Bölüm 03

Sınıflar ve Nesneler

Sınıf Nedir?

Sınıf (class) soyut bir veri tipidir. Nesne (object) onun somutlaşan bir cismidir.

Bu tanımın kendisi de çok soyut kalıyor. O nedenle bir örnekle biraz ayrıntıya inelim. *'motorlu taşıtlar'* denilince her birimizin kafasında bir kavram (sınıf) belirir. Muhtemelen ilk aklımıza gelenler motorsiklet, otomobil, otobüs, kamyon, tren gibi karada yürüyen taşıtların oluşturduğu sınıftır. Ama biraz düşününce bunlara denizde ve havada gidenleri de ekleyebiliriz. Bütün bunların ortak bir özeliği vardır. Hepsinin akaryakıtla çalışan motorları vardır. 'Motorlu taşıtlar' demek yerine, 'motorlu kara taşıtları' dersek sınıfa giren öğeleri biraz sınırlamış oluruz. Bu sınıftakilerin iki, üç, dört ya da daha çok tekerlekleri olduğunu, karayollarında yürüme yetenekleri olduğunu, yük veya insan taşıdıklarını anlarız. Motorlu taşıtlar sınıfı, motorlu kara taşıtları sınıfının bir üst sınıfıdır. Kavramı biraz daha daraltalım ve 'otomobiller' diyelim. Bu sınıf, motorlu kara taşıtları sınıfının bir alt sınıfıdır. Artık hepimizin kafasında bu sınıf oldukça iyi şekillenir. Ama hâlâ seçenekler çoktur. Otomobillerin markaları, modelleri, renkleri, motor güçleri, vb nitelikleri farklıdır. Belli bir markayı, örneğin BMW marka otomobiller sınıfını düşünürseniz, bu sınıfa ait olan araçların özeliklerini daha iyi biliyor olacaksınız. Bu sınıfı model, renk, bulunduğu ülke vb

nitelemelerle istediğiniz kadar daraltabilirsiniz. Ama bir otomobil sınıfı somut bir otomobile (bir cisim olarak) eş değildir.

Öte yandan, diyelim, Ahmet Bey'in BMW otomobili somut bir varlıktır. Bu otomobil, BMW marka otomobiller sınıfına, otomobiller sınıfına, motorlu kara taşıtları sınıfına ve motorlu taşıtlar sınıfına ait bir nesnedir, somut bir varlıktır. Bu otomobilin motor gücü, lastik ebadı, göstergelerin yeri ve biçimi gibi nitelikleri ait olduğu sınıf tarafından belirlenir.

Görüldüğü gibi, sınıf kavramı belli bir veya birden çok ortak özellikleri olan varlıkları tanılamaya yarayan soyut bir kavramdır, bir kümedir. Nesne ise bir sınıfa ait belirgin bir öğe, bir varlıktır.

Nesne Yönelimli (object oriented – OO) Programlama kavramı, bir önceki kavram olan yapısal programlama kavramının doğal bir genişlemesidir. Algol ve benzeri dillerin etkisiyle 1970 li yıllarda ortaya çıkan yapısal programlama kavramı Pascal dilinde *record*, C dilinde *struct* diye adlandırılan soyut veri tipini yarattı. Bunu şöyle açıklamaya çalