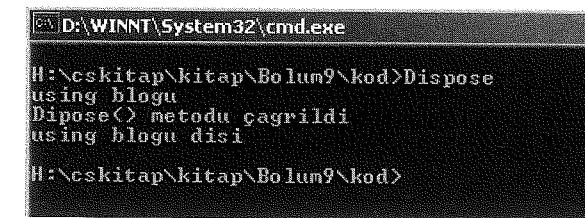
class AnaSınıf
{
    public static void Main()
    {
        Deneme d = new Deneme();

        using (d)
        {
            Console.WriteLine("using bloğu");

            } //d.Dispose() metodu çağrılır.

            Console.WriteLine("using bloğu dışı");
        }
    }
}
```

Bu programı derlediğimizde aşağıdaki ekran görüntüsünü alırız.



Gördüğünüz gibi using bloğunun hemen sonunda Dispose() metodu çağrıldı.

using bloğunu aşağıdaki gibi de yazabilirdik.

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

    } //d1.Dispose() ve d2.Dispose() metodu çağrılır.
```

Bu şekildeki bir kullanımında, d1 ve d2 nesnelerine using bloğunun dışından erişeme-  
ziz. Çünkü d1 ve d2 nesneleri using bloklarında tanımladıkları için bir üst faaliyet  
alanında görünmezler.

Yukarıdaki kullanımını test etmek için aşağıdaki programı yazıp derleyin.

```
using System;
class Deneme : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("Dispose() metodu çağrıldı");
    }
}

class AnaSınıf
{
    public static void Main()
    {
        using (Deneme d1 = new Deneme(), d2 = new Deneme())
        {
            Console.WriteLine("using bloğu");

            //d1.Dispose() ve d2.Dispose() metodu çağrırlar.

            Console.WriteLine("using bloğu dışı");
        }
    }
}
```

Bu programı derleyip çalıştırduğumızda ise aşağıdaki ekran görüntüsünü elde ederiz.

```
H:\cskitap\kitap\Bolum9\kod>Dispose2
using bloğu
Dispose() metodu çağrıldı
Dispose() metodu çağrıldı
using bloğu dışı
H:\cskitap\kitap\Bolum9\kod>
```

Bir using parantezi içinde sadece bir türden nesnenin tanımlaması yapılabilir. Buna göre Deneme1 ve Deneme2 ayrı birer sınıf olmak üzere aşağıdaki kullanım geçersizdir.

```
using (Deneme1 d1 = new Deneme1(), Deneme2 d2 = new Deneme2())
```

Diğer bir ilginç nokta aşağıdaki kullanımın da geçersiz olmasıdır.

```
using (Deneme d1 = new Deneme(), Deneme d2 = new Deneme())
```

## using ile Takma İsim (Alias) Verme

Daha önce Grafik2D ve Grafik3D isim alanlarını bildirdik. Her iki isim alanında Nokta isimli sınıflar bildirmiştik. Bu iki isim alanını aşağıdaki gibi programımıza eklediğimizde

```
using Grafik2D;
using Grafik3D;
```

varsayılan isim alanında iki tane Nokta isimli sınıf olmasına rağmen derleme zamanında bir hata almayız. Ancak Nokta türünden bir nesne tanımlamaya çalıştığımızda hata alırız. Aşağıdaki programı yazıp derleyin.

```
using System;
using Grafik3D;
using Grafik2D;

namespace Grafik3D
{
    public class Nokta
    {
        public Nokta()
        {
            Console.WriteLine("Nokta3D");
        }
    }
}

namespace Grafik2D
{
    public class Nokta
    {
        public Nokta()
        {
            Console.WriteLine("Nokta2D");
        }
    }
}

class AnaSınıf
{
    static void Main()
    {
        Nokta n = new Nokta(5, 6);
    }
}
```

Gördüğünüz gibi derleme sırasında “Nokta’ is an ambiguous reference” yani “Nokta belirsiz bir referanstr” hatası alırız. Derleyiciye hak vermemeğ elde değil. Çünkü

```
Nokta n = new Nokta(5, 6);
```

deyimi ile Grafik3D isim alanında Nokta sınıfı mı yoksa Grafik2D isim alanında Nokta sınıfı mı kastediliyor belli değil. Bu yüzden aynı isimli türlerin aynı isim alanında bulunması belirsizliğe yol açar. Peki, bu durumda ne yapacağız? İki şekilde çözebiliriz bu sorunu: Birincisi nesne tanımlarken tam isim kullanmak yani

```
System.Grafik2D.Nokta n = new System.Grafik2D.Nokta(5, 6);
```

ya da

```
System.Grafik3D.Nokta n = new System.Grafik3D.Nokta(5, 6);
```

şeklinde kullanmak. Bu kullanımı daha önceden görmüştük. İkinci bir yöntem ise her Nokta sınıfı için ayrı bir takma isim belirlemek, mesela Grafik2D isim alanında Nokta sınıfı için Nokta2D ismini kullanmak, Grafik3D isim alanında Nokta sınıfı için de Nokta3D ismini kullanmaktır. Bir sınıfa takma isim vermek için **using** anahtar sözcüğü aşağıdaki gibi kullanılır.

```
using Nokta2D = Grafik2D.Nokta;
using Nokta3D = Grafik3D.Nokta;
```

Artık Nokta2D dediğimizde Grafik2D isim alanında Nokta sınıfından bahsettiğimiz anlaşılacak. Aynı şey Nokta3D içinde geçerlidir. Şimdi **using** anahtar sözcüğünün bu şekildeki kullanımına bir örnek verelim.

```
using System;
using Grafik3D;
using Grafik2D;

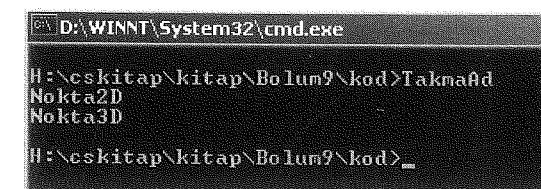
using Nokta2D = Grafik2D.Nokta;
using Nokta3D = Grafik3D.Nokta;

namespace Grafik3D
{
    public class Nokta
    {
        public Nokta()
        {
            Console.WriteLine("Nokta3D");
        }
    }
}

namespace Grafik2D
{
    public class Nokta
    {
        public Nokta()
        {
            Console.WriteLine("Nokta2D");
        }
    }
}

class AnaSinif
{
    static void Main()
    {
        Nokta2D nokta2D = new Nokta2D();
        Nokta3D nokta3D = new Nokta3D();
    }
}
```

Programı çalıştırıp derlediğinizde aşağıdaki ekran görüntüsünü elde etmeniz gerekir.



Takma ad kullanımını .NET sınıf kütüphanesindeki sınıflar üzerinde de uygulayabiliriz. Örneğin Console sınıfının ismini yazmak size zor geliyorsa aşağıdaki programda olduğu gibi tek harfli bir takma isim kullanarak yazdığınız kodlardaki harf sayısını azaltabilirsiniz.

```
using System;

using K = System.Console;

class AnaSinif
{
    public static void Main()
    {
        K.WriteLine("Takma isim");
    }
}
```

## İç içe (nested) Geçmiş İsim Alanları

Büyük çaplı yazılım projelerinde (.NET sınıf kütüphanesi gibi) isim alanlarında farklı isim alanları bildirilebilir. Örneğin System isim alanındaki Collections isim alanı gibi. İç içe geçirilen isim alanları `:` operatörü ile **using** anahtar sözcüğü kullanılarak kaynak koda eklenebilir. Aşağıda buna bir örnek göremektesiniz.

```
using System;
namespace Alan1
{
    class A
    {
        public A(){}
    }
}

namespace Alan2
{
    class B
    {
        public B(){}
    }
}

class AnaSinif
```

```

{
    public static void Main()
    {
        Alan1.A a = new Alan1.A();
        Alan1.Alan2.B b = new Alan1.Alan2.B();
    }
}

```

Yukarıdaki program ile aşağıdaki eşdeğerdir. using anahtar sözcükleri ile Alan1 ve Alan2 isim alanları eklenerek Main() metodun içerisinde A ve B sınıflarının tam isimleri kaldırılmıştır.

```

using System;
using Alan1;
using Alan1.Alan2;

namespace Alan1
{
    class A
    {
        public A(){}
    }

    namespace Alan2
    {
        class B
        {
            public B(){}
        }
    }
}

class AnaSınıf
{
    public static void Main()
    {
        A a = new A();
        B b = new B();
    }
}

```

İç içe geçmiş iki isim alanı bildirme yerine, aşağıdaki gibi ? operatörünü kullanarak tek bir isim alanı da bildirebiliriz.

```

namespace Alan1
{
    namespace Alan2
    {
    }
}
yerine

```

```

namespace Alan1.Alan2
{
}

```

şeklinde bir bildirim yapabiliriz.

## External Alias (Harici Takma İsimler)

İsim alanları konusunda daha önce bahsedildiği gibi farklı isim alanları altında aynı isimli sınıflar tanımlandığında ve her iki isim alanı da using deyiği ile eklendiğinde kod içinden her birine erişebilmek için sınıflara ya da diğer tiplere **takma isimler (alias)** verebiliyorduk. Böylece referans edilmiş her bir tipe takma isimleriyle erişebiliyorduk.

Oluşturulan tiplere takma isimler vererek anlam bütünlüğünü bozmak iyi bir çözüm sayılmaz. Ayrıca sadece tiplere takma isim vermek bütün çıkışmalarımızı kısa yoldan çözmeyecektir. Bu yüzden C# 2.0'da itibaren C# dilinde **dll kütüphanelerine (assembly)** harici takma isimler verebilmekteyiz. Aşağıdaki iki senaryoda **harici takma isimler (external alias)** son derece güzel bir çözüm sağlamaktadır.

**Senaryo 1:** Bir proje içerisinde aynı kütüphanenin (assembly, dll) farklı iki versiyonun kullanımını düşünelim. Yeni bazı durumlarda eski versiyonun kullanımını varsayıyoruz. Hem yeni hem de eski versiyonda bütün tipleri ve isim alanlarının aynı olduğunu ve sadece metodların farklı şekilde çalıştığını varsayıyalım. Bu durumda programımız içinden her iki kütüphanedeki tiplere takma isim vermeden erişemeyiz. Her bir tipe ayrı ayrı takma isim vermek de çok zahmetli olacağından bu yöntemi seçmiyoruz. Bu senaryoda harici takma isimler işimizi kolaylaştıracaktır.

**Senaryo 2:** Farklı iki firmadan benzer işler yapan iki kütüphane satın aldığımızı düşünelim. Örneğin GR3D firmasından 3 boyutlu çizim işlemleri yapan kütüphaneyi GR2D firmasından ise 2 boyutlu çizim işlemleri yapan kütüphaneyi satın aldığımızı düşünelim. Her kütüphanede de **GraphicsObjects** isimli bir isim alanının ve bu isim alanlarında da Nokta, DikDortgen gibi sınıfların olduğunu düşünelim. İsim alanları ve sınıflar aynı isimli olduğundan projemizde kullanım sırasında bir çakışma yaşanacaktır. Bu durumda da kütüphanelere harici takma isim vererek çakışmanın önüne geçebiliyoruz.

Örneğimizi 2. senaryo üzerinden yapalım. Öncelikle GR3D firmasının kütüphanesini oluşturalım. Eğer Visual Studio.NET kullanıyorsak yeni bir proje **Class Library** projesi açılmalıdır. Eğer komut satırı derleyicisini kullanıyorsak yeni bir .cs uzantılı dosya açarak aşağıdaki kodları sınıfı yazıyoruz.

GR3D firmasının kütüphanesi

```

// GR3D.cs dosyası.
using System;
namespace GraphicsObjects

```

```
{
    //3d Nokta
    public class Nokta
    {
    }

    //3d Dörtgen
    public class Dörtgen
    {
    }
}
```

Aynı şekilde GR2D firmasının kütüphanesinde GR2D.cs dosyasında ya da yeni bir **Class Library** projesi açarak yazalım.

GR2D firmasının kütüphanesi

```
// GR2D.cs dosyası.
using System;

namespace GraphicsObjects
{
    //2d Nokta
    public class Nokta
    {

    }

    //2d Dörtgen
    public class Dörtgen
    {
    }
}
```

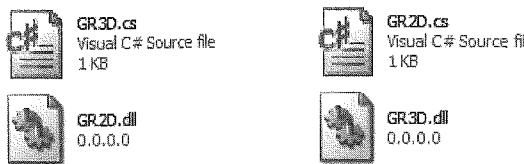
Her iki kütüphaneyi de dll haline getirmek için komut satırından aşağıdaki iki komutu ayrı ayrı çalıştırmanız gerekiyor.

```
csc /t:library GR3D.cs
csc /t:library GR2D.cs
```



Eğer Visual Studio.NET kullanıyorsanız Build menüsünden Build Solution sekmesini seçip derlemeniz yeterlidir.

Komut satırından yukarıdaki iki derleme işlemi yapıldığında dizin yapınız aşağıdaki gibi olmalıdır.



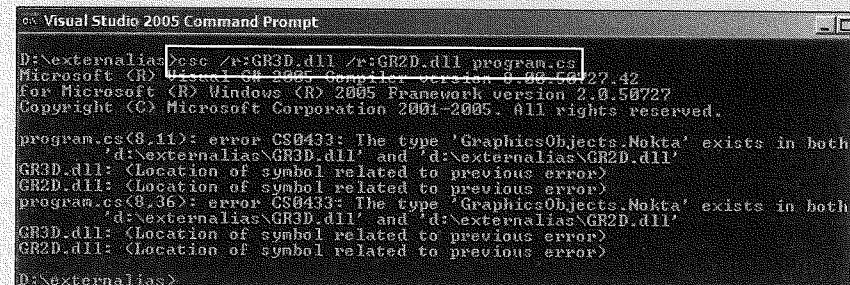
Yukarıda oluşturduğumuz her iki dll içerisinde de GraphicsObjects.Nokta ve GraphicsObjects.Dörtgen isimli sınıfların olduğunu hatırlatalım. Şimdi aşağıdaki konsol uygulamasında her iki kütüphaneyi de kullanmaya çalışalım.

```
//program.cs Dosyası
using System;
using GraphicsObjects;

class Program
{
    static void Main(string[] args)
    {
        Nokta noktanesnesi = new Nokta();
    }
}
```

Yukarıdaki programı **GR2D.dll** ve **GR3D.dll** kütüphanelerine aşağıdaki gibi referans vererek derlediğimizde Nokta tipi her iki kütüphanede olduğu için derlenmeyecektir.

Komut satırından (csc.exe ile) harici bir kütüphaneye (dll) referans vermek için /r isimli parametre kullanılır. Bu parametreden sonra dll'in adını yazmak yeterlidir. Visual Studio .NET ile referans vermek için projeye sağ tıklandıktan sonra Ad Reference sekmesine tiklandıktan sonra açılan pencereden Browse seçilir ve ilgili dll kütüphanesi bulunup OK düğmesine basılır.



Gördüğü gibi derleme işlemi gerçekleşmemektedir. Ancak dll kütüphanelerine C# 2.0'da eklenen **extern alias** (*harici takma isim*) verildiğinde derleme işlemi gerçekleştirilecektir. Kod içerisindeki hangi kütüphaneye erişeceğiz o kütüphaneye verdığımız takma ismi kullanacağız.

Derleme işlemi aşağıdaki gibi yapılarak **extern alias** (*harici takma isim*) verilmektedir. Tabi bu işlemin gerçekleşmesi için programımız içinde **using** bildirimlerinden önce takma isimleri tanımlamak gerekiyor.

C# 2.0'a harici takma isimleri tanımlamak için **extern alias** isimli bir anahtar sözcük eklenmiştir. Bu anahtar sözcük ile referans verecek kütüphanelere takma isim verilmektedir.

Önemli Not: **extern alias** anahtar sözcüğü bütün **using** deyimlerinden önce yazılmalıdır.

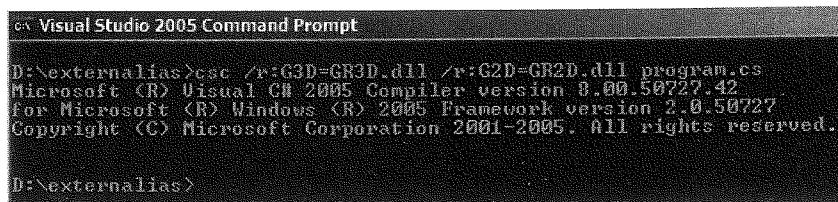


Buna göre *program.cs* dosyasını aşağıdaki düzenleyebiliriz.

```
//program.cs Dosyası (yeni)

extern alias G2D;
extern alias G3D;
using System;
class Program
{
    static void Main(string[] args)
    {
        //.....
    }
}
```

Komut satırından derleme işlemini yeni takma isimlere göre aşağıdaki gibi derleme-  
miz gerekiyor.



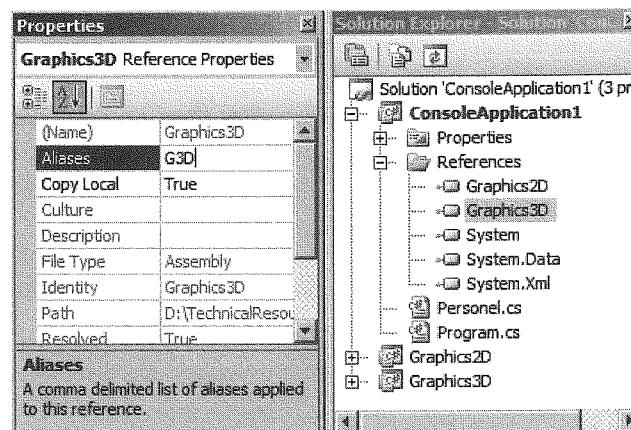
```
Visual Studio 2005 Command Prompt

D:\externalalias>csc /r:G3D=GR3D.dll /r:G2D=GR2D.dll program.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

D:\externalalias>
```

Böylece GR3D.dll kütüphanesindeki tipler için G3D takma ismini GR2D.dll kütüpha-  
nesindeki tipler için G2D takma ismini vermiş olduk.

Eğer **Visual Studio.NET** kullanıyorsanız external alias vermek için **References** sek-  
mesinde bulunan dll'lere tıkladıkten sonra **Properties Window**'dan **Aliases** özelliğini  
aşağıdaki gibi değiştirebilirsiniz. Bu işlemden sonra **Build** işlemi yapıldığında takma  
isim otomatik olarak verilecektir.



Pratikte çok fazla kullanılmasa da bir kütüphaneye birden fazla harici takma  
isimde verilebilir. Birden fazla harici takma isim vermek için takma isimleri virgül  
ile ayırmamız yeterlidir.



Takma isim verildikten sonra işlem başarılı bir şekilde gerçekleştiyse her iki Nokta ve  
Dortgen sınıflarına artık erişebiliriz. Ancak bu erişim klasik yollarla yapılmamaktadır.  
Aşağıda erişim yöntemini anlatılmaktadır.

## :: Operatörü

Harici takma isim verilmiş dll kütüphanelerindeki tip ve isim alanlarına erişmek için  
C# 2.0'a yeni bir operatör eklenmiştir. C++ programcılarının aşina olduğu operatör  
çift iki nokta operatördür. Yani :: operatörü sadece harici takma isim verilmiş kütüp-  
haneler ile birlikte kullanılır. Örneğin programımız içerisinde **extern alias** tanımı  
yapılduktan sonra aşağıdaki gibi Nokta ya da Dortgen sınıflarına erişilebilir.

```
G2D::GraphicsObjects.Nokta n = new G2D::GraphicsObjects.Nokta();
G2D::GraphicsObjects.Dortgen n = new G2D::GraphicsObjects.Dortgen();
```

```
G3D::GraphicsObjects.Nokta n = new G3D::GraphicsObjects.Nokta();
G3D::GraphicsObjects.Dortgen n = new G3D::GraphicsObjects.Dortgen();
```

Aynı şekilde **GraphicsObjects** isimli isim alanında **using** ile aşağıdaki gibi bildirebiliriz.

```
using G2D::GraphicsObjects;
```

ya da

```
using G3D::GraphicsObjects;
```

## global Harici Takma İsmi

.NET içerisinde referans verilmiş herhangi bir kütüphaneye siz farkında olmasa-  
nın bile otomatik olarak bir takma isim verilir. (Tabii ki isterseniz değiştirebilirsiniz)  
Bu takma ismin adı **global**'dır. Herhangi bir C# programını derlediğinizde **System.dll**  
gibi birçok temel kütüphane otomatik olarak referans verildiğinden bu kütüphanalere  
aynı zamanda otomatik olarak global isminde harici bir takma isim verilir.

**global** isimli harici takma ismine kod içinden

```
global::System.Math = new global::System.Math();
```

şeklinde erişmek mümkündür.



**global** kelimesi aynı zamanda bir anahtar sözcüktür.

global harici takma ismi özellikle referans verilmiş bazı tiplerin başka sınıflarca ezilmesi veya isim benzerliğinden dolayı görünmesinin engellenmesi durumunda oldukça kullanışlıdır. Gördüğünüz gibi esnek bir şekilde isim alanı belirleme açısından harici takma isimler oldukça önem kazanmaktadır.



**global**, **partial** gibi C# diline sonradan eklenen kelimeler anahtar sözcükler olmasına rağmen geriye dönük uyumluluktan ötürü bu anahtar sözcükler herhangi bir değişken ismi olarak kullanılabilir. Örneğin `int global=5;` şeklinde bir değişken tanımlamak C# 2.0 da geçerlidir.

## System İsim Alanı

Bu kısımda .NET sınıf kütüphanesinde System isim alanında bulunan birtakım faydalı işler yapan önemli sınıfları inceleyeceğiz. Bu sınıflardan dizilerle ilgili işlemler yapan Array sınıfını ve rastgele sayı üretme işine yarayan Random sınıfını daha önce incelemiştik. Ayrıca metodlar konusunda da matematiksel işlemler yapmamızı sağlayan birtakım statik metodlar içeren Math sınıfını incelemiştik. Şimdi ise C#'ta tarih ve saat işlemlerini yapmak için hazırlanmış DateTime ve TimeSpan yapısını, gereksiz nesne toplayıcısı ile ilgili olan GC sınıfını, değişkenlerin bitleri ile ilgili işlemler yapmaya yarayan BitArray sınıfını, türler arasında dönüşümler yapan Convert sınıfını ve bellek işlemlerinde kullanılan Buffer sınıfını yakından inceleyeceğiz. Ayrıca temel veri türlerinin CTS'deki karşılığı olan yapıları da yakından inceleyeceğiz. String sınıfını ise bir sonraki bölümde detaylı bir şekilde inceleyeceğiz.

## Temel Tür Yapıları

Bölüm 1'de debynmiş olduğumuz C# temel veri türlerinin aslında System isim alanında bulunan çeşitli yapı türlerini temsil ettiğini söylemişlik. Yani int, bool, char gibi anahtar sözcüklerinin birer takma isimden farkı yoktur. O halde asıl bizim System isim alanında bulunan bu veri yapılarını incelememiz gereklidir.

Aşağıda her bir temel türde System isim alanında karşılık gelen yapılar verilmiştir.

C#'taki Takma Adı	System İsim Alanındaki Veri Yapısı
bool	Boolean
char	Char
sbyte	SByte
byte	Byte

ulong	UInt64
uint	UInt32
ushort	UInt16
long	Int64
int	Int32
short	Int16
float	Single
double	Double
decimal	Decimal

Şimdi sırası ile bu yapı nesnelerinin özelliklerini ve metodlarını görelim. Önce tam-sayı türlerinden başlayalım. Tamsayı türleri Byte, SByte, Int16, Int32, Int64, UInt16, UInt32 ve UInt64 yapılarıdır. Bu yapıların tamamında MaxValue ve MinValue adında iki tane statik üye eleman vardır. Bu elemanlar ilgili yapı türünün tutabileceği en büyük ve en küçük sayıyı tutar. Örneğin, UInt64 yapısının tutabileceği en büyük ve en küçük sayıyı ekrana yazdırınmak için aşağıdaki deyimleri yazabiliriz.

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

Tamsayı yapıları ile ilgili önemli metot Parse() metodudur. Parse metodu bir string değişkeninde, ilgili tür ile ilgili karakterleri bulur ve geri dönüş değeri olarak C#'taki temel veri türü karşılığına döner. Örnek olarak aşağıdaki programı inceleyin.

```
using System;
class AnaSinif
{
    public static void Main()
    {
        string a = "56";
        int m = Int32.Parse(a);
        Console.Write(m);
    }
}
```

Int32.Parse() metodunun geri dönüş değeri int türünden olduğu için rahatlıkla int türünden bir değişkene atayabildik. Burada dikkat etmemiz gereken nokta string değişkeninin düzgün formatlı olmasıdır. Mesela aşağıdaki deyimleri içeren bir program derlenir ancak çalışma zamanında hata verdirir.

```
string a = "ab56";
int m = Int32.Parse(a);
```

Aynı şekilde Double yapısındaki Parse() metodu da string türünden bir nesneyi double türüne döndürür. Bu diğer yapı türleri içinde geçerlidir. Tamsayı veri yapıları ile ilgili diğer önemli metodlar CompareTo(), Equals() ve ToString() metodlarıdır.

`CompareTo()` metodunun prototipi aşağıdaki gibidir.

```
int CompareTo(object o)
```

Bu metot ile metodu çağrıran nesnenin değeri o nesnesinin değeri ile karşılaştırılır. Eğer değerler eşitse 0 değeri döndürülür, metodu çağrıran daha küçükse negatif, daha büyükse pozitif değer döndürülür.

`Equals()` metodunun prototipi aşağıdaki gibidir.

```
bool Equals(object o)
```

Bu metot, metodu çağrıran nesnenin değeri ile `o` nesnesinin değerini karşılaştırır. Değerler eşitse true, eşit değilse false değerini döndürülür.

`ToString()` metodunun prototipi aşağıdaki gibidir.

```
string ToString()
```

Bu metotla, metodu çağrıran nesnenin string şeklindeki değeri döndürülür. İşlevsel olarak `Parse()` metodununun tersidir diyebiliriz.



Bu metodların farklı imzalara sahip aşırı yüklenmiş olanları da vardır. Bu kitapta bunların hepsini tek tek anlatmak mümkün değildir. En çok kullanılanları öğrendiğiniz takdirde diğerlerini yardım dosyalarından rahatlıkla öğrenebilirsiniz.

Şimdi gerçek sayı türleri ile ilgili olan yapıları inceleyelim. Bu yapılar `Single` ve `Double` yapılarıdır. Bu yapılar tamsayı yapılarından farklı olarak aşağıdaki özelliklere sahiptir.

`Epsilon` —> En küçük pozitif değer (sıfırdan farklı)

`NaN` —> Herhangi bir sayıyı temsil etmeyen değer

`NegativeInfinity` —> Negatif sonsuz

`PositiveInfinity` —> Pozitif sonsuz

`.MaxValue` ve `.MinValue` bu yapılar içinde de mevcuttur.

`Double` ve `Single` yapıları `CompareTo()`, `ToString()`, `Equals()` ve `Parse()` metodlarının yanısıra aşağıdaki metodları da içerir.

```
bool static IsInfinity(float ya da double)
```

Bu metot, parametre olarak verilen float ya da double sayı sonsuzu temsil ediyorsa true, aksi halde false değerini döndürür.

```
bool static isNaN(float ya da double)
```

Bu metot, parametre olarak verilen float ya da double sayı, `NaN` (sayı olmayan) ise true, değilse false döndürür.

```
bool static IsPositiveInfinity(float ya da double)
bool static IsNegativeInfinity(float ya da double)
```

Bu metot, parametre olarak verilen float ya da double sayının pozitif ya da negatif sonsuz olup olmadığını kontrol eder. `IsPositiveInfinity()` metodu için eğer sayı pozitif sonsuz ise true, `IsNegativeInfinity()` metodu için sayı negatif sonsuz ise true değerini döndürür.

Şimdi de diğer önemli bir veri türü olan `char`'ın `System` isim alanında bulunan karşılığı olan `Char` yapısını inceleyelim.

`Char` yapısında diğer yapılarda olduğu gibi `.MaxValue` ve `.MinValue` özellikleri vardır. `Char` veri yapısında `CompareTo()`, `ToString()`, `Equals()` ve `Parse()` metodlarının yanı sıra aşağıdaki metodlar da bulunur.

```
public static double GetNumericValue(char ch)
```

`ch` nümerik bir karakter ise `ch`'ın nümerik değeri, aksi takdirde -1 değeri döndürür.

Aşağıda bir grup metod bulunmaktadır. Bu metodların hepsinin iki kullanımı vardır. Örneğin:

```
bool IsUpper(char ch);
```

ya da

```
bool IsUpper(string str, int indeks)
```

Birinci metod `ch` karakterinin ilgili koşulu sağlaması durumunda true değerini döndürür, ikinci metod ise `str` `string`'inin indeks nolu karakteri ilgili koşulu sağlıyorsa true değerini döndürür. Bu metodların isimleri ve koşulları aşağıda verilmiştir.

Metot Adı	Kosul
<code>IsControl</code>	Karakter kontrol karakteri ise
<code>IsDigit</code>	Karakter bir rakam ise
<code>IsLetter</code>	Karakter bir harf ise
<code>IsLetterOrDigit</code>	Karakter bir harf ya da rakam ise
<code>IsLower</code>	Karakter küçük harf ise
<code>IsNumber</code>	Karakter 16'lık tabanda bir sayı ise (0-9,A-F)
<code>IsPunctuation</code>	Karakter noktalama işaretleri ise
<code>IsSeparator</code>	Karakter boşluk gibi ayırıcı ise
<code>IsSurrogate</code>	Karakter UNICODE yedek karakteri ise
<code>IsSymbol</code>	Karakter sembol ise
<code>IsUpper</code>	Karakter büyük harf ise
<code>IsWhiteSpace</code>	Karakter tab ya da boşluk karakteri ise

Yukarıdaki bütün metodlar statik olduğu için Char ismi üzerinden erişilir. Karakter metodlarına bir örnek olması açısından aşağıdaki örneği inceleyin. Bu programda bir yazı içerisindeki bütün rakamlar ekrana yazdırılıyor.

```
using System;
class AnaSınıf
{
    public static void Main()
    {
        string a = "9:30 da bana 5 milyon verdi";
        for(int i = 0; i < a.Length; ++i)
            if(Char.IsDigit(a[ i ]))
                Console.Write(a[ i ]);
    }
}
```

Programı derlediğimizde ekrana

9305

yazdığını göreceksiniz. Bu programda bir yazının her bir karakterine [ ] operatörü ile eriştiğimize dikkat edin.

Char yapısının iki önemli metodu daha vardır. Bunlar ToUpper() ve ToLower() metodlardır. Bu metodlar küçük karakterleri büyük karakterlere, büyük karakterleri de küçük karakterlere dönüştürürler. Bu metodların prototipi aşağıdaki gibidir.

```
static char ToLower(char ch)
static char ToUpper(char ch)
```

Aşağıda kullanıcının girdiği bütün karakterleri büyük karakterlere çeviren bir program görmektesiniz.

```
using System;
class AnaSınıf
{
    public static void Main()
    {
        Console.Write("Yazıyı girin:");
        string yazi = Console.ReadLine();
        string BuyukYazi = "";
        for(int i=0; i<yazi.Length; ++i)
            BuyukYazi += Char.ToUpper(yazi[ i ]);
        Console.WriteLine("Buyuk yazı : " + BuyukYazi);
    }
}
```

Diğer bir veri yapısı da Boolean'dır. Boolean yapısı C#'taki bool türüne karşılık gelir. Boolean veri yapısında iki tane özellik vardır. Bu özellikler FalseString ve TrueString'tir. Bu özellikler de string türündendir, False ve True yazılarını içerirler. Örneğin aşağıdaki programı derleyip çalıştırıldığınızda ekrana

False

True

yazdığını göreceksiniz.

```
using System;
class AnaSınıf
{
    public static void Main()
    {
        Console.WriteLine(Boolean.FalseString);
        Console.WriteLine(Boolean.TrueString);
    }
}
```

Boolean yapısının kendisine has bir metodu yoktur. CompareTo(), Equals(), ToString() ve Parse() metodları Boolean yapısında olan diğer metodlardır.

C# dili diğer dillerden farklı bir tür barındırır. Bu decimal türüdür. decimal türü System isim alanında Decimal yapısı ile temsil edilir. Decimal veri türünün bellekte saklanması diğer türlere nazaran biraz daha karmaşıktr. Bu yüzden bilgisayarların decimal veri türleri ile iş yapması biraz daha zordur. Decimal türünden bir değişken birçok şekilde tanımlanabilir. Decimal bir sayı; yapıcı metodlar yardımıyla int, uint, long, ulong, float ya da double türden bir sayı ile tanımlanabilir. Diğer bir önemli tanımlama şekli ise aşağıdaki yapıcı metot ile tanımlamaktır.

```
Decimal(int alt, int orta, int ust, bool işaret, byte ondalık)
```

Decimal türü bildiğiniz gibi 128 bitlik bir türdür. Bu bitlerin bir tanesi sayının işaretini belirtir; 96 biti sayının tamsayı kısmını, kalanlar ise ondalık kısmını belirtir. Yukarıdaki yapıcı metotta birinci parametre ile 96 bitlik sayının ilk 32 biti, ikinci parametre ortadaki 32 biti, üçüncü parametre ise son 32 biti temsil etmektedir. Dördüncü parametre false ise sayı pozitif, true ise negatiftir. Son parametre ise 96 bitlik sayının kaçıncı basamağından sonra virgül içerdigini gösterir. Örneğin, 9865,74 sayısını aşağıdaki gibi oluşturabiliriz.

```
Decimal d = new Decimal(986574, 0, 0, false, 2);
```

Burada ikinci ve üçüncü parametrelerin sıfır olduğunu dikkat edin. Eğer bunlar sıfır olmasaydı, tahmin edemeyeceğimiz sonuçlar çıkardı. Örneğin, ikinci ve üçüncü parametreleri 1 yaparak oluşan değeri aşağıdaki gibi ekrana yazdırın.

```
using System;
class AnaSınıf
{
    public static void Main()
    {
        Decimal d = new Decimal(986574,1,1,false,2);
        Console.WriteLine(d);
    }
}
```

Programı derleyip çalıştırduğumız da aşağıdaki sayının yazıldığını göreceksiniz.

184467440780055054,86

Oldukça büyük bir sayı değil mi?

Decimal yapısında CompareTo(), Equals(), ToString() ve Parse() metodlarının yanı sıra decimal sayısını diğer bütün türlere dönüştürebilmek için çeşitli statik metodlar bildirilmiştir. Bu metodlar ToDouble(),ToInt16(),ToInt32() gibi sıralanabilir. Her bir metod C#'taki ilgili türden bir nesne ile geri döner. Örneğin,ToInt32() metodunun geri dönüş değeri int türünden bir nesnedir. Aynı şekilde ToSingle() metodunun geri dönüş değeri float türünden bir nesnedir. Bu metodların her birini kullanıp sonuçlarını görmeyi tavsiye ederim.

Bu dönüşüm metodlarının yanısıra iki decimal sayı üzerinde dört işlem yapan metodlar da bildirilmiştir. Bu metodlar aşağıdaki gibidir.

```
static decimal Add(decimal d1, decimal d2) // sonuç d1 + d2 dir.
static decimal Divide(decimal d1, decimal d2) // sonuç d1 / d2
dir.
static decimal Multiply(decimal d1, decimal d2) // sonuç d1 * d2
dir.
static decimal Subtract(decimal d1, decimal d2) // sonuç d1 - d2
dir.
static decimal Remainder(decimal d1, decimal d2) // sonuç d1 %
d2 dir.
```

Decimal yapısının diğer önemli metodları da aşağıda verilmiştir.

1. static decimal Floor(decimal d);

Bu metod ile d'den büyük olmayan en büyük sayı döndürür. Örneğin, d sayısı 5,65 ise Floor() metodunun geri dönüş değeri 5 olacaktır.

2. static int[] GetBits(decimal d);

Bu metod d sayısının 5 parametreli yapıcısı içindeki elemanları bir dizi içerisinde aktarıp dizi ile geri döner.

3. static decimal Negate(decimal d);

Bu metod ile d sayısının negatifini döndürür. Örneğin, d sayısı 89,7 ise Negate() metodunun geri dönüş değeri -89,7 olacaktır.

4. static decimal Round(decimal d, int sayı);

Bu metod ile d sayısı sayı ile belirtilen basamak hassaslığı kadar yuvarlatılır. Eğer sayı 0 ise decimal sayı tamsayıya yuvarlatılmış olur. Örneğin:

```
using System;
class AnaSınıf
{
    public static void Main()
    {
        Decimal d1 = new Decimal(986576,0,0,false,3);

        Console.WriteLine(Decimal.Round(d1,2));
    }
}
```

bu programı derleyip çalıştırduğumızda ekrana

986,58

yazdığını görürüz.

5. static decimal Truncate(decimal d);

Bu metod d sayısının tamsayı kısmını geri dönüş değeri olarak döndürür.

## Tarih ve Zaman İşlemleri

C#'ta tarih ve saat işlemleri System isim alanında bulunan DateTime ve TimeSpan yapıları ile gerçekleştirilmektedir. DateTime yapısı gün, ay, yıl, saat, dakika ve saniye gibi bilgileri tutmak için kullanılırken TimeSpan ise iki tarih arasındaki farkı temsil etmek için kullanılır.

Önce DateTime yapısının statik üye elemanlarını inceleyelim. Öncelikle DateTime yapısının içinde tutabileceğimiz en büyük ve en küçük tarih bilgisinin ne olduğunu bakanım. Bunun için aşağıdaki programı yazın.

```
using System;
class AnaSınıf
{
    public static void Main()
    {
        Console.WriteLine("En küçük : " + DateTime.MinValue);
        Console.WriteLine("En büyük : " + DateTime.MaxValue);
    }
}
```

Programı derleyip çalıştırduğumızda aşağıdaki ekran görüntüsünü elde ederiz.

```
D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum9\kod>DateTime
En küçük : 01.01.0001 00:00:00
En büyük : 31.12.9999 23:59:59
H:\cskitap\kitap\Bolum9\kod>
```



**.MaxValue** ve **.MinValue**, **DateTime** yapısının **readonly** üye elemanlarıdır.

**DateTime** yapısının önemli ve en çok kullanılan iki özelliği **Now** ve **Today**'dır. **Now** özelliği o anki sistemin saat ve tarih bilgilerini verir. **Today** özelliği ise o anki tarih bilgisini verir. **Today** özelliği ile saat bilgileri 0 olarak ayarlanır. (00:00:00) **Now** ve **Today** özelliklerinin her ikisi de **DateTime** yapısı türündendir.

Aşağıdaki deyimleri içeren bir programı çalıştırığınızda

```
Console.WriteLine(DateTime.Now);
Console.WriteLine(DateTime.Today);
```

aşağıdaki ekran görüntüsüne benzer bir görüntü elde edersiniz.

```
22.02.2002 17:39:24
22.02.2002 00:00:00
```

Aynı işlemleri aşağıdaki gibi, bir **DateTime** nesnesi oluşturarak da yapabiliyoruz.

```
DateTime TarihSaat = new DateTime();
TarihSaat = DateTime.Now;
DateTime Tarih = new DateTime();
Tarih = DateTime.Today;
Console.WriteLine(TarihSaat);
Console.WriteLine(Tarih);
```

**DateTime** yapısının kullanabileceğimiz çeşitli güzel özellikleri vardır. Bütün bu özellikler aşağıda verilmiştir. Açıklamalardaki parantez içinde bulunan türler, özelliğin türünü belirtmektedir.

Date	<b>DateTime</b> nesnesine ilişkin saat dışındaki bilgiyi verir. ( <b>DateTime</b> )
Month	<b>DateTime</b> nesnesinin ay bilgisini verir. (int)
Day	<b>DateTime</b> nesnesinin gün bilgisini verir. (int)
Year	<b>DateTime</b> nesnesinin yıl bilgisini verir. (int)

DayOfWeek	<b>DateTime</b> nesnesinin haftanın kaçinci günü olduğunu verir (DayOfWeek numaralandırması)
DayOfYear	<b>DateTime</b> nesnesinin yılın kaçinci gününe denk geldiğini verir. (int)
TimeOfDay	Saat 00:00:00 dan itibaren ne kadar zaman geçtiğini TimeSpan nesnesi olarak verir. (TimeSpan)
Hour	<b>DateTime</b> nesnesinin saat bilgisini verir (int)
Minute	<b>DateTime</b> nesnesinin dakika bilgisini verir (int)
Second	<b>DateTime</b> nesnesinin saniye bilgisini verir (int)
Millisecond	<b>DateTime</b> nesnesinin milisaniye bilgisini verir (int)
Ticks	<b>DateTime</b> nesnesindeki tarih ile 1 Ocak 0001, 00:00:00 tarihi arasındaki 100 nanosaniyelik periyotların sayısını verir (long)

Bir **DateTime** nesnesini aşağıdaki metotlar ile oluşturabiliriz. Bunların dışında da yapanıcı metotlar bulunmaktadır.

```
DateTime dt = new DateTime(long tick_sayisi);
DateTime dt = new DateTime(int yıl, int ay, int gün);
DateTime dt = new DateTime(int yıl, int ay, int gün,int saat,int dakika,int saniye);
DateTime dt = new DateTime(int yıl, int ay, int gün,int saat,int dakika,int saniye,int milisaniye);
```

**TimeSpan** yapısı iki **DateTime** nesnesinin arasındaki farkı temsil eden bir yapıdır. İki **DateTime** nesnesini birbirinden çıkarırsak sonuç bir **TimeSpan** nesnesidir. Aşağıdaki programda bir kullanıcının doğum tarihinden bugüne kadar geçen gün sayısını ve doğduğu günü buluyoruz.

```
using System;
class Tarih
{
    static void Main()
    {
        int yıl,ay,gun;
        Console.WriteLine("Doğum yılı: ");
        yıl = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Doğum ayı: ");
        ay = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Doğum günü: ");
        gun = Convert.ToInt32(Console.ReadLine());
```

```

DateTime Bugun = DateTime.Today;
DateTime DogumTarihi = new DateTime(yil,ay,gun);

TimeSpan fark = Bugun - DogumTarihi;

Console.Write("Doğduğunuz gün:");
Console.WriteLine(DogumTarihi.DayOfWeek);

Console.Write("Geçen gün sayısı: ");
Console.WriteLine(fark.Days);
}
}

```

Programı derleyip çalıştırıldığınızda aşağıdakine benzer bir görüntü elde edersiniz.

```

D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum9\kod>Dogum
Dogum yili:
1978
Dogum ayi:
10
Dogum gunu:
25
Dogdugunuz gun:Wednesday
Geçen gün sayısı: 8521
H:\cskitap\kitap\Bolum9\kod>

```



Girilen ay, gün ve yıl değerleri en büyük tarihten büyüğe çalışma zamanında hata alırız. Bu yüzden değerlerin doğru girilmesi gereklidir.

TimeSpan ve DateTime yapılarında tanımlanmış birçok metot bulunmaktadır. Bu metotlar genellikle TimeSpan ve DateTime yapıları arasındaki ilişkiye dayanmaktadır. Örneğin, bir DateTime nesnesi ile TimeSpan nesnesini toplayabiliriz. Aşağıdaki programda belirtilen gün kadar sonrasının hangi güne denk geldiğini bulan program mevcuttur.

```

using System;
class Tarih
{
    static void Main()
    {
        int GunSayisi;
        Console.Write("Gun sayısı : ");
        GunSayisi = Convert.ToInt32(Console.ReadLine());

        TimeSpan Fark = new TimeSpan(GunSayisi,0,0,0);

        DateTime dt = DateTime.Today + Fark;
    }
}

```

```

        Console.WriteLine(dt.DayOfWeek);
    }
}

```

Yukarıdaki programı çalıştırıldığınızda bilgisayarınızdaki gün Pazar ise ve eğer gün sayısını 8 olarak girerseniz ekrana Monday (Pazartesi) yazacaktır.

TimeSpan ve DateTime yapılarında bildirilmiş birtakım operatör metotları da mevcuttur. Örneğin, iki tarihin büyülük ve küçüklüğünü karşılaştırmak için > ve < operatörleri aşırı yüklenmiştir. Bu yapıların diğer metotları ve nasıl kullanıldıkları bu kitabın konusu olmadığı için daha fazla ayrıntıya girmeyeceğiz. .NET Framework sınıf kütüphanesindeki sınıflara ve yapılara ait metotları ve özelliklerini

<http://msdn.microsoft.com>

web adresinden sınıfın ya da yapının ismini aratarak rahatlıkla bulabilirsiniz.

**Bir Örnek:** Verilen iki tarih arasında geçen hafta sonu günlerinin sayısını bulan bir program yazınız.



İki DateTime nesnesinin farkı alınıp sonuç bir TimeSpan nesnesine aktarılacak. Daha sonra TimeSpan nesnesindeki gün sayısını bir for döngüsü ile tek tek küçük olan tarihe eklenip yeni tarihin Cumartesi ya da Pazar gününe denk gelip gelmediği kontrol edilecek.

Program aşağıdaki gibidir.

```

using System;
class Tarih
{
    static void Main()
    {
        DateTime Tarih1 = new DateTime(2001,5,2);
        DateTime Tarih2 = new DateTime(2003,5,9);

        TimeSpan Fark = Tarih2 - Tarih1;
        long HaftaSonuGunu = 0;

        DateTime Gecici;

        for(int i = 0; i <= Fark.Days ; ++i)
        {
            Gecici = Tarih1.AddDays(i);
            if(Gecici.DayOfWeek==DayOfWeek.Sunday ||

```

```

        Gecici.DayOfWeek==DayOfWeek.Saturday)
    {
        HaftaSonuGunu++;
    }

    Console.WriteLine("Gün sayısı:" + HaftaSonuGunu);
}
}

```

## BitConverter Sınıfı

Özellikle alt seviye programlama yaparken genellikle verilerin byte dizisi şeklinde olması istenir. Bu hem hız sağlamakta hem de kolaylık getirmektedir. Bitsel operatörleri kullanarak istediğimiz veri türünü byte dizileri şeklinde ifade edebilmemiz mümkün, ancak .NET sınıf kütüphanesinde bizim için bu işi yapacak bir sınıf (BitConverter sınıfı) bulunmaktadır.

BitConverter sınıfında bir tane özellik bulunmaktadır. Bu özellik

`IsLittleEndian`

özellidir. Bu özellik ile sistemimizin verileri hangi mimariye göre sakladığını kontrol ederiz. İki byte'lik bir veri düşünün. Bu veri bellekte iki şekilde bulunabilir. Yüksek anlamlı byte'ı (**MSB - most significant byte**) ilk sırada ya da düşük anlamlı byte (**LSB - least significant byte**) ilk sırada olabilir. Eğer düşük anlamlı byte ilk sırada ise bu mimariye "Little Endian" denilmektedir. Tam tersi durumu destekleyen mimariye de "Big Endian" denilmektedir.

`IsLittleEndian` özelliği ile eğer mimari "Little Endian" ise true, değilse false değeri üretilir. Intel'in pentium işlemcileri "Little Endian" mimarisine göre çalışmaktadır. Aşağıdaki programla sisteminizin "Little Endian" mı yoksa "Big Endian" mimarisini mi desteklediğini öğrenebilirisiniz.

```

using System;
class bitconverter
{
    static void Main()
    {
        if(BitConverter.IsLittleEndian)
            Console.WriteLine("Little Endian");
        else
            Console.WriteLine("Big Endian");
    }
}

```

`IsLittleEndian` özelliğinin yukarıdaki kullanımından dolayı statik olduğunu anlıyoruz.



BitConverter sınıfının en önemli metodu, statik `GetBytes()` metodudur. Bu metot ile bool, char, int, long, double, short, uint, ulong ve ushort türünden değişkenlerin değerini byte türünden bir diziye aktarabiliriz. Bu metodun prototipi aşağıdaki gibidir.

```
static byte[ ] GetBytes(değişken)
```

Aşağıdaki programda 19785 sayısının içeriği byte'lar tek tek ekrana yazdırılıyor.

```

using System;
class bitconverter
{
    static void Main()
    {
        int a = 19785;

        byte[ ] aByte = BitConverter.GetBytes(a);

        foreach(byte i in aByte)
            Console.Write(i + " ");
    }
}

```

Programı derleyip çalıştırıldığımızda ekrana

73 77 0 0

yazdığını görürüz. Ekran görüntüsünden de anlaşıldığı gibi düşük anlamlı byteler dizinin ilk elemanları olacak şekilde düzenlenmiştir.

BitConverter sınıfının diğer bir metot grubu ise bir byte dizisini temel veri türlerine dönüştüren metotlardır. Bu metotların tamamı statik oldukları için BitConverter ismi üzerinden erişilir. Bu metotlar parametre olarak bir byte dizisi ve int türden bir sayı alırlar.

Örneğin,

```
static intToInt32(byte[ ] dizi , int index)
```

metodu `dizi[index]` elemanından başlayarak 4 byte'ı int türüne dönüştür ve sonucu geri döndürür. Aynı şekilde `ToInt64()` metodu, belirtilen elemandan itibaren 8 byte'ı long türüne çevirip geri döndürür. Bu metotlar `ToBoolean()`, `ToChar()`, `ToInt16()`, `ToInt32()`, `ToInt64()`, `ToSingle()`, `ToDouble()`, `ToUInt16()`, `ToUInt32()` ve `ToString()` olarak sıralanabilir.

## Convert Sınıfı

Convert sınıfındaki statik metotlar tür dönüştürme için belki de en çok kullanılan metotlardır. Convert sınıfında BitConverter sınıfındaki metodların hepsi bulunur. Convert sınıfı ile veri kaybı olmayacağı şekilde bütün türler arasında dönüşüm ya-

pilabilmektedir. Temel sayı türleri ile ilgili dönüştürmeler zaten ya bilinçli şekilde ya da bilinçsiz şekilde yapılabiliyordu. Bu yüzden Convert sınıfı şu ana kadar da sıkça kullandığımız gibi string türünden değişkenleri sayı türlerine dönüştürmek için kullanılır. Özellikle kullanıcıdan Console.ReadLine() metodu ile alınan yazıları hızlıca tamsayıya dönüştürmek için kullanabiliriz.

Aşağıdaki programda Console.ReadLine() metodu ile alınan string türünden bir değişken int türüne dönüştürülüyor.

```
using System;
class Convert_sınıfı
{
    static void Main()
    {
        string a = Console.ReadLine();
        int b = Convert.ToInt32(a);
        Console.WriteLine(b);
    }
}
```

## Buffer Sınıfı

Buffer sınıfı ile tür bilgisinden bağımsız biçimde byte düzeyinde işlemler yapılır. Örneğin bir dizinin belirli alanının başka bir diziye kopyalarken tür bilgisine bakılmaması gibi. Buffer sınıfını türden bağımsız veriler üzerinde çalışırken kullanabiliriz. Buffer sınıfının aşağıda verilen metodları bulunmaktadır.

1. static void BlockCopy(  
    Array kaynak,  
    int kaynak\_index,  
    Array hedef,  
    int hedef\_index,  
    int adet)

Bu metod kaynak dizisinin kaynak\_index numaralı elemanından itibaren adet kadar byte'ı hedef dizisinin hedef\_index elemanından sonrasına kopyalar. Bu işlem tamamen türden bağımsızdır. Bu işlemin nasıl bir sonuç verdiği için aşağıdaki programı yazınız.

```
using System;
class buffer
{
    static void Main()
    {
        byte[ ] kaynak = { 1,2,3,1 };
        short[ ] hedef = new short[ 10 ];

        Buffer.BlockCopy(kaynak, 0, hedef, 0, 4);

        foreach (short i in hedef)
```

```
        Console.WriteLine(i + " ");
    }
}
```

Bu programı çalıştırduğumuzda ekrana

513 259 0 0 0 0 0 0 0

yazdığını göreceksiniz. Peki ne oldu da blok kopyalama sonucunda hedef dizinin sadece iki elemanı sıfırdan farklı ve neden değerleri değişti? Şimdi bu soruların cevabını bulalım.

Öncelikle short türünün 2 byte olduğunu belirtelim. Byte türü de bildiğiniz gibi 1 byte'tır. BlockCopy() metodu kopyalama işlemini byte düzeyinde yaptığı için kaynak dizinin her iki elemanı hedef dizisinde bir elemana denk düşecektir. Örneğin kaynak[0] ile kaynak[1], hedef[0] elemanını oluşturacaktır. Şimdi kaynak dizisinin elemanlarını yan yana koyarak 513 ve 259 sayılarının nasıl elde edildiğine bakalım.

hedef[0] → kaynak[1] kaynak[0]

0 0 0 0 0 1 0 0 0 0 0 0 0 1 → 513  
(2) (1)

hedef[1] → kaynak[3] kaynak[2]

0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 → 259  
(1) (3)

2. static int ByteLength(Array dizi)

Parametre ile verilen bir dizi içindeki toplam byte sayısını verir. Örneğin, 10 elemanlı short türden bir dizi parametre olarak ByteLength() metoduna gönderilirse sonuç olarak  $10 \times 2 = 20$  değeri döndürür.

```
short[ ] dizi = new short[ 10 ];
Console.WriteLine(Buffer.ByteLength(dizi));
```

3. static byte GetByte(Array dizi, int index)

dizi[index] elemanından itibaren ilk byte değerini döndürür. GetByte() metodunun etkisini en rahat byte türünden diziler üzerinde görebiliriz. Aşağıdaki programda byte türünden bir dizinin byte'ları GetByte metodu ile alınmaktadır.

```
using System;
class buffer
{
    static void Main()
    {
        byte[ ] dizi = { 0,1,2,3,4,5,6 };
```

```

        Console.WriteLine(Buffer.GetByte(dizi,5));
    }
}

```



Yukarıdaki byte dizisini int yaptığımızda dizinin elemanları ile aynı değeri elde edemeyiz.

#### 4. static void SetByte(Array dizi, int index, byte deger)

SetByte() metodu dizi[index] elemanından itibaren bir byte'lık alana deger sayısını yazar.

## GC (Garbage Collector) Sınıfı

Daha önce de belirtildiği gibi .NET platformunda bulunan gereksiz bilgi toplayıcısı (garbage collector) mekanızması, programınızın herhangi bir anında kullanılmayan referansları heap alanından siler. GC mekanızmasının ne zaman devreye gireceği kesin olarak bilinmediği için hangi nesnelerin ne zaman sonlandırılacağı da bilinmez. GC mekanızması bellek alanını optimum tutacak şekilde kendini ayarlar. .NET'in bize sunduğu GC sınıfı ile istedigimiz bir anda garbage collection mekanızmasının devreye girmesini isteyebiliriz. Bunun için GC sınıfının statik Collect() metodunu aşağıdaki gibi çağrılmamız yeterli olacaktır.

```
GC.Collect();
```

Programın herhangi bir anında o ana kadar tahsis edilmiş toplam bellek alanının byte cinsinden görmek için GC sınıfının GetTotalMemory(true) metodunu kullanılır. Bu metodun geri dönüş değeri byte türündendir. Eğer parametre true olarak girilirse değer döndürülmeden önce GC mekanızması ile anlamsız veriler yok edilir. Eğer parametre false ise hiçbir şey yapılmadan o anki toplam tahsis edilmiş alana geri dönülür.

## Temel I/O (Girdi/Çıktı) ve String İşlemleri

- C# I/O Sistemi
- Dosya ve Klasör İşlemleri
  - Directory Sınıfı
  - File Sınıfı
  - DirectoryInfo Sınıfı
  - FileInfo Sınıfı
  - Path Sınıfı
- Dosya Yazma ve Okuma İşlemleri
  - FileStream Sınıfı
  - FileStream ile Yazma ve Okuma
  - Dosya Akımı ile Text İşlemleri Yapmak
  - StreamReader Sınıfı
  - StreamWriter Sınıfı
- BinaryWriter ve BinaryReader Sınıfları
- Console I/O İşlemleri
  - Standart Akımların Yönlendirilmesi
- Temel String (Karakter Dizisi) İşlemleri
  - String Tanımlama
  - String Metotları
  - Arama İşlemleri
  - Budama ve Doldurma İşlemleri
  - Split() ve Join() Metotları
  - Diğer String İşlemleri
- Yazıları Biçimlendirme
  - String.Format() ve ToString() Metotları ile Biçimlendirme
  - Tarih ve Saat Biçimlendirme
  - Özel Biçimlendirme Oluşturma
- Düzenli İfadeler (Regular Expressions)
  - Düzenli İfadelerin Oluşturulması
  - Düzenli İfadelerin Gruplanması

Bir programlama dilini tasarlayanlar için en zor nokta, iyi bir I/O sistemi hazırlamaktır. Çünkü I/O sistemi ile birçok kaynağa erişebilmek mümkün olmalıdır. Örneğin standart giriş dediğimiz klavye, standart çıkış olan ekran ya da diskler I/O sistemi için birer malzemeden. I/O kelimesi Input/Output sözcüklerinden gelir. Türkçe'ye ise Girdi/Çıktı olarak çevrilmiştir. Ancak biz temel bir isimlendirme olduğu için I/O şeklinde kullanacağız.

Bu bölümde C# ile temel metin işlemlerini, standart girdi/çıkıtı işlemlerini yapan çeşitli sınıfları ve bu sınıflar arasındaki ilişkiyi yakından inceleyeceğiz.

Bu bölümün diğer konusu ise yazı (String) işlemleridir. Stringlere karakter katarı ya da karakter dizileri de denilmektedir. Karakter katarlarını formatlama ve düzenli ifadeler gibi konular da bu bölümde yakından inceleneciktir.

## C# I/O Sistemi

C#'ta I/O sistemi ile ilgili bütün sınıflar System.IO isim alanı altındadır. I/O sistemi C#'ta stream dediğimiz akımlar üzerine kuruludur. Akımlar bir girdi ya da çıktı sisteminde byte düzeyinde bilgiyi okuyan bir soyut birimdir. Yani akım ile dosyadan bilgi okunabilecegi gibi standart giriş olan klavyeden de bilgi okuyabiliriz. Kısaca bilgi akışı varsa burada akımdan bahsetmek mümkündür. Akımlar bilgileri standart input dediğimiz klavyeden ya da diskten alabilirler. Bütün bu akım kaynakları birbirlerine yönlendirilebilir. Örneğin, konsol ekranı dosya sistemine yönlendirilerek Console.WriteLine() metodu ile yazdıklarımızın ayını herhangi bir dosyaya yazdırılabilir. Standart olarak çıktı konsol ekranı olduğu için Console.WriteLine() metodu bilgileri ekrana yazmaktadır. Standart giriş çıkış aygıtlarının birbirlerine nasıl yönlendirildiklerini bölümün ilerleyen kısımlarında göreceğiz. Ancak şimdilik standart akımların Console.Out (standart çıkış ya da ekran), Console.In (standart giriş ya da klavye) ve Console.Error (standart hata akımı ya da ekran) olduğunu bilmenizde fayda var.

C#'ta akımlarla ilgili işlemler System.IO isim alanında bulunan Stream sınıfı ile yapıılır. Stream sınıfı bir akımın destekleyebileceği akıma yazma, akımdan okuma gibi minimum özellikleri barındırır. Diğer özel akım sınıfları da Stream sınıfından türeyerek özelleştirilmiştir. FileStream, MemoryStream ve BufferedStream akım sınıfları bunlara örnek olarak verilebilir. Stream sınıfının ön bilgi olması açısından aşağıdaki metodlara sahip olduğunu söyleyebiliriz.

1. int Read(byte[] buf, int index, int byteSayısı)

Bu metot ile buf[index]’ten itibaren byteSayısı kadar byte bilgi, akımdan buf dizisine okunur.

2. int ReadByte()

Bu metot ile akımdan bir byte okunur ve int olarak döndürülür.

3. void Write(byte[] buf, int index, int byteSayısı)

Bu metot ile buf[index]’ten itibaren byteSayısı kadar byte bilgi akıma yazılır.

4. void WriteByte(byte b)

Bu metot ile akıma b ile gelen bir byte yazılır.

5. long Seek(long ilerleme\_sayısi, SeekOrigin konum)

Bu metot ile akımın konumu, SeekOrigin(End, Begin, Current) numaralandırması ile belirtilmiş konumundan ilerleme\_sayısi kadar ileriye ötelenir.

6. void Flush()

Bu metot ile akımla ilgili tamponlanmış bilgiler silinir ve akımdaki bilgiler fizikal birime ilettilir.

7. void Close()

Bu metot ile akım kapatılır. Stream.Close() çağrılarında ilgili akıma ait kaynaklar iade edilir.

Bir Stream sınıfına ait metodlar bu kadar değildir. Yukarıdaki metodlar en çok kullanlanlardır. Diğer metodlarla ilgili bilgiler bu kitabın konusu dışındadır. Bu metodların yanı sıra Stream sınıfının aşağıdaki özellikleri de mevcuttur:

**bool CanRead:** Bu özellik akımdan okuma yapılabiliyorsa true değerine sahiptir.

**bool CanSeek:** Bu özellik eğer akım konumlandırılmayı destekliyorsa true değerine sahiptir.

**bool CanWrite:** Bu özellik akıma yazma yapılabiliyorsa true değerine sahiptir.

**long Length:** Akımın uzunluğunu verir.

**long Poistion:** Akımın o anki konumunu verir.

Stream Sınıfı ile ilgili bu bilgileri verdikten sonra temel dosya işlemlerini incelemeye başlayalım.

## Dosya ve Klasör İşlemleri

Dosyalar üzerinde okuma ve yazma işlemleri yapan sınıfları incelemeden önce, dosya ve klasörler ile ilgili işlemleri yapan Directory, File, Path, FileInfo, ve DirectoryInfo sınıflarını inceleyeceğiz. Bu sınıflar ile dosyalar ve klasörler üzerinde kopyalama, isim değiştirme gibi temel işlemleri yapabilmekteyiz. Bu sınıfların I/O sistemi ile çok fazla ilgisi olmamasına rağmen, genellikle I/O işlemleri sırasında bu gibi özellikler sıkça kullanıldığı için ilk önce bu konuları ele almaktan fayda var.

FileInfo ve File sınıfları, dosya sistemindeki dosyaları temsil ederken, Directory ve DirectoryInfo sınıfları klasörleri temsil etmektedir. Path sınıfı ise dosyaların ve klasörlerin yol (**path**) bilgileri ile ilgili işlemler yapmak için kullanılır.

Directory ve File sınıfları, sadece statik metodlar içerir. Yani herhangi bir nesne tanımlamadan bütün metodlarına erişebiliriz. Genellikle sadece bir dosya ya da bir klasör üzerinde tek bir işlem yapacaksak bu sınıfları kullanabiliriz. Klasör içindeki dosyalar ile ya da belirli bir dosya grubu ile ilgili ortak işlemler yapacaksak FileInfo ve DirectoryInfo sınıflarını kullanabiliriz.

DirectoryInfo ve FileInfo sınıflarındaki metodların hemen hepsi DirectoryInfo ve File sınıfında bulunmaktadır. Ancak DirectoryInfo ve FileInfo sınıfları ile belirli bir klasörün ya da dosyanın, dosya sistemindeki özelliklerini içeren bilgileri elde edebiliriz. Örneğin bir dosyanın en son ne zaman düzenlendiği gibi.

Bu kısımda dosyalar ve klasörler ile ilgili işlemler yaparken genellikle DirectoryInfo ve FileInfo sınıflarını kullanacağız. Bazı durumlarda ise Directory ve File sınıflarını kullanacağız.

Örnek işlemlere geçmeden önce her bir sınıfa ait metodları ve özellikleri tek tek inceleyelim.

## Directory Sınıfı

Directory sınıfı System.IO isim alanında bulunan ve bütün metodları statik olan klasörlerle ilgili işlemler yapmamızı sağlayan sınıfıdır. Directory sınıfının bütün üye elemanları metodlardır. Hiçbir özelliği bulunmamaktadır. Aşağıda bu metodları ve kullanımları ile ilgili bilgiler verilmektedir.

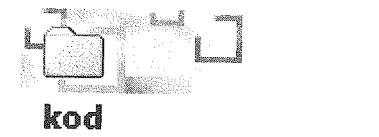
### 1. DirectoryInfo.CreateDirectory(string yol)

yol değişkeni ile belirtilen adreste bir klasör oluşturur ve bu klasörün bilgilerini içeren DirectoryInfo sınıfı türünden nesne ile geri döner. Programınızın çalıştığı klasörün içinde yeni bir klasör açmak için path bilgisi olarak klasörün ismini vermek yeterlidir. Örneğin, aşağıdaki programı çalıştırıldığımızda programın bulunduğu klasörün içinde "YeniKlasor" isminde yeni klasör açılmış olacaktır.

```
using System;
using System.IO;

class Klasor
{
    static void Main()
    {
        Directory.CreateDirectory("YeniKlasor");
    }
}
```

Programımızın bulunduğu klasörde aşağıdaki gibi yeni bir klasör oluşturulmuş olmalıdır.



Select an item to view its description.

See also:

[My Documents](#)

[My Network Places](#)

[My Computer](#)

C dizinin altında bu klasörü açmak isteseydik, yol bilgisini; "yol= @"c:\YeniKlasor" şeklinde vermemiydi.

### 2. void Delete(string yol)

Belirtilen yoldaki klasörleri silmek için kullanılır. İki biçim vardır:

```
void Delete(string yol)
void Delete(string yol, bool a)
```

Birinci metot ilgili yoldaki boş olan bir klasörü silmek için, ikinci metot ise eğer ikinci parametre true ise ilgili yoldaki klasörün içindeki dosya ve klasörlerle beraber tamamen silinmesi için kullanılır.

### 3. bool Exists(string yol)

Belirtilen yolda klasörün bulunup bulunmadığını true ya da false olarak bildirir. Klasör varsa true değeri döndürür.

### 4. DateTime GetCreationTime(string yol)

Belirtilen yoldaki klasörün hangi tarihte oluşturulduğunu bir DateTime nesnesi olarak döndürür.

### 5. string GetCurrentDirectory()

Çalışan programın hangi klasörde olduğunu verir.

### 6. string[] GetDirectories(string yol)

Belirtilen yoldaki bütün klasörlerin isimlerini string türünden bir dizi ile döndürür. Aşağıdaki programda bu metodun kullanımına örnek verilmiştir. Program D dizinindeki bütün klasörleri ekrana yazdırmaktadır.

```
using System;
using System.IO;
```



```

class Klasorler
{
    static void Main()
    {
        string[] klasorler = Directory.GetDirectories(@"D:\");
        foreach(string i in klasorler)
            Console.WriteLine(i);
    }
}

7. string GetDirectoryRoot(string yol)

```

Belirtilen klasörün bulunduğu kök dizin bilgisini ve volume bilgilerini string olarak verir.

8. string[] GetFiles(string yol)

Belirtilen yoldaki klasörün içinde bulunan bütün dosyaları string türünden diziye aktarır ve bu diziyi döndürür. Bu metot iki şekilde kullanılabilir. Ancak en çok kullanılan şekli yukarıdaki gibidir:

9. string[] GetFileSystemEntries (string yol)

Belirtilen yoldaki klasörün içinde bulunan bütün dosyaları ve klasörleri string türünden diziye aktarır ve bu diziyi döndürür. Dosyalar ve klasörler Dosya Sisteminde bulunma sırasına göre diziye yerleştirilir.

10. DateTime GetLastAccessTime(string yol)  
DateTime GetLastWriteTime(string yol)

Bu metotlar sırası ile belirtilen yoldaki klasöre en son ne zaman erişildiğini ve dosyanın üzerinde en son ne zaman yazma işleminin yapıldığı bilgisini, DateTime nesnesi olarak döndürür.

11. string[] GetLogicalDrives()

Bilgisayardaki bütün sürücülerini string türünden bir diziye aktarıp diziyi döndürür.

12. DirectoryInfo GetParent(string yol)

Belirtilen yoldaki klasörün bir üst dizininin bilgilerini içeren DirectoryInfo nesnesi ile geri döner. Eğer yol olarak "C:\WINNT\System" verilmişse metodun geri dönüş değeri "C:\WINNT" klasörü ile ilgili bilgileri içeren nesnedir.

13. void Move(string kaynak\_yol, string hedef\_yol)

kaynak\_yol ile belirtilmiş olan dosya tamamen hedef\_yol ile belirtilmiş olan yere taşınır.

14. DateTime SetLastAccessTime(string yol, DateTime zaman)  
DateTime SetLastWriteTime(string yol, DateTime zaman)  
DateTime SetCreationTime(string yol, DateTime zaman)

Bu metodlar sırası ile belirtilen yoldaki klasörün en son erişilen zamanını, klasöre en son yazılma tarihini ve klasörün oluşturulma tarihini ikinci parametre ile verilen zaman değişkenine göre yeniden düzenler.

15. void SetCurrentDirectory(string yol)

Programın çalıştığı klasörü yol ile belirtildiği gibi değiştirir.

Bütün yol ifadelerinde tam yol gibi göreceli yol da bulunabilir. Göreceli yol o an programın bulunduğu klasöre göre belirlenir.



## File Sınıfı

File sınıfının bazı metodları Directory sınıfının metodları ile aynıdır. Tek farkı klasörler yerine dosyalar üzerinde işlem yapmasıdır. Bu ortak metodlar aşağıda verilmiştir.

Exists(), Delete(), GetCreationTime(), GetLastAccessTime(), GetLastWriteTime(), Move(), SetCreationTime(), SetLastAccessTime() ve GetLastWriteTime().

Bu ortak metodların yanısıra File sınıfı dosyalar üzerinde özelleşmiş işlemler yapabilmesi için aşağıdaki metodları barındırır.

1. StreamWriter AppendText(string yol)

yol'da belirtilen dosya için daha sonra göreceğimiz bir StreamWriter nesnesi döndürür.

2. void Copy(string kaynak, string hedef)

kaynakta belirtilen dosya hedefe kopyalanır. Hedefte kaynak ile aynı isimli bir dosya var ise işlem başarısız olur. Bunun için Copy() metodunun aşağıdaki gibi de kullanabiliriz.

void Copy(string kaynak, string hedef, bool ustuneyaz)

Eğer üçüncü parametre true ise, kaynak ile aynı isimli bir dosya zaten mevcutsa üzerine yazılır. Parametre false ise birinci metodun kullanımı ile aynı olur.

3. FileStream Create(string yol)  
FileStream Create(string yol, int tampon)

Belirtilen yoldaki dosya oluşturulur ve dosyaya ilişkin FileStream nesnesi döndürülür. tampon değişkeni verilmemişse varsayılan tampon miktarı ayarlanır.

4. StreamWriter CreateText(string yol)

Belirtilen yolda üzerine yazmak için bir text dosyası oluşturulur ve ilgili dosya ilişkin StreamWriter sınıfı referansı döndürülür.

5. FileAttributes GetAttributes(string yol)

Belirtilen yoldaki dosyanın FileAttributes numaralandırması ile özelliği döndürülür. FileAttributes numaralandırması aşağıdaki sembollerini içerir:

Archive, Compressed, Device, Directory, Encrypted, Hidden, Normal,  
NotContentIndexed, Offline, ReadOnly, ReparsePoint, SparseFile,  
System, Temporary

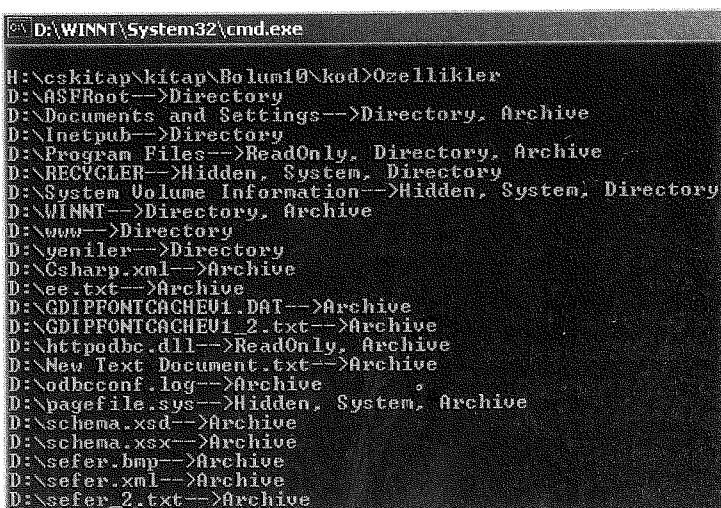
Aşağıdaki program D sürücüsü içindeki bütün dosya ve klasörlerin özelliklerini ekrana yazdırır.

```
using System;
using System.IO;

class DizinVeDosyaOzellikleri
{
    static void Main()
    {
        string[] dizi = Directory.GetFileSystemEntries(@"D:\\");

        foreach(string i in dizi)
        {
            Console.Write(i + "-->");
            Console.WriteLine(File.GetAttributes(i));
        }
    }
}
```

Ben programı çalıştırduğumda aşağıdaki ekran görüntüsünü elde ettim. Sizde buna benzer bir görüntü elde etmelisiniz.



6. FileStream Open(string vol, FileMode a)

```
c) FileStream Open(string yol, FileMode a, FileAccess b, FileShare c)
```

`Open()` metodu belirtilen yoldaki dosyayı açar ve ilgili dosyaya ilişkin `FileStream` nesnesini döndürür.  `FileMode`,  `FileAccess` ve  `FileShare`, `System.IO` isim alanından bulunan numaralandırmalarıdır. Bunlar dosyanın ne şekilde açılacağını ve dosya üzerinde ne şekilde işlem görüleceğini belirtirler.

**FileMode Numaralandırmasında Bulunan Semboller**

**Append:** Açılan dosyanın sonuna ekleme yapmak için kullanılır. Eğer dosya bulunmazsa dosya oluşturulur.

**Create:** Yeni bir dosya oluşturmak için kullanılır. Eğer belirtilen dosya varsa üzerine yazılır.

**CreateNew:** Yeni dosya oluşturmak için kullanılır, ancak belirtilen dosya mevcutsa çalışma zamanında hata verilir.

**Open:** Dosyayı açmak için kullanılır.

**OpenOrCreate:** Belirtilen dosya varsa acılır yoksa yenişi oluşturulur.

**Truncate:** Belirtilen dosya açılır ve içi tamamen silinir.

#### **FileAccess Numaralandırmasında Bulunan Semboller**

**Read:** Dosya okumak için açılır.

**ReadWrite:** Dosya okunmak ve yazılmak üzere açılır.

**Write:** Dosya sadece yazmak için açılır.

## **FileShare Numaralandırmasında Bulunan Semboller**

**Inheritable:** Dosyanın child (yavru) prosesler tarafından türetilmesini sağlar.

**None:** Dosyanın başka prosesler tarafından açılmasını engeller.

**Read:** Dosyanın başka proseslerce de açılabilmesini sağlar.

**ReadWrite:** Dosyanın başka proseslerce de açılıp okunabilmesini ve üzerine yazılabilmesini sağlar.

**Write:** Dosyaya başka proseslerin de yazabilmesini sağlar.

#### 7. FileStream OpenRead(string yol)

Belirtilen dosyayı yalnızca okumak için açar ve ilgili dosyaya ilişkin FileStream nesnesini döndürür.

#### 8. StreamReader OpenText(string yol)

Belirtilen dosyayı yalnızca text modunda okumak için açar ve ilgili dosyaya ilişkin –daha sonra göreceğimiz– StreamReader nesnesini döndürür.

#### 9. FileStream OpenWrite(string yol)

belirtilen dosyayı yazma modunda açar ve ilgili dosyaya ilişkin FileStream nesnesini döndürür.

## DirectoryInfo Sınıfı

DirectoryInfo sınıfı tek bir dizin ile ilgili bilgileri elde etmek için kullanılır. Aşağıdaki program D:\WINNT\System32 dizini ile ilgili bütün özellikleri ekran'a yazdırır. Siz bu örneği başka dizinler ile test ederek DirectoryInfo sınıfının özelliklerini inceleyebilirsiniz.

```
using System;
using System.IO;

class DirectoryInfoSınıfı
{
    static void Main()
    {
        string yol = @"D:\WINNT\System32";

        DirectoryInfo d = new DirectoryInfo(yol);

        Console.Write("Özellikler : ");
        Console.WriteLine(d.Attributes);

        Console.Write("Olusturulma Tarihi : ");
        Console.WriteLine(d.CreationTime);

        Console.Write("Klasör varmı? : ");
        Console.WriteLine(d.Exists);

        Console.Write("Uzantı : ");
        Console.WriteLine(d.Extension);
        Console.Write("Tam Yol : ");
        Console.WriteLine(d.FullName);
```

```
Console.WriteLine("Son erişim zamanı : ");
Console.WriteLine(d.LastAccessTime);
```

```
Console.WriteLine("Son yazma zamanı : ");
Console.WriteLine(d.LastWriteTime);
```

```
Console.WriteLine("Klasör adı : ");
Console.WriteLine(d.Name);
```

```
Console.WriteLine("Bir üst klasör : ");
Console.WriteLine(d.Parent);
```

```
Console.WriteLine("Kök dizin : ");
Console.WriteLine(d.Root);
```

}

Programın ekran görüntüsü aşağıdaki gibidir.

```
D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum10\kod>DirectoryInfo
Özellikler : Directory, Archive
Olusturulma Tarihi : 13.07.2002 18:19:56
Klasör varmı? : True
Uzantı :
Tam Yol : D:\WINNT\System32
Son erişim zamanı : 23.02.2002 15:39:08
Son yazma zamanı : 23.02.2002 10:59:24
Klasör adı : System32
Bir üst klasör : WINNT
Kök dizin : D:\

H:\cskitap\kitap\Bolum10\kod>
```

Klasörlerin uzantısı olmadığı için d.Extension ile ekran'a bir şey yazdırılmadığına dikkat edin.



Şimdi de DirectoryInfo sınıfının metodlarını inceleyelim. Bu metodlar statik olmadığı için metodlara nesne üzerinden erişilmelidir.

#### 1. void Create()

Bu metod ile klasör oluşturulur. Örneğin, D sürücüsünde deneme isimli bir klasör aşağıdaki gibi oluşturulabilir.

```
using System;
using System.IO;

class DizinOlustur
{
```

```

static void Main()
{
    string yol = @"D:\deneme";
    DirectoryInfo d = new DirectoryInfo(yol);
    d.Create();
}

```

**2. DirectoryInfo CreateSubdirectory(string yol)**

Belirtilen yolda bir alt dizin oluşturur. Örneğin D:\deneme dizinin altında \deneme2\ deneme3 diye bir dizin oluşturmak için aşağıdaki deyimleri yazmalıyız.

```

using System;
using System.IO;

class AltDizinOlustur
{
    static void Main()
    {
        string yol = @"D:\deneme";
        DirectoryInfo d = new DirectoryInfo(yol);
        d.Create();

        DirectoryInfo alt = d.CreateSubdirectory("deneme2");

        alt.CreateSubdirectory("deneme3");
    }
}

3. void Delete()
void Delete(bool a)

```

Birinci metot ile eğer klasörün içi boşsa klasör silinir. İkinci metotta ise eğer parametre true olarak girilirse klasörde bulunan her şey silinir.

**4. DirectoryInfo[] GetDirectories()**

İlgili klasörde bulunan bütün dizinleri DirectoryInfo dizisi halinde döndürür. Aşağıdaki programda bu metodun kullanımına bir örnek görebilirsiniz. Programda D sürücüsündeki bütün dizinlerin adı ekrana yazdırılıyor. Siz bu programı daha da geliştirerek dizinlerle ilgili diğer özellikleri de yazdırabilirsiniz.

```

using System;
using System.IO;

class AltDizinGoster
{
    static void Main()
    {
        string yol = @"D:\";

```

```

DirectoryInfo d = new DirectoryInfo(yol);

DirectoryInfo[] dizinler = d.GetDirectories();

foreach(DirectoryInfo dir in dizinler)
    Console.WriteLine(dir.Name);
}

```

**5. FileInfo[] GetFiles()**

İlgili klasörde bulunan bütün dosyalar FileInfo türünden diziye yerleştirilir ve bu dizi döndürülür. Çalışma şekli GetDirectories() metodu ile aynıdır. Değişen tek şey klasörlerle değil de dosyalar ile ilgili iş yapmasıdır.

**6. FileSystemInfo[] GetFileSystemInfos()**

İlgili klasördeki bütün dosyalar ve klasörler FileSystemInfo türünden bir diziye aktarır ve bu dizi döndürülür. FileSystemInfo sınıfı FileInfo ve DirectoryInfo sınıfının türüne sınıftır. Bu sınıf pek fazla kullanılmamaktadır.

**7. void MoveTo(string hedef)**

İlgili dizin, içindeki dizin ve dosyalarla beraber, hedef ile belirtilen yere taşınır. Örneğin deneme klasörü aşağıdaki gibi yeni klasörüne taşınabilir.

```

string yol = @"D:\deneme";
DirectoryInfo d = new DirectoryInfo(yol);
d.MoveTo(@"D:\yeni");

```

Hedef ile kaynak klasörler aynı sürücüde olmalıdır. Bu metot ile klasörleri farklı sürüclere taşımak mümkün değildir.

**8. void Refresh()**

İlgili klasörün özelliklerini dosya sisteminden (diskten) tekrar yükler.

## FileInfo Sınıfı

FileInfo sınıfı tek bir dosya ile ilgili özellikleri içerir. FileInfo sınıfının özellikleri aşağıda toplu olarak verilmiştir. Bunun için yeni bir text dosyası açın, dosyanın yolunu da kendinize göre değiştirip aşağıdaki programı yazın.

```

using System;
using System.IO;

class DosyaBilgileri
{
    static void Main()
    {
        string yol = @"D:\deneme.txt";

```

```

FileInfo dosya = new FileInfo(yol);

Console.WriteLine("Özellikler : ");
Console.WriteLine(dosya.Attributes);

Console.WriteLine("Oluşturulma Zamanı : ");
Console.WriteLine(dosya.CreationTime);

Console.WriteLine("Dosya mevcut mu? : ");
Console.WriteLine(dosya.Exists);

Console.WriteLine("Uzantısı : ");
Console.WriteLine(dosya.Extension);

Console.WriteLine("Dosya tam adı : ");
Console.WriteLine(dosya.FullName);

Console.WriteLine("Son ulaşma zamanı : ");
Console.WriteLine(dosya.LastAccessTime);

Console.WriteLine("Son yazma zamanı : ");
Console.WriteLine(dosya.LastWriteTime);

Console.WriteLine("Boyut : ");
Console.WriteLine(dosya.Length);

Console.WriteLine("Dosya adı : ");
Console.WriteLine(dosya.Name);

Console.WriteLine("Bulunduğu klasör : ");
Console.WriteLine(dosya.DirectoryName);
}

```

Bu programın ekran görüntüsü aşağıdaki gibidir.

```

D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum10\kod>FileInfo
Özellikler : Archive
Olusturulma Zamani : 23.02.2002 16:26:22
Dosya mevcut mu? : True
Uzantisi : .txt
Dosya tam adı : D:\deneme.txt
Son ulaşma zamanı : 23.02.2002 16:36:16
Son yazma zamanı : 23.02.2002 16:26:32
Boyut : 37
Dosya adı : deneme.txt
Bulunduğu klasör : D:\

H:\cskitap\kitap\Bolum10\kod>

```

Şimdi de FileInfo sınıfının farklı olan metodlarını inceleyelim.

File sınıfında gördüğümüz AppendText(), Create(), CreateText(), Delete(), Open(), OpenRead(), OpenText(), OpenWrite() metotları FileInfo sınıfında mevcuttur. Bu metotlar aynı işleri yaptığı için tekrar anlatılmayacaktır. Bu metotlara ek olarak FileInfo sınıfı aşağıdaki metotları da içerir.

1. FileInfo CopyTo(string hedef)  
FileInfo CopyTo(string hedef, bool a)

İlgili dosya, 'hedef' ile belirtildiği şekilde kopyalanır. Eğer aynı dosya hedefte varsa birinci metot başarısız olur. Ancak ikinci metottaki ikinci parametre true olarak girilirse hedef dosya varsa bile üzerine yazılır.

2. void MoveTo(string hedef)

İlgili dosya hedefte belirtilen yere taşınır. Hedefteki dosya kaynak dosyanın isminden farklı da olabilir.

3. void Refresh()

İlgili dosya bilgileri dosya sisteminden tekrar alınarak FileInfo nesnesi yenilenir.

## Path Sınıfı

Path sınıfı string türünden aldığı bir yol bilgisi üzerinde çeşitli işlemler yapan statik üye metodlara sahiptir. Aşağıda bir yol bilgisi üzerinden bu metodların kullanımı gösterilmiştir.

```

using System;
using System.IO;

class PathSınıfı
{
    static void Main()
    {
        string yol = @"D:\dizin\deneme.txt";

        Console.WriteLine("Eski uzanti : ");
        Console.WriteLine(Path.GetExtension(yol));

        string yeni_yol = Path.ChangeExtension(yol, "jpg");

        Console.WriteLine("Yeni uzanti : ");
        Console.WriteLine(Path.GetExtension(yeni_yol));

        string yol2 = @"D:\klasör";

        Console.WriteLine("Yeni yol : ");
        Console.WriteLine(Path.Combine(yol, yol2));
    }
}

```

```

Console.WriteLine("Klasör : ");
Console.WriteLine(Path.GetDirectoryName(yol));

Console.WriteLine("Dosya adı : ");
Console.WriteLine(Path.GetFileName(yol));

Console.WriteLine("Uzantısız dosya adı : ");
Console.WriteLine(Path.GetFileNameWithoutExtension(yol));

Console.WriteLine("Tam yol : ");
Console.WriteLine(Path.GetFullPath(yol));

Console.WriteLine("Kök dizin: ");
Console.WriteLine(Path.GetPathRoot(yol));

Console.WriteLine("Geçici dosya adı : ");
Console.WriteLine(Path.GetTempFileName());

Console.WriteLine("Geçici dosya dizini : ");
Console.WriteLine(Path.GetTempPath());

Console.WriteLine("Dosya uzantısı var mı? : ");
Console.WriteLine(Path.HasExtension(yol));
Console.WriteLine("Alt dizin ayıracı : ");
Console.WriteLine(Path.AltDirectorySeparatorChar);

Console.WriteLine("Dizin ayıracı : ");
Console.WriteLine(Path.DirectorySeparatorChar);

Console.WriteLine("Geçersiz yol karakterleri : ");
Console.WriteLine(Path.InvalidPathChars);

Console.WriteLine("Yol ayırıcı karakter: ");
Console.WriteLine(Path.PathSeparator);

Console.WriteLine("Kök dizin ayıracı : ");
Console.WriteLine(Path.VolumeSeparatorChar);
}

```

Bu programı çalıştırığınızda aşağıdaki ekran görüntüsünü elde ederiz.

```

H:\cskitap\kitap\Bolum10\kod>Path.exe
Eski uzanti : .txt
Yeni uzanti : .jpg
Yeni yol : D:\klasör
Klasör : D:\dizin
Dosya adı : deneme.txt
Uzantısız dosya adı : deneme
Tam yol : D:\dizin\deneme.txt
Kök dizin: D:\ 
Geçici dosya adı : D:\DOCUME\1\ADMINI\1\ALGNLOCALES\Temp\tmp120.tmp
Geçici dosya dizini : D:\DOCUME\1\ADMINI\1\ALGNLOCALES\Temp\
Dosya uzantısı var mı? : True
Alt dizin ayıracı : \
Dizin ayıracı : \
Geçersiz yol karakterleri : <>*!@#$%^&*@
Yol ayırıcı karakter: ;
Kök dizin ayıracı : \
H:\cskitap\kitap\Bolum10\kod>

```

Path sınıfının en önemli metodu **GetTempFileName()** metodudur. Bu metot sisteminizin **temp** klasöründe oluşturduğu bir dosyanın ismini döndürür. Dosyalarla ilgili geçici işler yapmak istediğinizde bu методu kullanabilirsiniz.

Dosyalarla ilgili yazma ve okuma işlemeye geçmeden önce şu ana kadar öğrendiklerimizi uygulayabileceğimiz bir program yazalım. Bu program DOS işletimi sisteminin **dir** programına benzer bir şekilde çalışacak. Programı çalıştırımıza başladığımız anda DOS komut satırına benzer bir ekran gelecek. Bu anda **dir** komutunu yazarsak programın bulunduğu dizindeki dosyaları bilgileri ile beraber listeleyecek. Daha sonra yazdığımız her yol bilgisinde varsayılan klasör değişecek. **pwd** komutu ise o anda hangi dizinde olduğumuzu bize bildirilecek.

Programın akışı şu şekilde olacaktır:

Varsayılan dizin olarak programın bulunduğu dizin belirlenecek, sonsuz bir döngü içinde kullanıcıdan bir yazı alınacak. Eğer yazı 'q' ya da 'Q' ise programdan çıkışılacak. Yazı "dir" ise varsayılan dizindeki dosyalar ve klasörler bir metot yardımıyla ekrana yazılacak ve döngü başa dönecek. Bu metot string türünden bir yol bilgisi almalıdır. Eğer yazı "pwd" ise programın o anda varsayılan dizini ekrana yazılacak ve döngünün başına dönülecek (continue anahtar sözcüğü ile). Sonra girilen yazının bir klasör olup olmadığı kontrol edilecek, eğer klasörse varsayılan dizin değişkeni yenisiyle değiştirilecek, klasör değilse bir hata mesajı verilerek döngünün tekrar başına gelinecek. Programın kaynak kodu aşağıdaki gibidir.

```

using System;
using System.IO;

class Listeleme
{
    static void Main()
    {
        string Dizin = Directory.GetCurrentDirectory();

```

```

while(true)
{
    Console.Write("\nDizin Gir >> ");
    string yeni_dizin = Console.ReadLine().Trim();

    if(yeni_dizin.ToLower() == "q")
        break;
    if(yeni_dizin.ToLower() == "dir")
    {
        Listele(Dizin);
        continue;
    }
    if(yeni_dizin.ToLower() == "pwd")
    {
        Console.WriteLine("\n Dizin: {0}\n", Dizin);
        continue;
    }

    if(!Directory.Exists(yeni_dizin))
    {
        Console.WriteLine("\nYanlış yol...\n");
        continue;
    }
    else
    {
        Dizin = yeni_dizin;
    }

    Listele(Dizin);
}
}

public static void Listele(string Dizin)
{
    string[] Dosyalar = Directory.GetFiles(Dizin);
    string[] Dizinler = Directory.GetDirectories(Dizin);

    int DosyaSayisi = 0;
    int DizinSayisi = 0;

    foreach(string DosyaAdi in Dosyalar)
    {
        FileInfo fi = new FileInfo(DosyaAdi);
        Console.WriteLine("{0,-30}{1,-30}",
                         fi.Name, fi.CreationTime);
    }
}

```

```

        DosyaSayisi++;
    }

    foreach(string DizinAdi in Dizinler)
    {
        DirectoryInfo di = new DirectoryInfo(DizinAdi);
        Console.WriteLine("{0,-30}{1,-30}",
                         "<dir>" + di.Name, di.CreationTime);
        DizinSayisi++;
    }

    Console.WriteLine("\n\n" + Dizin + " dizininde,");
    Console.WriteLine("\n\n{0} dizin, {1} dosya
                      bulundu\n", DizinSayisi, DosyaSayisi);
}

```

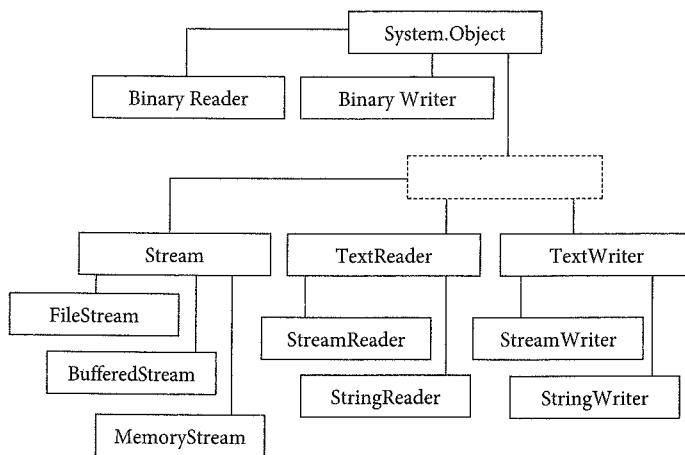
Aşağıda programın çalışma anından bir örnek göremektesiniz.

<dir>Setup	13.07.2002	18:19:56
<dir>ShellExt	13.07.2002	18:19:56
<dir>pool	13.07.2002	18:19:56
<dir>URITemp	24.08.2002	00:42:59
<dir>ubem	13.07.2002	18:19:56
<dir>Windows Media	13.07.2002	18:35:37
<dir>wins	13.07.2002	18:19:56

d:\winnt\system32 dizinde.  
39 dizin, 2020 dosya bulundu  
Dizin Gir >> pwd  
Su an ki dizin: d:\winnt\system32  
Dizin Gir >> dsadasdas  
Yanlis yol...  
Dizin Gir >>

## Dosya Yazma ve Okuma İşlemleri

C#'ta dosyalardan okuma ya da dosyalara yazma yaparken akımlardan faydalanzızz. Akımlar (*streams*) daha önce de denildiği gibi içinde bilgi taşıyan bir soyutlamadır. System.IO isim alanında dosya yazma okuma ile ilgili çok sayıda sınıf vardır. Bu sınıflar Stream sınıflarında olduğu gibi bir akımdan okuma yapan ya da bir akıma yazma yapan sınıflar da olabilir. Ayrıca binary (ikili) düzeyde işlem yapan çeşitli sınıflar da vardır. Bu kısımda bu sınıfları en çok kullanılan özellikleriyle inceleyeceğiz. Akıma yazma ve akımdan okuma ile ilgili sınıfların toplu olarak System.IO isim alanındaki organizasyonunu şema halinde inceleyelim:



Bir dosyadan byte düzeyinde bir veri almak için **FileStream** sınıfı kullanılır. **StreamReader** ve **StreamWriter** sınıfları da metin tabanlı dosyalardan okuma ve yazma yapmak için tasarlanmıştır. **BinaryReader** ve **BinaryWriter** sınıfları ise binary (ikili) dosyalarda işlem yapmak için kullanılır. **StringReader** ve **StringWriter** sınıfları da dosyadan formatlı bir şekilde yazı okumak veya dosyaya formatlı bir şekilde yazı yazmak için kullanılır (C dilindeki sprintf fonksiyonu gibi).

Biz bu konuda bu sınıfların en çok kullanılan özelliklerini inceleyeceğiz. Bu sınıfların tamamını bu kitap dahilinde incelemek mümkün değildir. Bütün bu sınıflarla ilgili ayrıntılı bilgileri MSDN dökümanlarında bulabilirsiniz.

## FileStream Sınıfı

**FileStream** sınıfı ile diskte bulunan bir dosya açılır, **StreamWriter** ve **StreamReader** sınıfları yardımıyla dosya üzerinde işlemler yapılır. Dosyalar üzerinde metin tabanlı işlemler yapabileceğimiz gibi byte düzeyinde işlemler de yapılabilir. Bir dosyadan byte düzeyinde bilgi almak binary (ikili) işlemler kapsamına girmektedir. Zaten işletim sistemi için dosya kavramı byte düzeyindedir. Dosyadaki bilgilere anlam veren bize riz. Örneğin byte düzeyinde alınan verileri karakterlere çevirerek dosyalarda metin tabanlı işlemler yaparız.

Bir **FileStream** nesnesi çok değişik yöntemlerle oluşturulabilir. Aşağıda **FileStream** sınıfının yapıcı metotları ile oluşturulmuş **FileStream** nesnelerini görürsünüz.

```

string yol = @"D:\dosya.txt";
FileStream fs1 = new FileStream(yol, FileMode.OpenOrCreate);
FileStream fs2 = new FileStream
    (yol, FileMode.OpenOrCreate, FileAccess.Write);
  
```

```

FileStream fs3 = new FileStream
    (yol, FileMode.OpenOrCreate, FileAccess.Write,
    FileShare.None);
  
```

```

FileInfo fil = new FileInfo(yol);
FileStream fs4 = fil.OpenRead();
  
```

```

FileInfo fi2 = new FileInfo(yol);
FileStream fs5 = fi2.OpenWrite();
  
```

```

FileInfo fi3 = new FileInfo(yol);
FileStream fs6 = fi3.Create();
FileInfo fi4 = new FileInfo(yol);
FileStream fs7 = fi4.Open(FileMode.OpenOrCreate);
  
```

Gördüğünüz gibi birçok şekilde **FileStream** nesnesi oluşturabiliyoruz. Hatta daha önce gördüğümüz **FileInfo** sınıfının bazı metodlarını kullanarak da dosya akımı oluşturma şansına sahibiz.

Şimdilik işlemlerimizi dosyaya ilişkin akım nesnesini oluşturken dosyanın yolunun doğru girdiğini ve dosyanın gerçekten var olduğunu kabul edeceğiz. Dosyalarla ilgili olası hatalarda çalışma zamanında çeşitli hatalar alırız. Bu hataların ne olduğunu isitsnai durum yakalama (exception handling) konusunda işleyeceğiz.

**FileMode**,  **FileAccess** ve  **FileShare** numaralandırmalarını bir önceki kısımda anlatmıştık. Numaralandırmaların her bir simbolü dosya ile nasıl bir ilişkide olacağımı zı belirliyor. Dosyalarla ilgili işlemlerimiz bittiğinde **FileStream** sınıfının  **Close()** metodu ile **FileStream** nesnesi tarafından tutulan kaynaklar boşaltılır. Böylece ilgili dosya başka prosesler tarafından da işlenebilir hale gelir.  **Close()** metodu aşağıdaki gibi kullanılmalıdır.

```

fs.Close();
  
```

## FileStream 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. Bu metodların prototipleri aşağıdaki gibidir.

```

int ReadByte()
  
```

Bu metod ile akımdan bir byte'lik bilgi okunur ve akımdaki okuma pozisyonunu bir artırır ki bir sonraki okumada tekrar aynı byte değeri okunmasın. Okunan byte değeri int türüne dönüştürülür ve bu değer ile geri dönülür.

İkinci metot ise aşağıdaki gibidir:

```
int Read(byte[] dizi, int baslangic, int adet)
```

Bu metod ile adet kadar byte dizisi akımdan okunur, okunan bu veriler, byte dizisine dizi[baslangic] elemanından itibaren yerleştirilir. Geri dönüş değeri okunan byte miktarını verir. adet ile okunan byte miktarı her zaman aynı olmayabilir. Bu durum özellikle dosya sonuna gelindiğinde görülür. Eğer dosya sonuna gelindiye okuma yapılamayacağından 0 değerine geri dönülür.

Şimdi komut satırı argümanı olarak alınan dosyanın içeriğini konsol ekranında gösteren bir program yazalım. Program yazarken bir döngü kurulacak ve döngü koşulunda dosya akımdan bir byte okunacak. ReadByte() metodunun sonucu -1 olana kadar bu işlem devam edecek. Her okunan byte değeri char türüne dönüştürülerek ekrana yazılacak. Programın kaynak kodu aşağıdaki gibidir.

```
using System;
using System.IO;
class DosyaAkimi
{
    static void Main(string[] args)
    {
        string yol = args[0];

        FileStream fs = new FileStream(yol, FileMode.Open);

        int OkunanByte;

        while((OkunanByte = fs.ReadByte()) != -1)
        {
            Console.Write((char)OkunanByte);
        }
    }
}
```

Kaynak dosyayı FileReader.cs adıyla kaydedip derledikten sonra komut satırına aşağıdaki komutu yapıp enter'a basın.

```
FileReader FileReader.cs
```

Bu komut ile FileReader.cs dosyasının bütün içeriğini aşağıdaki gibi konsola yazdırılmış oluyoruz.

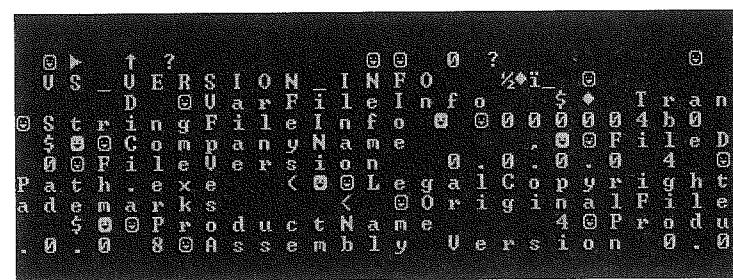
```
D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum10\kod>FileReader FileReader.cs
using System;
using System.IO;

class DosyaAkimi
{
    static void Main(string[] args)
    {
        string yol = args[0];
        FileStream fs = new FileStream(yol, FileMode.Open);
        int OkunanByte;
        while((OkunanByte = fs.ReadByte()) != -1)
        {
            Console.Write((char)OkunanByte);
        }
    }
}
H:\cskitap\kitap\Bolum10\kod>
```

Komut satırından dosyanın yanlış girilmesi ya da dosyanın bulunamaması gibi durumlarda çalışma zamanında hata alınır. Bu tür durumları önlemeyi henüz görmediğimiz için dosyanın yolunu doğru girmeniz gereklidir.



Bu program ile metin tabanlı olmayan bir dosya açmaya çalıştığımızda ekrana anlamsız karakterler yazacaktır. Aşağıdaki görüntü exe uzantılı bir dosyanın açılması sonucu elde edilmiş görüntündür.



Şimdi de bir dosyaya byte düzeyinde veri yazmak için kullanılan Write() ve WriteByte() metodlarını inceleyelim.

Dosya akımına bir byte'lık bilgi yazmak için,

```
void WriteByte(byte veri)
```

metodu kullanılır. Eğer yazma işlemi başarısız olursa istisnai durumlar meydana gelir.

Dosya akımına bir byte dizisi yazdırmak içinse,

```
void Write (byte[] dizi, int baslangic, int adet)
```

metodu kullanılır. Bu metot ile dizi'den, `dizi[baslangic]` elemanından itibaren adet sayısı kadar byte dosya akumuna yazılır. Yazılma işleminden sonra dosya akımının konum göstericisi yazılan byte sayısı kadar ötelenir.

Dosya akımına yazılan veriler dosya sistemindeki dosyaya hemen aktarılmaz. Dosya akımı tamponlama mekanizması ile çalıştığı için belirli bir miktar kadar veri yazılına kadar dosya güncellenmez. Ancak `FileStream()` sınıfının `Flush()` metodunu kullanarak istediğimiz anda tamponu boşaltıp dosyayı tampondaki bilgilerle güncelleyebiliriz.

Şimdi dosya akımına yazma işlemi ile ilgili bir örnek verelim. Hatırlarsanız metodlar bölümünde bir komut satırından girilen karakter dizisini şifreleyen bir program yazmıştık. Şimdi bu şifreleme işini dosya bazında yapacağız. Komut satırından girilen dosyanın girilen bir şifreye göre her byte'ı XOR işlemine tabi tutulacak. XOR işleminin geri dönüşümlü olmasından dolayı aynı şifre ile, şifrelenmiş dosyayı tekrar açabileceğiz.

```
using System;
using System.IO;

class DosyaSifreleme
{
    static int Main(string[] args)
    {
        if(args.Length !=2)
        {
            Console.WriteLine("Hatalı argüman... ");
            return 0;
        }

        string kaynak = args[ 0] ;
        string hedef = args[ 1] ;
        string sifre;

        Console.Write("Sifre :");
        sifre = Console.ReadLine();

        int XOR=0;

        for(int i =0; i<sifre.Length; ++i)
        {
            XOR = XOR + (int)(sifre[ i]*10);
        }

        FileStream fsKaynak = new FileStream
            (kaynak, FileMode.Open);
```

```
FileStream fsHedef = new FileStream
    (hedef, FileMode.CreateNew, FileAccess.Write);

int OkunanByte;
byte SifreliByte;

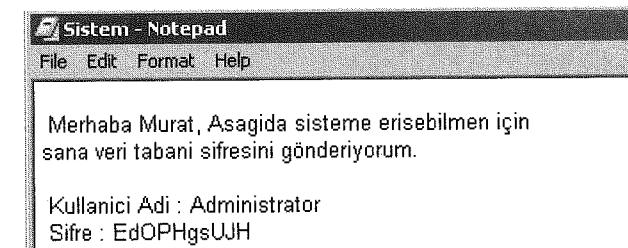
while( (OkunanByte = fsKaynak.ReadByte()) != -1)
{
    SifreliByte = (byte)((int)OkunanByte ^ XOR);
    fsHedef.WriteByte(SifreliByte);
}

fsKaynak.Close();
fsHedef.Close();

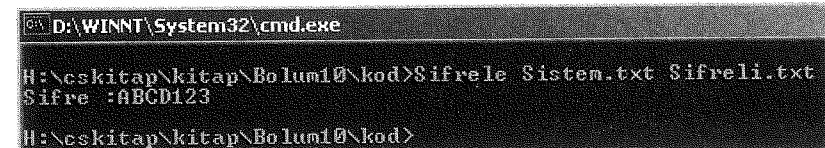
return 0;
}
```

Programı test etmek için programı önce derleyin (Ben, adını `Sifrele.exe` yaptım). Daha sonra programın bulunduğu klasörde aşağıdaki gibi yeni bir text dosyası oluşturun ve adını da `Sistem.txt` verin.

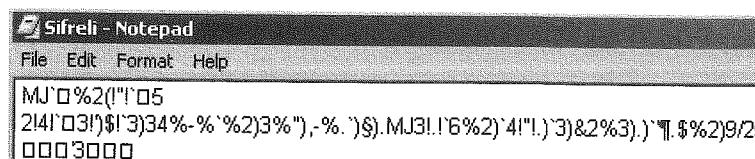
`Sistem.txt`:



Şimdi komut satırından aşağıdaki gibi programı çalıştırın ve dosyanızın şifrelenmesi için bir anahtar kelime girin. Bu gireceğiniz şifreyi dosyayı tekrar çözmek için kullanacaksınız. Bu yüzden şifreyi unutmayın.



Evet dosyamız artık şifreli durumda. Şifrelenmiş dosyayı yani `Sifreli.txt` dosyasını text editöründe açtığımızda aşağıdaki gibi anlamsız bilgilerle dolduğunu görürüz.



Sifrele programınız elinizde olduğu sürece orijinal Sistem.txt dosyasını bilgisayarınızdan silebilirsiniz. Orijinal dosyayı elde etmek için yukarıdaki işlemlerin aynısını tekrar yapın. Yani komut satırından programı aşağıdaki gibi çalıştırın.

Sifrele Sifreli.txt Orijinal.txt

Tabi şifreyi "ABCD123" olarak girmeniz gereklidir. Aksi halde oluşturacak dosya da yine anlamsız karakterler görürsünüz. Bu şifreleme programını daha da geliştirip işinizi eğlenceli hale getirebilirsiniz.



Bu programda dosyaların doğru girilmesi gereklidir. Aksi halde çalışma zamanında istisnai durumlar oluşur.

Gördüğünüz gibi bir dosyayı byte'lar seviyesinde inceleme ile ilgili birçok iş yapabiliyoruz. Örneğin, komut satırından belirtlen bir dosyayı byte düzeyinde başka bir dosyaya kopyalamak öğrendiğimiz bilgilerle rahatlıkla yapılabilir. FileStream dosya akımı, dosyaların türünden bağımsız olarak çalıştığı için bir resim dosyası üzerinde de işlemler yapabiliyoruz.

FileStream sınıfının bu metod ve özelliklerinin yanı sıra daha birçok özelliği vardır. Bu özellikler ve metodlardan en önemlileri aşağıda özetlenmiştir.

#### Özellikler:

**bool CanRead:** Bu özellikle akımdan okunma yapılmış yapılmayacağı öğrenilir.

**bool CanSeek:** Bu özellikle akımda konumlandırma yapılmış yapılmayacağı öğrenilir.

**bool CanWrite:** Bu özellikle akıma yazma yapılmış yapılmayacağı öğrenilir.

**long Position:** Bu özellikle akımda o anda bulunan konum bilgisi öğrenilir.

**long Length:** Bu özellikle akımın byte olarak büyüklüğü öğrenilir.

#### Metotlar:

**void Lock(long pozisyon, long uzunluk)**

Bu metod ile akımın pozisyonдан itibaren uzunluk kadar alanı başka proseslerin erişime engellenir.

**long Seek(long offset, SeekOrigin a)**

Akımin konumu SeekOrigin ile belirtilmiş konumdan offset byte kadar ötelenir. Seek() metodunu kullanarak dosya akım konumlandırıcısını istediğimiz şekilde değiştirebiliriz. Bu sayede dosya akımının istenilen bölgesinden veri okuma şansına sahip oluyoruz. SeekOrigin numaralandırmasının içерdiği semboller aşağıdaki gibidir.

**Begin:** Akımın başlangıç noktası

**Current:** Akımın o anda bulunduğu nokta

**End:** Akımın en son noktası

## Dosya Akımı ile Text İşlemleri Yapmak

Bir dosya akımının metin tabanlı veriler şeklinde de okumamız mümkündür. Bu tür durumlar text dosyalarından okuma ve yazma yaparken karşımıza çıkar. Hatta metin dosyasına formatlı bilgi yazma gibi birtakım özellikler de metin tabanlı işlemler sayesinde olmaktadır. Karakter tabanlı dosyaları okumak yüksek seviyeli bir işlemidir. Yani alta nelerin olduğu pek bilinmeden her şey yazılır. Metin tabanlı dosyalarla işlem yapmak için StreamReader ve StreamWriter sınıfları mevcuttur. Bu sınıflar ile bir dosya akımı okunduğunda akım otomatik olarak metin moduna çevrilir. Buradaki ana tema; varolan bir dosya akımı (file stream) StreamWriter sınıfı yardımı ile okunur ya da akıma yeni şeyler eklenir. StreamReader ve StreamWriter sınıfları ile çalışırken karakter kodlamasının ne şekilde yapıldığına çok dikkat etmeyeceğiz. Çünkü bu sınıflar sistemin karakter kodlamasına göre kendilerini ayarlarlar.

## StreamReader Sınıfı

Bir StreamReader nesnesini birçok şekilde oluşturabiliriz. Ancak en sık kullanılan biçimleri aşağıda verilmiştir.

```
string dosya = @"D:\dosya.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);
```

Bunların dışında daha birçok StreamReader nesnesi oluşturma yöntemi vardır. Bu yöntemlerin hepsini MSDN dökümanlarından öğrenebilirsiniz.

Diğer sınıflarda olduğu gibi StreamReader nesneleri ile işimiz bittiğinde Close() metodunu kullanarak kaynakların iade edilmesini sağlamamız gereklidir.

StreamReader sınıfının en önemli metodları aşağıda verilmiştir.

#### 1. string ReadLine();

Akımdan bir satırlık veriyi okur ve string türü olarak geri döner. Eğer veri okunamazsa null değer ile geri dönülür. Okunan satır sonuna '\n' karakteri eklenmeyecektir.

#### 2. string ReadToEnd();

Akımdaki verilerin tamamını string olarak geri döndürür. (Okuma işlemi okumanın yapıldığı andaki konumdan itibaren olacaktır.) Okuma yapılamazsa null değer yerine boş string ile geri dönülür.

#### 3. int Read();

Akımdan bir karakterlik bilgi okunur ve bu karakterin int'e dönüşmüş hali ile geri dönülür. İşlem başarısız olursa -1 değerine geri döner.

#### 4. int Read(char[ ] dizi, int indeks, int adet);

Akımdan adet kadar karakteri, dizi[indeks] elemanından itibaren dizeye yerleştirir.

#### 5. int Peek();

Akımdan bir karakterlik bilgi okur ve bu karakterin int'e dönüşmüş hali ile geri döner. İşlem başarısız olursa -1 değerine geri döner. En önemli nokta ise akımın konum göstericisinin değiştirilmemesidir. Yani Peek() metodundan sonra Read() metodunu kullanırsak aynı karakteri okuruz.

Şimdi bir text dosyasının nasıl satır satır okunabileceğine örnek verelim.

Aşağıdaki bilgileri içeren bir text dosyası oluşturun ve programı yazın.

Dosya.txt

-----

Sefer Algan

Mehmet Demir

Gökcen Yıldırım

Aziz Durmaz

Cemal Öztürk

Ahmet Hançer

using System;

using System.IO;

class AkımOkuyucusu

```
{
    static void Main()
    {
        string dosya = "Dosya.txt";
        FileStream fs = new FileStream(dosya, FileMode.Open);

        StreamReader sr = new StreamReader(fs);

        string Line;

        while((Line = sr.ReadLine()) != null)
            Console.WriteLine(Line);
        fs.Close();
    }
}
```

Programı çalıştırığınızda konsol ekranında Dosya.txt dosyasına benzer bir görüntü elde etmelisiniz.

## StreamWriter Sınıfı

StreamWriter sınıfı bir akıma karakter tabanlı bilgileri yazmak için kullanılır. StreamReader sınıfı ile tamamen ters bir işlem yapar. StreamWriter nesneleri aşağıdaki gibi oluşturulabilir.

```
string dosya = @"D:\dosya.txt";

FileStream fs = new FileStream(dosya, FileMode.Open);
StreamWriter srl = new StreamWriter(fs);

StreamWriter sr2 = new StreamWriter(dosya);

FileInfo fi = new FileInfo(dosya);
StreamWriter sr3 = fi.CreateText();
```

StreamReader sınıfında olduğu gibi Close() metodu ile StreamWriter nesnelerine ilişkin kaynaklar iade edilir.

Bunların dışında daha birçok StreamWriter nesnesi oluşturma yöntemi vardır. Bu yöntemlerin hepsini MSDN dökümanlarından öğrenebilirisiniz.

StreamReader sınıfının en önemli metodları aşağıda verilmiştir.

#### 1. void Write(string str);

Write() metodunun birçok aşırı yüklenmiş versiyonu vardır. Bu şekli ile akıma str yazısı eklenir. Yazının sonuna herhangi bir sonlandırıcı karakter konulmaz. Bu metod ile diğer bütün temel veri türlerini akıma yazdırabiliriz. Örneğin Write(5), Write(false)

veya Write('c') gibi. Aynı şekilde bir karakter dizisinin tamamını da akıma Write() metodu ile aşağıdaki gibi yazdırabiliriz.

```
2. void WriteLine(string str);
```

WriteLine() metodu Write() metodu ile hemen hemen aynı işi yapar. Tek fark WriteLine() metodu akıma yazarken eklediği yazının sonuna '\n' satır atlama karakterini de ekler. Write() metodundan farklı olarak WriteLine() metodunu parametresiz de çağırabiliriz. Bu durumda akıma sadece bir '\n' karakteri yazılır.

```
3. void Flush();
```

Tampondaki bilgilerin boşaltılmasını ve dosya sistemindeki dosyanın güncellenmesini sağlar.

Ayrıca StreamWriter sınıfının NewLine özelliği ile satır ayıracı olan karakterleri belirleyebiliriz. Varsayılan olarak bu karakterler '\n' ve '\r'dir.

StreamWriter sınıfının kullanımına bir örnek verelim. Aşağıdaki programda kullanımının girdiği yazılar bir dosyaya kaydediliyor. Kullanıcı 'q' ya da 'Q' girerse program sonlanıyor.

```
using System;
using System.IO;

class AkimYazici
{
    static void Main()
    {
        string dosya = "Dosya.txt";
        FileStream fs = new FileStream
            (dosya, FileMode.Append, FileAccess.Write);

        StreamWriter sw = new StreamWriter(fs);

        while(true)
        {
            string Line = Console.ReadLine();

            if(Line.ToLower() == "q")
                break;

            sw.WriteLine(Line);
        }
        sw.Flush();
        sw.Close();
    }
}
```

## BinaryWriter ve BinaryReader Sınıfları

Bu sınıflar bir akıma istenilen türden verileri yazmak için kullanılır. Örneğin şu ana kadar byte ve karakter düzeyinde işlemler yaptık, ancak bu sınıflar ile dosya akımına istediğimiz temel veri türünden bilgileri yazabiliris. Her iki sınıf türünden nesne oluşturma, StreamReader ve StreamWriter sınıfları ile aynı olduğu için tekrar anlatılmayacaktır. Bu sınıflarda sırasıyla Write(int), Write(char), Write(char[]), Write(byte) ve ReadByte(), ReadChar(), ReadUInt32(), ReadString() gibi metodlar bulunmaktadır. Bu metodlar bütün veri türleri için bildirilmiştir.

Aşağıdaki programda bir dosyaya bu sınıflar aracılığı ile çeşitli veriler ekleniyor ve daha sonra bu veriler alınarak ekrana yazdırılıyor.

```
using System;
using System.IO;

class IkiliDosyalar
{
    static void Main()
    {
        int i = 5;
        decimal d = 15.54M;
        char c = 'A';

        string dosya = "Dosya.txt";

        FileStream fs1 = new FileStream
            (dosya, FileMode.OpenOrCreate);
        BinaryWriter bw = new BinaryWriter(fs1);

        bw.Write(i);
        bw.Write(d);
        bw.Write(c);

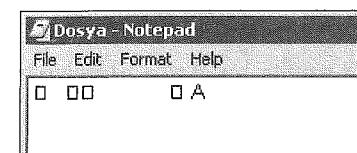
        bw.Close();

        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();
    }
}
```

Programı çalıştırığınızda ekrana yazılan bilgiler anlamlı iken bilgilerin yazıldığı Dosya.txt dosyasının içeriği yandaki gibi anlamsız bilgilerle doldurulmuştur.



Bu tür dosyalara binary (ikili) dosya denilmektedir. Yani bilgiler text formatında saklanmamıştır.

## Console I/O İşlemleri

I/O işlemleri için gerekli olan sınıflardan System.IO isim alanı içinde olmayan tek sınıf Console sınıfıdır. Console sınıfını gerek ekrana birşeyler yazdırmak için gerekse de kullanıcından bilgi almak için şimdije kadar sıklıkla kullandık. Şimdi Console sınıfını daha yakından inceleyip alta nelerin döndüğüne bakalım.

Konsol I/O işlemleri için önceden tanımlanmış 3 tane standart akım mevcuttur. Bu akımlar TextWriter türü olan Console.Out, Console.Error ve TextReader türünden olan Console.In'dır. Konsol ekranına yazdığımız yazılar aslında TextWriter sınıfının metodları ile olmaktadır. Console.WriteLine ise bizim için sadece bir aracılık görevi yapar. Örneğin aşağıdaki her iki deyim de eşdeğerdir. Yani her ikisi de ekrana "Merhaba!" yazısını yazdıracaktır.

```
Console.Out.WriteLine("Merhaba!");  
Console.WriteLine("Merhaba!");
```

Kısaltası Console.WriteLine() metodunun altında yatan sınıf TextWriter sınıfıdır.

Konsol ekranına bir yazı yazdırmak için Console sınıfının WriteLine() ve Write() metodunu kullanıyoruz. İki metot arasındaki tek fark WriteLine() metodunun yazılacak yazının sonuna bir satır sonu karakteri ('\n') eklemesidir. Bu iki metodun aşırı yüklenmiş birçok metodu vardır. Bu metotları kitabıń çeşitli bölümlerinde inceledik. String işlemleri kısmında daha da ayrıntılı bir şekilde inceleyeceğiz.

Konsoldan bilgi almak için Console sınıfının Read() ve ReadLine() metodlarını kullanıyoruz. Read() metodu okunacak akımdan bir karakter okur ve bu karakterin int değerine geri döner. Eğer akımdan veri okunamazsa -1 değerine geri döner. Read() metodu biz enter tuşuna basana kadar bekler. Enter tuşuna bastıktan sonra girdiğimiz karakterlerin ilk ile geri döner. Ancak bilgileri girdiğimiz akımda ilk karakter dışında hala karakterler mevcut. Read() metodu tampon akımdan okuma yaptığından dolayı diğer karakterleri henüz okumadığı için tampondan silmemiştir. İkinci bir defa Read() metodunu çağrıdığımızda ilk girdiğimiz yazındaki ikinci karakteri okur. Bu böyle devam eder gider. En önemli nokta ise biz enter tuşuna bastığımızda '\n' ve 'r' karakterleri de tampona eklenir. Bu karakterler açıkça okumadan tampondan silmezler. Eğer tamponda herhangi bir bilgi kalmamışsa Read() veri girilmesi için bekler. Eğer tamponda okuma yapabileceği veriler varsa beklemeden tampondan okuma yapar.

Read() metodunun bu zorlu işlemlerinden kurtulmak için ReadLine() metodunu kullanmalıyız. ReadLine() metodu biz enter tuşuna basana kadar yazdığımız karakterleri okur ve string türü olarak geri döndürür. En güzel özelliği ise tampondaki enter tuşundan dolayı gelen '\n' karakterini tampondan silmesidir. Bu yüzden ReadLine() metodunu ard arda dilediğimiz gibi kullanabiliriz.

Console.In özelliğini kullanarak da Read() ve ReadLine() metodlarını çağırabiliriz. Ayrıca Console.In ile aşağıdaki metodları da çağırabiliriz. Bu metodlar TextReader sınıfının üye metodlarıdır.

**int Peek()** : Bu metod ile standart girdi akımından bir karakter okunur ancak bu karakter tampondan silinmez.

**int ReadBlock(char[] dizi, int index, int adet)** : Bu metod ile standart girdi akımından adet kadar karakter diziye index elemanından itibaren yerleştirilir.

**string ReadToEnd()** : Bu metod ile standart girdi akımındaki bütün veriler okunarak tampondan temizlenir. Okunan veriler bir string nesnesi olarak geri döndürülür.

## Standart Akımların Yönlendirilmesi

C# ta bütün I/O işlemleri akımlar (stream) üzerine kuruludur. Standart akımları başka akımlara yönlendirmek mümkündür. Mesela Console.In standart girdi akımını bir NetworkStream (ağ akımı) akımına yönlendirirsek Console.Read() ve Console.ReadLine() ile bu ağ akımından bilgi okunur. Ağ akımları bu kitabın konuları dahilinde olmadığı için biz yönlendirme işlemini standart akımlar ile dosya akımları arasında gerçekleştireceğiz.

İşletim sistemi düzeyinde de standart akımları yönlendirebilmemiz mümkündür. Komut satırından < ve > imleçlerini kullanırsak ilgili akımları dosya akımlarına yönlendirmiş oluruz. Örneğin

```
using System;  
using System.IO;  
  
class KonsolIO  
{  
    static void Main()  
    {  
        Console.WriteLine("Yönlendirildim..");  
        Console.WriteLine("Standart çıktı akımı dosya  
akımına yönlendirildi.");  
    }  
}
```

programını komut satırından aşağıdaki gibi çalıştırırsanız,

```
ProgramAdı > deneme.txt
```

Console.WriteLine() metodu ile yazdırılmış bütün yazılar deneme isimli dosyaya yazdırılacaktır. Bu da standart çıktı olan konsol ekranının dosya akımına yönlendirilmiş olmasına mümkün olur.

Programı komut satırından,

```
ProgramAdı < deneme.txt
```

şeklinde çalıştırırsak bu sefer de standart girdi akımını deneme.txt dosyasına yönlendirmiş oluruz. Yani Console.Read() ya da Console.ReadLine() metodlarını kullandığımızda konsoldan bilgi alma yerine okuma işlemini deneme.txt dosyasından yaparlar.

C# programlarımız içinde akımları yönlendirmek için Console sınıfının aşağıdaki metotları kullanılır.

```
static void SetOut(TextWriter tw)
static void SetError(TextWriter tw)
static void SetIn(TextReader tr)
```

Şimdi Console.In akımını SetIn metodu ile bir dosya akımına yönlendirelim. Aşağıdaki programı yazın,

```
using System;
using System.IO;

class KonsolIO
{
    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 ve programı oluşturduğunuz klasörde “deneme.txt” isimli bir dosya açın ve dosyaya iki satır yazı yazın.

Programı çalıştırığınızda, dosyadaki iki satırın ekrana yazıldığını göreceksiniz. Dosyanın okuma işini ReadLine() yapmıştır. Aynı şekilde Console.WriteLine() metodu

ile de dosyaya yazma işlemi yapabiliriz. Bunun için elbette ki standart çıktı olan Console.Out akımını bir dosya akımına yönlendirmemiz gereklidir.

Console.Error standart akımını Console.SetError() metodu ile bir dosya akımına yönlendirerek program içinde oluşan hataları bir dosya içinde tutabiliyoruz. Console.Error akımına yazmak için yine WriteLine() ya da Write() metodu kullanılır.

## Temel String (Karakter Dizisi) İşlemleri

Programlarınızı yazarken en sık kullandığımız veriler yazılardır. Yazıları bir karakter dizisi olarak da adlandırabiliriz. Yaygın bir kullanım olduğu ve C# dilinde temel veri türlerinden biri olduğu için karakter dizilerine bu kısımda string diyeceğiz. C#'ta string işlemlerinin tamamı System.String sınıfındaki üye özellik ve metodlarla yapılmıştır. Bu bölümde string veri türünü ve en çok kullanılan metodlarını inceleyeceğiz. Ayrıca string formatlama gibi önemli bir konuya da değineceğiz.

Bir String nesnesi bir kere tanımlandıktan sonra, stringde bulunan herhangi bir karakter bir daha değiştirilemez. Ancak string üzerinde çeşitli metodlar ile işlemler yapıp metodların geri dönüş değerleri yeni string nesnelerine aktarılabilir. O halde stringde bulunan her karakter readonly özelliğine sahiptir. Bunu test etmek için aşağıdaki deyimleri içeren bir program yazın ve derleyin.

```
string a = "ali";
a[0] = 'b' ;
```

Stringlerin içindeki karakterlere ulaşabilmek için bir indeksleyici tanımlanmıştır. Bu yüzden stringlerle bir karakter dizisi gibi işlem yapmamız mümkün değildir. Yukarıdaki deyimlerin hata vermesinin sebebi String sınıfında tanımlanan indeksleyicinin sadece get bloğuna sahip olmasıdır. Yani sadece okunabilir bir indeksleyici olmasıdır.

### String Tanımlama

String nesnesi oluşturmanın birçok yolu vardır. Bu yollardan en sık kullanılanları aşağıda listelenmiştir.

Aşağıdaki tanımlama biçimi şu ana kadar gördüğümüz tanımlama biçimidir. Bu şekildeki tanımlama ile String sınıfının yapıcı metodunu gizlice çağrılmaktadır.

```
string a = "CsharpNedir";
```

String oluşturmanın diğer bir yolu karakter dizilerini kullanmaktadır. Karakter dizileri ile string nesneleri aşağıdaki gibi oluşturulabilir.

```
char[] dizi = { '1', '2', '3', '4' };
String str = new String(dizi);
```

Yukarıda tanımlanan str nesnesinde "1234" yazısı bulunmaktadır. Karakter dizilerinden string nesnelerini bir de aşağıdaki şekilde oluşturabiliriz.

```
char[] dizi = { '1', '2', '3', '4' };
String str = new String(dizi, 1, 2);
```

Console.WriteLine(str) dersek ekrana "23" yazdığını görürüz. Bu metod ile karakter dizisinin birinci elemanından itibaren 2 eleman string nesnesini oluşturuyor.

n tane x karakteri içeren bir String nesnesi aşağıdaki gibi oluşturulur.

```
String str = new String('x', n);
```

Bu yapıçı metodların dışında parametre olarak gösterici (pointer) alan birçok yapıçı metod mevcuttur. Gösterici kullanımı güvensiz kod yazma kapsamına girdiği için bu bölümde bu metodlara değinilmeyecektir.

## String Metotları

String sınıfının onlarca metodu tanımlanmıştır. Bu metodların hepsini ezbere bilmek gerçekten zor bir iştir. Bu yüzden bu bölümde kullanma ihtiyalimizin yüksek olduğu metodları inceleyeceğiz. Zaten metodların temel olarak ne iş yaptığıni anlaysak aşırı yüklenmiş diğer metodları MSDN dökümanlarından rahatlıkla öğrenebiliriz.

### String.Concat()

İlk olarak String sınıfının statik Concat() metodunu inceleyelim. Concat() metodу bir stringin sonuna başka stringler eklemek için kullanılır. Stringlerin sonuna ekleme işlemi + operatörü ile de yapılabildiği için Concat() metodу çok sık kullanılmaz, yine de Concat() metodunun değişken sayıda parametre alan versiyonu istediğimiz kadar stringi arka arkaya eklediği için tercih edilebilir. Aşağıdaki programda Concat() metodunun ve + operatörünün kullanımını ile ilgili bazı örnekler görmektesiniz.

```
using System;

class StringConcat
{
    static void Main()
    {
        //String Concat(String str1, String str2)
        String str1 = String.Concat("Sefer ", "Algan");
        Console.WriteLine(str1);

        //String Concat(params string[] stringler)
        String str2 = String.Concat("a", "b", "b", "1", "4", "K");
        Console.WriteLine(str2);
    }
}
```

```
//operator+
String str3 = "Sefer " + "Algan";
Console.WriteLine(str3);
//String Concat(String str1, String str2, String str3)
String str4 = String.Concat("C#", "nedir", "?com");
Console.WriteLine(str4);

//String Concat(object o2, object o1)
String str5 = String.Concat(5, 'c');
Console.WriteLine(str5);
}
```

Yukarıdaki bütün kullanımarda stringler ard arda eklenmiştir. Concat() metodunun daha birçok kullanım şekli vardır. Fakat bu kadarı kapsamlı uygulamalar geliştirmek için yeterlidir. Siz de takdir edersiniz ki bir string ile ilgili belki yüzlerce metod yazılabılır, fakat bu metodların çoğu birbirleriyle bağlantılı çalışır. Yeri gelecek kendi metotlarımızı kendimiz geliştireceğiz.

### String.Compare()

İki stringin karakter dizisini karşılaştırmak için Compare() metodу kullanılır. Karşılaştırma işlemleri == ve != operatörleri ile de yapılabilir. Ancak Compare() metodunun birçok özelliği vardır. Bu özelliklerden faydalananarak program geliştirme hızımızı artırabiliriz.

Compare() metodу genellikle iki karakter dizisini büyükük açısından karşılaştırır. Aşağıda Compare() metodunun değişik kullanım biçimlerini inceleyebilirsiniz. Bu metodların tamamının geri dönüş değeri int türüdür. Ayrıca hepsi statik metottur.

1. int Compare(String str1, String str2)
- str1, str2'den büyükse sıfırdan büyük bir değere geri dönülür. str2 büyükse sıfırdan küçük bir değere geri dönülür. Eğer iki string eşitse sıfır değerine geri dönülür.
2. int Compare(String str1, String str2, bool BuyukKucuk)
- Birinci metot ile aynı işi yapar, tek farkı büyük ya da küçük harf duyarlığını belirleyemeyizdir. Eğer üçüncü parametre true ise büyük/küçük harf duyarlığı dikkate alınmaz. Yani 'a' ile 'A' aynıdır. Eğer parametre false ise birinci metot ile eşdeğerde olur.
3. int Compare(String str1, int index1, String str2, int index2, int adet)
- Birinci metot ile aynı mantıkla çalışır. Tek farkı str1[index1] den itibaren adet kadar karakteri str2[index2] den itibaren adet kadar karakter ile karşılaştırmasıdır.
4. int Compare(String str1, int index1, String str2, int index2, int adet, bool BuyukKucuk)

Üçüncü metot ile aynı şekilde sonuç üretir. Aradaki fark, ikinci metotta da olduğu gibi büyük ve küçük harf duyarlılığının devreye girmesidir.

String sınıfının bu statik metodlarının yanı sıra CompareTo() isimli statik olmayan bir metodu da vardır. Bu metot ile string karşılaştırma işlemi nesne üzerinden gerçekleştirilir. Bu metot aşağıdaki iki şekilde çağrılabılır.

```
int CompareTo(String str);
int CompareTo(object o);
```

Bu metot ile değişen tek şey parametre sayısıdır. Yukarıdaki statik metodlardaki birinci parametre yerine, bu metodu çağırılan string nesnesi alınır.

## Arama İşlemleri

String sınıfının metodları ile bir string içinde bir karakteri arayabileceğimiz gibi bir alt karakter dizisi de arayabiliriz. IndexOf() metodu aranacak yazının ya da karakterin ilk bulunduğu yerin indeksi ile geri döner. Arama işlemi başarısızsa -1 değerine geri dönlür. IndexOf() metodunun kullanımına bir örnek verelim:

```
using System;
class StringArama
{
    static void Main()
    {
        string yazi = "Pek yakında sinemalarda";

        Console.WriteLine(yazi.IndexOf("akın"));
        Console.WriteLine(yazi.IndexOf('k'));
    }
}
```

Bu programı derlediğimizde ekrana 5 ve 2 yazacaktır.

Diğer bir arama metodu ise aranın karakterin ya da yazının, aranacak yazı içinde en son nerede bulunduğu söyler. Bu metot LastIndexOf() metodudur. Diğer metot gibi bu metot da belirtilen ifadeyi bulamazsa -1 değerine geri döner. Yukarıdaki programın ayınısını LastIndexOf() metodu için yazarsak ekrana bu sefer 5 ve 6 yazacaktır. Programın yeni hali aşağıdaki gibidir.

```
using System;

class StringArama
{
    static void Main()
    {
        string yazi = "Pek yakında sinemalarda";
```

```
        Console.WriteLine(yazi.LastIndexOf("akın"));
        Console.WriteLine(yazi.LastIndexOf('k'));
    }
}
```

Düzen iki önemli metot da IndexOfAny() ve LastIndexOfAny() metodlarıdır. Bu metodlar bir karakter dizisinin herhangi bir elemanın ilk bulunduğu yerin indeksi ile geri döner. Her iki metot da aranan karakter dizisinin hiçbir elemanını bulamazsa -1 değerine geri döner. Aşağıda bu iki metodun kullanımına bir örnek göremektesiniz.

```
using System;

class StringArama
{
    static void Main()
    {
        string yazi = "Pek yakında sinemalarda";

        char[] dizi = { 'a', 'k' };

        Console.WriteLine(yazi.IndexOfAny(dizi));
        Console.WriteLine(yazi.LastIndexOfAny(dizi));
    }
}
```

Programın çıktısı 2 ve 22'dir. Çünkü ilk 'a' ya da 'k' karakteri stringin 2. karakterinde bulunuyor. Bu karakterlerden herhangi biri son olarak 22. karakterde bulunuyor.

Bazen bir stringin sonunun ve başının belli bir karakter dizisi ile sonlanıp sonlanmadığını merak ederiz. Bunu öğrenmek için StartsWith() ve EndsWith() metodlarını kullanırız. String, eğer StartsWith() metodunun parametre ile verilen yazı ile başlıyorrsa true değeri döndürülür. Aynı durum EndsWith() metodu için de geçerlidir. Tabi bu sefer stringin sonuna bakılır.

Bu iki metodu bir örnekle inceleyelim:

```
using System;

class StringArama
{
    static void Main()
    {
        string str1 = "AY";
        string str2 = "A";
        string yazi = "AYAAYAYAYA";
        if(yazi.StartsWith(str1))
            Console.WriteLine("Yazı AY ile başlıyor.");
    }
}
```

```

if(yazi.EndsWith(str2))
    Console.WriteLine("Yazı A ile sonlanıyor.");
}

```

Programı çalıştırıldığınızda ekrana

```

Yazı AY ile başlıyor.
Yazı A ile sonlanıyor.

yazacaktır.

```

Bütün bu metodların diğer aşırı yüklenmiş versiyonlarını incelemek için MSDN dökümanlarına başvurun.

## Budama ve Doldurma İşlemleri

Bazen bir yazının sonundaki veya başındaki boşluk karakterlerini silmek isteyebiliriz. Bu durum özellikle üyelik sistemlerinde kullanıcının, kullanıcı adını ya da şifresini girerken yanlışlıkla boşluk karakteri girdiği zamanlarda görülür. String sınıfının Trim() metodu bir yazının başındaki ve sonundaki karakterleri atar. Trim() metodu aşağıdaki şekillerde kullanılabilir.

```

string Trim()
string Trim(params char[] dizi)

```

İkinci metot ile yazının başından ve sonundan dizi içinde bulunan karakterler silinir.

Bazen de bir yazının sağına ve soluna yeni karakterler eklemek isteyebiliriz. Bu yöntem genellikle yazıları formatlamada kullanılır. Örneğin, bir yazının sağına karakterler eklemek için:

```

string PadRight(int toplam)
string PadRight(int uzunluk, char ch)

```

metotları kullanılır. Birinci metot ile yazının uzunluğu "toplam" olana kadar yazının sağına boşluklar eklenir. İkinci metotta ise "ch" ile belirtilen karakter eklenir. Aynı işlemleri yazının sol tarafına yapmak için

```

string PadLeft(int toplam)
string PadLeft(int uzunluk, char ch)

```

metotları kullanılır.

Aşağıdaki programda bu metodların etkisini görebilirsiniz.

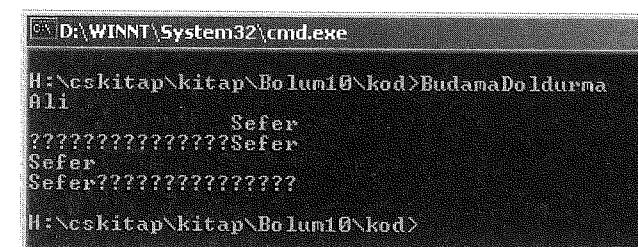
```
using System;
```

```

class BudamaDoldurma
{
    static void Main()
    {
        string str = " Ali ";
        Console.WriteLine(str.Trim());
        string str1 = "Sefer";
        Console.WriteLine(str1.PadLeft(20));
        Console.WriteLine(str1.PadLeft(20,'?'));
        Console.WriteLine(str1.PadRight(20));
        Console.WriteLine(str1.PadRight(20,'?'));
    }
}

```

Programı çalıştırıldığınızda aşağıdaki çıktıyı elde etmelisiniz.



## Split() ve Join() Metotları

Bir yazı içerisinde düzenli bir şekilde devam eden alt karakter dizilerini parçalara ayırarak string türden bir diziye yerleştirmek için Split() metodu kullanılır. Split() metodu genellikle text veritabanı dosyalarında verileri düzenli bir şekilde çekmek için kullanılır. Örneğin

```
str = "Ali, Mehmet, Sefer, Volkan, Ebru"
```

gibi bir yazın virgül ile ayrılmış her bir alt yazısı elde etmek için Split() metodunu kullanabiliriz. Split() metodunun iki farklı kullanım şekli aşağıda verilmiştir.

```

string[] Split(params char[] ayirici)
string[] Split(params char[] ayirici, int toplam)

```

Birinci metot ile, metodu çağırılan yazı, ayırıcı karakterlere göre parçalara ayrılır ve bu parçalar string dizisi olarak geri döndürülür. İkinci metotta ise bu parçalama işlemi en fazla toplam sayısı kadar yapılır.

Split() metodunun kullanımına bir örnek verelim. Yukarıda verilen

```
str = "Ali, Mehmet, Sefer, Volkan, Ebru"
```

yazısındaki virgülle ayrılmış her alt karakter dizisi ayrı ayrı ekrana yazdırılıyor.

```
using System;
class Ayirma
{
    static void Main()
    {
        string str = "Ali,Mehmet,Sefer,Volkan,Ebru";
        char[] ayirici = {','};
        string[] isimler = str.Split(ayirici);
        foreach(string i in isimler)
            Console.WriteLine(i);
    }
}
```

Bu programın ekran görüntüsü aşağıdaki gibi olur.

```
Ali
Mehmet
Sefer
Volkan
Ebru
```



Split() metodu ile kullandığımız ayırıcı karakterleri istediğimiz kadar artırabiliriz. Eğer dizi boş bırakılırsa varsayılan ayırıcı olarak boşluk karakteri kabul edilir. Ayrıca Split() metodunun statik olmadığını da belirtmek gerekir.

Split() metodunun tam tersi iş yapan Join() metodu da alt karakter dizilerini belirlenen bir ayıracı göre birleştirip tek string haline getirir. Join() metodunun iki kullanım şekli aşağıdaki gibidir.

```
static string Join(string ayirici, string[] yazarlar)
static string Join(string ayirici, string[] yazarlar, int baslama, int toplam)
```

Birinci metotta yazarlar dizisindeki her yazının arasına ayırıcı yazısı eklenerek tek bir string haline çevrilir ve bu birleştirilmiş yazı ile geri dönülür. İkinci metotta ise yazarlar[baslama]dan itibaren toplam kadar yazı elemanı birleştirilir. Bu metodlardan daha çok birincisi kullanılır.

Şimdi Join() metodunun kullanımına bir örnek verelim. Bir önceki örnekte yapılan işlemler bu sefer tersten yapılacak.

```
using System;
class Birlesme
{
    static void Main()
    {
        string[] str = {"Ali", "Mehmet", "Sefer", "Volkan", "Ebru"};
        string isimler = String.Join(",", str);
        Console.WriteLine(isimler);
    }
}
```

Programın ekran görüntüsü aşağıdaki gibi olacaktır.

```
Ali,Mehmet,Sefer,Volkan,Ebru
```

Join() metodunun da statik olduğuna dikkat edin!



## Diğer String İşlemleri

Bu kısımda String sınıfının diğer önemli metodlarını inceleyeceğiz. Bu metodlar açıklamalarıyla beraber aşağıda verilmiştir.

### 1. string ToUpper()

Yazının tamamını büyük harf karakterlerine çevirir ve bu yeni yazı ile geri döner.

### 2. string ToLower()

Yazının tamamını küçük harf karakterlerine çevirir ve bu yeni yazı ile geri döner.

### 3. string (int indeks, int adet)

Yazının “indeks” nolu karakterinden itibaren “adet” sayıda karakteri yazдан siler ve bu yeni yazı ile geri döner.

### 4. string Insert(int indeks, string str)

Metodu çağrıran yazının “indeks.” elemanından sonrasında str yazısı eklenir ve oluşan bu yazı ile geri dönülür.

### 5. string Replace(char c1, char c2)

Yazında geçen bütün ‘c1’ karakterlerinin yerine ‘c2’ karakteri yerleştirir ve bu oluşan yeni yazı ile geri dönülür.

### 6. string Replace(string str1, string str2)

Yazında geçen bütün str1 yazlarını, str2 yazısı ile yer değiştirir ve bu oluşan yeni yazı ile geri döner.

7. `string Substring(int indeks)`

Metodu çağırın yazının indeks elemanından sonraki karakterlerini içeren yazıya geri döner.

8. `string Substring(int indeks, int toplam)`

Metodu çağırın yazının indeks elemanından sonraki toplam adet karakteri içeren yazıya geri döner.

## Yazılıarı Biçimlendirme

Bu kısımda yazıların ekrana veya başka bir akıma istenilen bir formatta nasıl yazılacağını göreceğiz. Öncelikle yazılar üzerinde biçimlendirme yapabilmemiz için ilgili metodun biçimlendirmeyi destekliyor olması gereklidir. `Console.WriteLine()`, `String.Format()` ve `ToString()` metodları biçimlendirmeyi destekleyen metodlardandır.

İlk olarak `WriteLine()` metodunu kullanarak yazıları nasıl biçimlendirebileceğimizi inceleyelim. Hatırlarsanız `WriteLine()` metodu ile değişkenleri yazdırırmak istediğimizde

```
Console.WriteLine("{ 0 } numaralı atlet { 1 }. oldu", numara,
sirano);
```

şeklinde yazıyorduk. Burada ekrana yazılacak yazındaki {0} ifadesine "numara" değişkeninin, {1} ifadesinin yerine de "sirano" değişkeninin geleceği bildirilmektedir. Bu değişkenleri istediğimiz kadar artırabiliriz. Şu ana kadar bilinçli bir şekilde yazı biçimlendirmesi yapmadık. Yazıları biçimlendirmek için,

```
{ degisken_no, genislik : format}
```

biçim komutunu kullanmamız gereklidir. `genislik` ve `format` belirtilmediğinde varsayılan değerler kabul edilir.

`genislik` ile yazılacak yazının en küçük boyutu belirleniyor. Eğer `genislik` 0'dan büyüğe yazılar sağa dayalı, 0'dan küçüğe sola dayalı olarak yazılır. Örneğin, aşağıdaki programın çıktısını inceleyerek nelerin döndügüne anlamaya çalışın.

```
using System;
class Formatlama
{
    static void Main()
    {
        int a = 54;
        Console.WriteLine("{ 0,10 } numara", a);
        Console.WriteLine("{ 0,-10 } numara", a);
    }
}
```

Programın çıktısı aşağıdaki gibidir.

Buradaki biçim komutu ile yazının sağa mı sola mı dayalı olacağını ve yazı için en az ne kadar alan belirleneceğini belirtti. Henüz veri tipleri ile ilgili biçimlendirme yapmadığımıza dikkat edin.

Şimdi de veri türleri için ne şekilde biçimlendirme yapılabileceğine bakalım. Bir değişkenin değerini 16 sayılık tabanında yazmak için,

```
Console.WriteLine("{ 0:X } ", 155);
```

yazmalıyız. Bu deyim ile ekrana 9B yazılacaktır. Buradaki "X" karakteri 155 sayısını 16'lı sayıma sisteminde yazdırılmasını bildiriyor. Her format belirleyicisi için duyarlılık ifadesi de vardır. Bu ifadeler format belirteceden hemen sonra yazılır. Örneğin "X" belirteci için duyarlılık maximum sayıda karakter anlamına gelmektedir. Örneğin:

```
Console.WriteLine("{ 0:X5 } ", 155);
```

ifadesi ile ekrana

0009B

yazdırılır.

Her format belirleyicisi için bu duyarlılık farklı anamlara gelmektedir. Aşağıda en sık kullanılan format belirleyicileri ve duyarlılıklarının anamları mevcuttur.

Belirleyici	Açıklama	Duyarlılık Anlamı
C / c	Para birimi	Ondalık basamakların sayısını verir.
D / d	Tamsayı verisi	En az basamak sayısını belirtir, gerektiğinde boş olan basamaklar sıfırla beslenir.
E / e	Bilimsel Notasyon	Ondalık basamak sayısını verir.
F / f	Gerçek sayılar(float)	Ondalık basamak sayısını verir.
G / g	E ve F biçimlerinden	Ondalık basamak sayısını verir.hangisi kısa ise o kullanılır
N / n	Virgül kullanarak	Ondalık basamak sayısını verir.gerçek sayıları yazar.
P / p	Yüzde	Ondalık basamak sayısını verir.
R / r	Stringe dönünen türün tekrar eski değerine geri dönmesini sağlayabilecek biçim	Yok
X / x	Onaltılık sayı sisteminde yazar.	En az basamak sayısını belirtir, gerektiğinde boş olan basamaklar sıfırla beslenir

Şimdi de bu biçimlendirmelerin sonucunu görmek için aşağıdaki test programını yazalım.

```
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 çıktısı aşağıdaki gibi olacaktır.

```
D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum10\kod>Formatlama
105,000 TL
105
5.689E+002
568.8700
105
568.9
%10.500,00
00069
H:\cskitap\kitap\Bolum10\kod>
```

`Console.WriteLine("{0:N1}", f);` biçimlendirmesinden sonra duyarlılıktan dolayı `f` sayısının ondalık kısmı bir basamak yuvarlatılmıştır.

## String.Format() ve ToString() Metotları ile Biçimlendirme

String sınıfının Format() metodu tipki WriteLine() metodu gibi çalışmaktadır. Tek farkı WriteLine biçimlendirdiği yazılımı ekrana yazarken Format() metodu yazılımı bir string nesnesine aktarır. String.Format() metoduna örnek verelim:

```
using System;

class Formatlama
{
    static void Main()
    {
        int a = 50;

        string str = String.Format("{0:C3}", a);
        Console.WriteLine(str);
    }
}
```

Programı derleyip çalıştırığınızda ekrana 50,000 TL yazdığını göreceksiniz.

Format(), statik bir metot olduğu için nesne tanımlamadan metodu kullanabildik.

ToString() metodunda ise biçimlendirme komutu parametre olarak girilir. Bir önceki örneği ToString() metodu ile aşağıdaki gibi yapabiliyoruz.

```
using System;

class Formatlama
{
    static void Main()
    {
        int a = 50;

        string str = a.ToString("C3");
        Console.WriteLine(str);
    }
}
```

Gördüğünüz gibi değişen tek şey biçimlendirme komutunun yazıldığı yerdir.

## Tarih ve Saat Biçimlendirme

Hatırlarsanız tarih ve saat işlemlerini DateTime yapısı ile gerçekleştiriyorduk. Bu yapı nesnesi de sıkılıkla biçimlendirilir. Aşağıda tarih ve saat ile ilgili biçimlendirmeleri özetleyen programı inceleyebilirsiniz.

```
using System;

class TarihFormat
{
    static void Main()
    {
        DateTime dt = DateTime.Now;

        Console.WriteLine("d--> {0:d}", dt);
```

```

Console.WriteLine("D--> { 0:D} \n",dt);
Console.WriteLine("t--> { 0:t} ",dt);
Console.WriteLine("T--> { 0:T} \n",dt);

Console.WriteLine("f--> { 0:f} ",dt);
Console.WriteLine("F--> { 0:F} \n",dt);

Console.WriteLine("g--> { 0:g} ",dt);
Console.WriteLine("G--> { 0:G} \n",dt);

Console.WriteLine("m--> { 0:m} ",dt);
Console.WriteLine("M--> { 0:M} \n",dt);

Console.WriteLine("r--> { 0:r} ",dt);
Console.WriteLine("R--> { 0:R} \n",dt);

Console.WriteLine("s--> { 0:s} \n",dt);

Console.WriteLine("u--> { 0:u} ",dt);
Console.WriteLine("U--> { 0:U} \n",dt);

Console.WriteLine("y--> { 0:y} ",dt);
Console.WriteLine("Y--> { 0:Y} \n",dt);
}

```

Programın çıktısı ise aşağıdaki gibidir.

```

D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum10\kod>TarihFormat
d--> 25.02.2002
D--> 25 Şubat 2002 Pazartesi
t--> 03:04
T--> 03:04:09

f--> 25 Subat 2002 Pazartesi 03:04
F--> 25 Subat 2002 Pazartesi 03:04:09

g--> 25.02.2002 03:04
G--> 25.02.2002 03:04:09

m--> 25 Subat
M--> 25 Subat

r--> Mon, 25 Feb 2002 03:04:09 GMT
R--> Mon, 25 Feb 2002 03:04:09 GMT

s--> 2002-02-25T03:04:09

u--> 2002-02-25 03:04:09Z
U--> 25 Subat 2002 Pazartesi 01:04:09

y--> Subat 2002
Y--> Subat 2002

H:\cskitap\kitap\Bolum10\kod>

```

## Özel Biçimlendirme Oluşturma

Standart biçimlendirme komutlarının yanısıra önceden tanımlanmış özel karakterler yardımıyla kendi biçimlerimizi oluşturabiliriz. Bu özel biçimlendirici karakterler aşağıda verilmiştir.

- # → Rakam değerleri için kullanılır.
- ,
- .
- 0 → Yazılacak değerin başına ya da sonuna 0 karakteri ekler.
- % → Yüzde ifadelerini belirtmek için kullanılır.
- E0, e0, E+0,e+0,E-0,e-0 → Yazılı bilimsel notasyonda yazmak için kullanılır.

Aşağıda bazı özel biçimlendirmeler için örnekler verilmiştir.

```

using System;

class OzelBicimlendirme
{
    static void Main()
    {
        Console.WriteLine("1- ){ 0:#,##} ",1554785);
        Console.WriteLine("2- ){ 0:#.##} ",1554.785);
        Console.WriteLine("3- ){ 0:#,##E+0} ",1554785);
        Console.WriteLine("4- ){ 0:#%} ",0.25);
    }
}

```

Programı çalıştırduğumızda aşağıdaki ekran görüntüsü elde edilir.

```

D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum10\kod>OzelBicimlendirme
1- >1.554.785
2- >1554.79
3- >1.555E+3
4- >25%
H:\cskitap\kitap\Bolum10\kod>

```

## Düzenli İfadeler (Regular Expressions)

Düzenli ifadeler (regular expressions) değişken sayıda karakter dizilerinden olabileceği ancak belirli koşulları sağlayanabilen ifadelerdir. Düzenli ifadeler programdaki ihtiyaca göre düzenlenir. Diyelim ki bir text dosyası içinde @ karakteri geçen bütün satırları elde etmek istiyoruz. Burada satırındaki karakterlerin uzunluğu ve ne olduğu önemli değil; yeter ki @ karakteri olsun. Belirtilen bu satırları elde etmenin çeşitli yolları olabilir. Ancak şartlarımız arttıkça işlemi koda dökmek zorlaşacaktır. Örneğin milyonlarca e-mail adresi olabilir. Ama bir tane e-mail adresi formatı vardır. Her e-mail adresi mutlaka @ karakteri ve en az bir ? karakteri içermelidir. Eğer birden fazla

nokta varsa, noktalardan biri mutlaka @ karakterinden sonra olmalıdır. Gördüğünüz gibi bir karakter dizisinin gerçek bir e-mail adresi olup olmadığını test etmek bir hayli zor. Bu yüzden C#’ta bu tür düzenli ifadeleri temsil etmek için Regex sınıfı geliştirilmiştir. Regex sınıfı System.Text.RegularExpressions isim alanında bulunmaktadır. Bir karakter dizisinin, oluşturulan düzenli ifadeye uyup uymadığını belirlemek için ise yine aynı isim alanında bulunan Match sınıfı siniftan faydalansılır.

Düzenli ifadeler başlı başına bir kitap olacak konudur. Bu konu üzerinde yazılmış kitaplar zaten mevcuttur. Biz burada temel hatlarıyla düzenli ifadelerin ne olduğunu ve birkaç temel uygulama ile düzenli ifadelerin nasıl yazılabileceğini göreceğiz.

Regex ve Match sınıflarının kullanımına geçmeden önce düzenli ifadelerin hazırlanmasını inceleyelim.

## Düzenli İfadelerin Oluşturulması

1. Bir düzenli ifadenin satır başında mutlaka istenilen bir karakter ile başlanması isteniyorsa ^ karakteri kullanılır. Örneğin ^9 düzenli ifadesinin anlamı yazının mutlaka 9 ile başlaması demektir. “9Abcf” yazısı bu düzenli ifadeye uyarken “dasA” yazısı uymamaktadır.

2. Belirli karakter gruplarını içermesi istenen düzenli ifaderler için \ karakteri kullanılır. Aşağıda bunlara örnekler verilmiştir.

\D ifadesi ile yazının ilgili yerinde rakam olmayan tek bir karakterin bulunması gereği belirtilir.

\d ifadesi ile yazının ilgili yerinde 0-9 arasında tek bir sayının olacağı belirtiliyor.

\W ifadesi ile alfanümerik olmayan karakterin olması gereği bildiriliyor. Alfanümerik karakterler a-z, A-Z ve 0-9 aralıklarındaki karakterlerdir.

\w ile yazındaki ilgili yerde sadece alfanümerik bir karakterin olabileceği belirtilir.

\s ifadesi ile yazının ilgili yerinde boşluk karakterleri (tab, space) dışında herhangi bir karakterin olabileceği bildiriliyor.

\s ifadesi ile ilgili yerde sadece boşluk karakterlerinden birinin olabileceği bildiriliyor.

Şu ana kadar gördüğümüz bilgiler ışığında ilk karakteri 5 ile başlayan ikinci karakteri herhangi bir sayı olan ve son karakteri de boşluk olmayan bir düzenli ifade aşağıdaki gibi gösterilebilir. (Düzenli ifadeyi sağlayacak yazı mutlaka 3 karakterli olmalıdır.)

`^5\d\s`

Yukarıdaki ifadenin tamamına **filtre** denilmektedir.

3. Belirtilen gruptaki karakterlerden bir ya da daha fazlasının olmasını istiyorsak + işaretini kullanırız. Örneğin:

`\w+`

filtresi bir ya da daha fazla sayıda alfanümerik karakterin olabileceği anlamına gelmektedir. “2ASD” yazısı bu düzenli ifadeye uyarken “@Asc” yazısı uymaz. Çünkü @ karakteri alfanümerik değildir.

+ işaretini yerine \* işaretini kullanırsak çarpıdan sonraki karakterlerin olup olmayacağı serbest bırakılır.

4. Birden fazla karakter grubundan bir ya da birkaçının ilgili yerde olabileceği belirtmek istiyorsak mantıksal veya () operatörünü kullanırız. Örneğin:

`m | n | s`

düzenli ifadesi ile ilgili yerde sadece m, n ya da s karakterinin bulunabileceği bildirilir. Bu ifadeyi parantez içine alıp sonunda + işaretini koyarsak bu karakterlerden bir ya da birkaçının bulunabileceği belirtmiş oluruz:

`(m | n | s) +`

5. Sabit sayıda karakterin olmasını istiyorsak {adet} şeklinde belirtmeliyiz. Örneğin:

`\d{ 3 } -\d{ 5 }`

düzenli ifadesi ile “215-69857” yazısı sağlanır. Ama “54-56875” yazısı bu düzenli ifadeyi sağlamaz. Aradaki “-” işaretinin de mutlaka olması gerekir.

6. ? karakteri, kullanıldığı yerde önüne geldiği karakter en fazla bir en az sıfır defa olabileceği bildirir. Örneğin:

`\d{ 3 } ? A`

düzenli ifadesine “548A” ve “875BA” uyarken “478BBA” uymaz.

7. . İşareti ile ilgili yerde ‘\n’ karakteri dışında herhangi bir karakter bulunabilir. Örneğin:

`\d{ 3 } . A`

düzenli ifadesine “587sA”, “574AA”, “8957A” yazıları uymaktadır.

8. \b ile bir kelimenin belirtilen karakter dizisi ile sonlanması gerekiği bildirilir. Örneğin:

`\d{ 3 } dır\b`

düzenli ifadesine “584dır” ve “dsa325dır” yazıları uyarken “sda985dır8” yazısı uymaz.

9. \B ile bir kelimenin başında ya da sonunda olmaması gereken karakterler bildirilir. Örneğin:

```
\d{3} dir\b
```

düzenli ifadesine "584dir" ve "dsa325dir" yazıları uymazken, "sda985dir8" yazısı uyar.

10. Köşeli parantezler kullanarak bir karakter aralığı da belirtebiliriz. Örneğin, ilgili yerde sadece büyük harf karakterlerinin olmasını istiyorsak [A-Z] şeklinde kullanmalıyız. Aynı şekilde küçük harf karakterleri için [a-z] kullanabiliriz. Bu aralık ilk ve son karakterler olmayabilir; örneğin [A - P] ifadesi ile A ile P arasındaki karakterler alınır. Bu ifadeler sayılar için de geçerlidir. Örneğin [0-9] gibi.

Bu temel ifadeleri gördükten sonra Regex ve Match sınıfları ile elimizdeki yazıların düzenli ifadelere uyup uymadığını nasıl bulacağımızı inceleyelim.

Regex sınıfı bir düzenli ifadeyi tutar. Bir Regex nesnesi oluşturmak için aşağıdaki yapıcı metot kullanılabilir.

```
Regex rgx = new Regex(string filtre)
```

filtre parametresi yukarıda anlatılan düzenli ifadeleri temsil etmek için kullanılan sembollerden oluşan bir yazıdır. Regex sınıfının Match() metodu kendisine gönderilen bir yazının düzenli ifadeye uyup uymadığını kontrol eder ve uyan sonuçları Match sınıfından bir nesne ile geri döndürür. Match sınıfının NextMatch() metodu ise verilen yazida bulunan bir sonraki düzenli ifadeyi döndürür. Yazının düzenli ifadeye uyup uymadığının denetimi ise Match sınıfının Success özelliği ile yapılır. Eğer düzenli ifadeye uygun bir yapı varsa Success özelliği true olur.

Düzenli ifadelerle ilgili diğer sınıf ise MatchCollection sınıfıdır. Bu sınıf ile bir yazı içerisinde düzenli ifadeye uyan bütün Match nesneleri tutulur. MatchCollection nesnesi aşağıdaki gibi oluşturulabilir.

```
MatchCollection mc = Regex.Matches(str,filtre);
```

Burada Regex sınıfının statik Matches() metodu kullanılmıştır. Bu metodun ilk parametresi kontrol edilmek istenen yazı, ikinci parametre ise düzenli ifadenin kendisidir. Bir MatchCollection nesnesi oluşturulduktan sonra foreach döngüsü yardımıyla bu koleksiyondaki bütün Match nesnelerine erişebiliriz. Match nesnesine eriştikten sonra düzenli ifadeye uyan karakter dizisinin orijinal yazıldığı yerini ve yazının kendisini ToString() metodunu kullanarak elde edebiliriz. MatchCollection sınıfının Count özelliği ile düzenli ifadeye uyan alt karakter dizilerinin sayısı verilir. Eğer Count özelliği sıfır ise düzenli ifadeye uyan yazı bulunamadı demektir.

Şimdi bu anlatılanları bir örnekle pekiştirelim: Düzenli ifademiz aşağıdaki gibi olsun.

```
A\d{3}(a|o)+
```

Bu düzenli ifade ile başlangıcı A karakteri ile olan ve bu karakterden sonra 3 tane rakam sonra da 'a' ya da 'o' karakterinden bir ya da birden fazla sayıda olan karakter grubu doğru kabul edilir. Aşağıdaki programı inceleyin. Değişik yazılar girerek düzenli ifadelerin mantığını öğrenmeye çalışın.

```
using System;
using System.Text.RegularExpressions;

class DuzenliIfadeler
{
    static void Main()
    {
        string filtre = @"A\d{3}(a|o)+";
        string str = "";

        while(true)
        {
            Console.Write("Yazi girin: ");
            str = Console.ReadLine();

            if(str.ToLower() == "q")
                break;

            DuzenliIfadeBul(str,filtre);
        }
    }

    public static void DuzenliIfadeBul(string str,
                                         string filtre)
    {
        MatchCollection mc = Regex.Matches(str,filtre);

        if(mc.Count == 0)
        {
            Console.WriteLine("Düzenli ifade bulunamadi");
            return;
        }

        foreach(Match bulunan in mc)
        {
            Console.Write("Bulunan yer : ");
            Console.WriteLine(bulunan.Index);

            Console.Write("Bulunan yazı : ");
            Console.WriteLine(bulunan.ToString());
        }
        Console.WriteLine();
    }
}
```

Programdaki filtreyi istediğiniz gibi değiştirip denemeler yapabilirsiniz. Bir yazı içinde düzenli ifadeye uyabilecek birden fazla karakter grubu olabilir. Bu karakter gruplarının tamamına erişebilmek için Match sınıfının NextMatch() metodu kullanılır. Match sınıfının ToString() metodunu ise düzenli ifadeye uyan karakter grubunu döndürür. Match sınıfının diğer önemli özelliği ise Index özelliğiidir. Bu özellik düzenli ifadeye uyan her karakter grubunun yazı içinde kaçinci karakterden itibaren bulunduğu gösterir.

Yukarıdaki programın örnek bir çıktısı aşağıda verilmiştir.

```

D:\WINNT\System32\cmd.exe
H:\eskitap\kitap\Bolum10\kod>Regex
Yazi girin: ds0kja0645ao@587oaa
Bulunan yer : 6
Bulunan yazi : @645ao
Bulunan yer : 12
Bulunan yazi : @587oaa

Yazi girin: A58ao
Düzenli ifade bulunamadi
Yazi girin: q

H:\eskitap\kitap\Bolum10\kod>

```



Yazıyı girdikten sonra ekrana yazılanlar girilen yazının düzenli ifadeye uyan kısımlarıdır.

## Düzenli İfadelerin Gruplanması

Bir düzenli ifadeye uyan alt karakter dizisini çeşitli gruplara ayıralım. Örneğin bir email adresini düşünün. Geçerli bir e-mail adresi kullanıcı@csharpnedir.com şeklinde olabilir. Bir yazı içinde email adreslerini bulduktan sonra bu e-mail adresindeki kullanıcı adını, domain ismini (csharpnedir) ve domain tipini (com,com.tr,org vs) elde edebiliriz. Düzenli ifadeleri grüplamak için parantezler kullanılır. Her bir grup için ayrı bir parantez kullanılmalıdır. Şimdi bir e-mail adresini doğrulayacak düzenli ifadeyi yazalım. Düzenli ifademizde 3 ayrı grup olacaktır.

`\b(\w+) @(\w+) \. (\w+)`



Bu ifade ile sadece bir e-mail adresinin formatının doğru olup olmadığı tespit edilebilir. Gerçek uygulamalarda bu ifadenin daha gelişmiş versiyonları kullanılır. Yukarıdaki düzenli ifadede gruplar parantez içinde belirtilmiştir. \b ile kullanıcı adının tek bir kelime olmasını sağlıyoruz. \w+ ile ilgili yerde sadece alfanümerik karakterlerin bir ya da birden fazla bulunabileceğini belirtiyoruz. \. ile ilgili yerde ‘.’ karakterinin olması gerektiğini bildiriyoruz. Hatırlarsanız ‘.’ karakterinin düzenli ifade de özel bir anlamı vardı. Bu yüzden nokta karakteri \. şeklinde gösterilir.

Şimdi bir yazı içindeki e-mail adreslerini elde edip her bir gruba nasıl ulaşacağımızı inceleyelim. Match sınıfının Groups özelliği ile her bir gruba erişebiliriz. Match sınıfında bildirilen indeksleyici sayesinde her bir gruba Groups[1],Groups[2] şeklinde erişebiliriz. Aşağıda grüplama ile ilgili bir örnek verilmiştir. Bir önceki program ile hemen hemen aynıdır. Sadece bulunan e-maillerin her bir gruptaki yazıları da ekrana yazdırılmıştır.

```

using System;
using System.Text.RegularExpressions;

```

```

class Gruplama
{
    static void Main()
    {
        string filtre = @"\b(\w+) @(\w+) \. (\w+)";
        string str = "";
        while(true)
        {
            Console.Write("Yazi girin: ");
            str = Console.ReadLine();
            if(str.ToLower() == "q")
                break;
            DuzenliIfadeBul(str,filtre);
        }
    }

    public static void DuzenliIfadeBul(string str, string
filtre)
    {
        MatchCollection mc = Regex.Matches(str,filtre);
        if(mc.Count == 0)
        {
            Console.WriteLine("Düzenli ifade bulunamadi");
            return;
        }
        foreach(Match bulunan in mc)
        {
            Console.Write("e-mail : ");
            Console.WriteLine(bulunan.ToString());
            Console.Write("Kullanici : ");
            Console.WriteLine(bulunan.Groups[1].ToString());
            Console.Write("Domain : ");
            Console.WriteLine(bulunan.Groups[2].ToString());
            Console.Write("Domain Tipi : ");
            Console.WriteLine(bulunan.Groups[3].ToString());
        }
        Console.WriteLine();
    }
}

```

Programa yazı olarak cs@csharpnedir.com email adresini girdiğimizde aşağıdaki ekran görüntüsünü elde ettim.

```

D:\WINNT\System32\cmd.exe
H:\eskitap\kitap\Bolum10\kod>Gruplama
Yazi girin: .... cs@csharpnedir.com ....
e-mail : cs@csharpnedir.com
Kullanici : cs
Domain : csharpnedir
Domain Tipi: com

Yazi girin: cs @ csharpnedir .com
Düzenli ifade bulunamadi
Yazi girin: q

H:\eskitap\kitap\Bolum10\kod>

```



e-mail için yazdığımız bu düzenli ifade e-mail doğrulama için kesin bir çözüm değildir. Konunun ana amacından uzaklaşmamak için düzenli ifadenin yapısı basit tutulmuştur.

Regex sınıfının önemli ve kullanışlı diğer metodları ise Split() ve Replace() metodlarıdır. Split() metodu, bir yazılı belirlenen bir düzenli ifadeye göre parçalara ayırır ve bütün bu parçaları string türden diziye aktarır bu dizinin referansı ile geri döner. String sınıfının Split() metoduna çok benzemektedir. Aşağıda Split() metodunun kullanımına bir örnek verilmiştir.

```
using System;
using System.Text.RegularExpressions;

class Split
{
    static void Main()
    {
        string filtre = ",+";
        string str = "98,78,,7,987,87,,7";
        Regex rx = new Regex(filtre);
        string[] parcalar = rx.Split(str);
        foreach(string s in parcalar)
            Console.WriteLine(s);
    }
}
```

Programı çalıştırığınızda str içindeki sayılar alt alta yazdırılacaktır. Yazıyı parçalara ayırma işlemini bir ya da daha fazla virgül karakterine göre yaptığına dikkat edin. Bu işlemi sağlayan ise filtrenin “,” şeklinde olmalıdır.

Replace() metodu ise düzenli ifadeye uyan karakter dizilerini başka bir yazı ile yer değiştirmek için kullanılır. Aşağıda Replace() metodunun kullanımına bir örnek verilmiştir.

```
using System;
using System.Text.RegularExpressions;

class Replace
{
    static void Main()
    {
        string filtre = @"\d+-\d+";
        string str = "598-874 deneme yazısı 7-95";
        string Yenistr;
        Regex rx = new Regex(filtre);
        Yenistr = rx.Replace(str, "****");
        Console.WriteLine(Yenistr);
    }
}
```

Programı çalıştırığınızda “598-874” ve “7-95” yazılarının yerine “\*\*\*\*” yazısının geldiğini göreceksiniz.

## Nesne Yönelimli Programlama ve Kalıtım

- Nesne Yönelimli Programlama Neden Önemli?
- Nesne Yönelimli Programlama
- Diğer Programlama Teknikleri
- Nesne Kavramı
- Sınıf Kütüphanesi Oluşturma
  - Sınıf Kütüphanesini Kullanma
- Kalıtım (Inheritance)
  - Türetmenin Yapılması ve Temel Kavramlar
  - İsim Saklama (Name Hiding)
  - Temel ve Türeyen Sınıf Nesneleri
- Sanal Metotlar
- Özeti (Abstract) Sınıflar
- sealed Anahtar Sözcüğü
- Arayüzler (Interface)
  - Arayüz Bildirimi
  - Arayüzlerin Uygulanması
  - Arayüz Referansları
  - Açık (Explicit) Arayüz Uygulama
- Partial (Kısmi) Tipler

*Modern programlama dillerinin çoğu nesne yönelimli programlama teknigini destekler hale geldi. Nesne yönelimli programlama teknigi, yazılım geliştirme aşamasını oldukça kısaltan ve sistematik hale getiren bir tekniktir. C# dilinin de %100 nesne yönelimli olduğunu düşünürsek bu bölümün önemi daha da artmaktadır. Bu bölümde kısaca nesne yönelimli programlama teknigini anlatacağız, ardından nesne yönelimli programlama tekniginin en önemli yapı bloklarından olan kalıtım (inheritance) ve çok biçimliliği (polimorfizm) inceleyeceğiz. Bölümün sonunda ise diğer bir yapı olan arayüz (interface) kavramı üzerinde duracağız.*

## Nesne Yönelimli Programlama Neden Önemli?

1960'lı yıllarda genellikle küçük uygulamalar geliştiriliyordu. Bu yüzden kullanılan diller Assembly, Fortran ve Cobol gibi basit dillerdi. Bu diller bazı özel uygulamalar için özelleştirilmiş olduğu için kendi alanlarında büyük başarılar sağlayabiliyorlardı. Ancak gün geçtikçe teknoloji hızla gelişti, o dönemde yazılan programlar artık isteklere cevap vermemekteydi. Üstelik yeni ihtiyaçlara yönelik programlar geliştirmek gittikçe zorlaşıyordu. Daha fazla özelliğe sahip yeni dillere ihtiyaç duyulmaya başlandı. Bu yeni diller daha genel amaçlı olmalıydı. Tam bu noktada günümüzde de halen kullanımı yoğun bir şekilde devam eden C programlama dili ortaya çıktı. C dili ile artık fonksiyon mantığını kullanarak hem sistemli programlama yazılabilirdi hem de aynı fonksiyonlar değişik programlarda tekrar kullanılabilirdi.

C dili güçlü bir dil olmasına rağmen bir sistemi tüm yönleri ile incelemek C dilinde imkansız hale gelmiştir. Bu, özellikle Internet çağının dediğimiz günümüzde temel bir gereksinim halini almıştır. Çünkü programlar artık sabit bir makinada çalışan yalnız programlar değildir. Birçok hizmet verebilecek ve birçok kanalı destekleyebilecek programların olduğunu düşünürsek C dilinin de ihtiyaçlara etkin ve hızlı bir şekilde cevap veremediğini görüyoruz. Bu C dilinin acizliğini ya da kötü olduğunu asla göstermez. Burada değişen tek şey günün ihtiyaçları ve bunun neticesinde ortaya çıkan program geliştirmedeki hızlığının zorunlu hale gelmesidir.

C ve benzeri diller yapısal programlama tekniğini desteklerler; bu diller bilimsel uygulamalarda büyük başarılar sağlamalarına rağmen büyük çaplı projelerde özellikle günümüz projelerinde pek fazla başarı sağlamamaktadırlar. 80'li yılların ortalarında geliştirilen ilişkisel veri modelleme teknikleri sayesinde nesne yönelimli programlama tekniğinin temeli atılmış oldu. Nesne yönelimli programlama tekniğine uygun bir dil olarak Smalltalk ortaya çıktı. Ancak en büyük gelişme C dilinin devamı gibi görünen C++ dilinin ortaya çıkmasıyla oldu. Çünkü C++, hem klasik C'de olduğu gibi yapısal programlamaya imkan tanıyor hem de çağın yeni teknigi olan nesne yönelimli programlama tekniğine tam destek veriyordu. Nitekim günümüzde kadar birçok iş uygulamasında ve bilimsel uygulamalarda C++ dili tercih edilmeye başlandı.

Kullanımı C++ dili ile ivme kazanan nesne yönelimli programlama tekniği, artık dünya üzerinde kabul gören en yaygın teknik olmaya başladı. Bu yüzden dil tasarımcıları, dillere nesne yönelimi desteği vermek zorunda kalmıştı. Hatta bazı özel amaçlı küçük programlama dilleri bile bu teknikten faydalananmaktadır. C++ dilinin bu başarısı nesne yönelimli diğer programlama dillerinin ortayamasına yol açtı. Bunların en başında JAVA dili gelmektedir. JAVA ile C++ dilinden farklı olarak Internet uygulamalarına ağırlık verilmiş ve daha yaygın bir kullanım hedeflenmiştir. Nitekim JAVA ve C++, zamanla nesne yönelimli programlama tekniğini destekleyen en önemli ve en çok kullanılan iki dil halini almıştır.

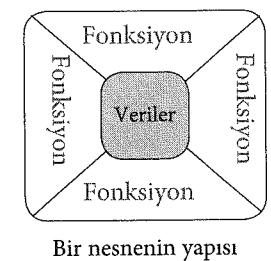
Bütün bu gelişmelerin yanında teknolojinin büyünün bir ivme ile ilerlemesi beklenenlerdeki artışları da beraberinde getiriyordu. Her ne kadar C++ ve JAVA dilleri halen büyük bir yoğunlukla kullanılıyor olsa da, bu dillerin de önemli dezavantajları vardır. Bunu gören Microsoft firması yeni bir dil olarak C#'ı geliştirdi. C# dili, C++ ve JAVA gibi nesne yönelimli programlama tekniği desteği veriyor. Bu üç dili karşılaştırarak, C#'ta nesne yönelimli programlama desteğinin yüzde yüz olduğu söylenebilir.

Günümüzün popüler dillerinin hemen hepsi nesne yönelimli programlama tekniğini desteklediğini göz önünde bulundurarak bu tekniğin ne kadar önemli ve gerekli olduğunu anlayabiliriz. Peki, nesne yönelimli programlama nedir ve neden önemlidir? Bundan sonraki başlıkta bu sorunun cevabı verilecektir.

## Nesne Yönelimli Programlama

Nesne yönelim teknigi, gerçek hayatı programlar için simule edecek yöntemler topluluğudur. Her nesnenin kendine has özellikleri vardır. Nesneler tamamen birbirlerinden soyutlanarak farklılaştırılır. Tabi her nesnenin birbirine mesaj göndermesi de son derece doğaldır. Bu gerçek yaşamda da böyledir. Nesne yönelimli programlama tekniginde gerçek bir sistem parçalara ayrılır ve bu parçalar arasında ilişkiler kurulur. Her bir parça hiyerarşik yapıda olabileceği gibi, parçalar birbirinden tamamen bağımsız da olabilir. Birbirinden bağımsız olan parçalar çeşitli yöntemlerle aralarında haberleşirler, bazen de büyük parçayı önceden tanımlanmış diğer küçük parçalar oluşturur. Burada önemli olan her parçayı etkili bir şekilde tasarlamaktır. Bu parçalardan kastımız daha önce anlatılan sınıf nesnelerinden başka bir şey değildir. Şimdi gelin nesne yönelimli programlama kavramlarını kısaca açıklayalım.

1. Nesne yönelimli programlama tekniğinin en küçük yapı taşı nesnelerdir (**objects**). Nesneler yapılarında veriler bulunmaktadır. Ayrıca bu veriler arasında belirli ilişkiler sağlayan fonksiyonlar vardır. Bir nesneyi yandaki gibi şekillendirebiliriz. Nesnelerin, verileri bu şekilde barındırması ve fonksiyonları içermesine **sarmalama (encapsulation)** denilmektedir.
2. Nesne içindeki veriler ve fonksiyonlar nesnenin dışarıya nasıl bir hizmet verdiği belirler. Ancak bu hizmetin ne şekilde verildiği belli değildir. Nesnenin servislerinden faydalana bilmek için nesnenin dış dünyaya sunduğu arayüzü bilmemiz yeterlidir. Nesne yönelimli teknik jargonunda buna Bilgi Saklama (**Information Hiding**) denilmektedir.
3. Nesneler birbirlerinden bağımsız olmasına rağmen aralarında mesajlaşabilirler. Hangi nesnenin hangi nesneye mesaj göndereceği derleme aşamasında belli olmayıpabilir. Bu durumda nesne yönelimli programlama tekniğinin diğer bir önemli özelliği olan **geç bağlama (late binding)** devreye girer.



Bir nesnenin yapısı

4. Derleme zamanında hangi nesnelerin hangi fonksiyonlarının kullanılacağı belli olmayabilir. Aynı şekilde nesneler arası mesajlaşmanın hangi nesneler arasında olacağı da derleme zamanında belli olmayabilir. Bu durumda nesne yönelimli programlama tekniğinin çalışma zamanında bağlama mekanizmasından faydalanyılır (*late binding*).

5. Tüm nesneler birer sınıf örneğidir. Sınıflar nesnenin özelliklerini belirler. Nesneler çalışma zamanında oluşturabileceği gibi derleme zamanında da oluşturulabilir.

6. Kalıtım yolu ile sınıflar birbirlerinden türetiliblir. Bir sınıf diğer bir sınıfın türediği zaman, türediği sınıfın bütün özelliklerini içerir. Bunun yanında kendine has özellikleri de barındırabilir. Nesne yönelimli programlama tekniğinin en önemli kavramı kalıtım yolu ile türetmedir. Kalıtım yolu ile türetilmiş sınıflar ile hiyerarşik bir sınıf organizasyonu gerçekleştirebiliriz. Bunun en güzel örneği .NET sınıf kütüphanesidir.

7. Nesne yönelimli programlama tekniğinde nesneler çok biçimli olabilir. Çok biçimlilik (*polymorphism*) kavramı da türetme ile yakından alakalıdır ve önemli bir kavramdır. Anlamı, bir nesnenin farklı şekillerde davranışabilmesidir.

## Diğer Programlama Teknikleri

Programlama teknikleri geçmişten günümüze çeşitli süreçlerden geçmiştir. Bu teknikleri kısaca sıralayalım.

### 1. Nesne Tabanlı Programlama (Object-based Programming)

Bu teknikte nesneler sıkılıkla kullanılır. Veriler nesneler içinde toplanmıştır. Fakat bütün nesneler birbirlerinden bağımsızdır. Ayrıca sınıf, kalıtım ve geç bağlama gibi kavramlar bu teknikte kullanılmaz. ADA programlama dili bu teknike en iyi örnektir.

### 2. Sınıf Tabanlı Programlama (Class-based Programming)

Nesne tabanlı programlamaya ek olarak bu teknikte sınıf kavramı ve sınıf örnekleri (*instances*) kavramı yer alır. Nesne tabanlı programlamada olduğu gibi türetme ve çok biçimlilik bu teknikte de yoktur.

### 3. Nesne Yönelimli Programlama (Object Oriented Programming)

Sınıf tabanlı programmanın bütün özelliklerinin yanında basit kalıtım kavramını da içerir. Smalltalk dili bu teknike örnek olarak verilebilir.

## 4. İleri Nesne Yönelimli Programlama (Advanced OOP)

Nesne yönelimli programlama tekniklerinin, bütün kavramları yanısıra çoklu türetme ve nesneler arasında ilişki belirleme gibi ileri teknikleri ile birlikte kullanıldığı programlama biçimidir. C++, JAVA ve C# dilleri bu teknique örnek olarak verilebilir.

## Nesne Kavramı

Nesneler gerçek hayatı gördiğimiz ya da varlığını bildiğimiz eşyalarıdır. Nesne yönelimli programlama tekniğinde sınıflar nesnelerin biçimlerini belirler. Oluşturduğumuz nesneler sınıf türlerinden nesneler olarak da adlandırılabilir. Her bir nesne kendi içinde tutarlı bir yapıya sahiptir. Yani veriler arasında sıkı bir bağ vardır. Eğer böyle olmása nesne yönelimli programlama tekniklerini verimli bir şekilde kullanamazdık.

C#'ta nesne yönelimli programlama yapısını sınıflar sağlar. Sınıfları kullanarak nesneler tanımlarız. Hatırlayacağınız gibi bir nesne aşağıdaki gibi oluşturuluyordu.

```
using System;

class Sınıf
{
    public int a;

    public int A()
    {
        return a;
    }
}

class MainMetodu
{
    static void Main()
    {
        Sınıf x = new Sınıf();
    }
}
```

Yukarıdaki programda Sınıf bir nesnenin şeklini belirlemektedir. Kısacası bir tür bilgisi saklamaktadır. *new* anahtar sözcüğünü kullanarak bu türden x isimli bir nesne tanımlıyoruz. Bu x nesnesinin bir tane veri taşıyan yapısı, bir tane de iş yapan metodu mevcuttur.

Sınıflar bir tür bilgisi olduğuna göre bu türü paketleyip istediğimiz programda kullanabilmemiz gereklidir.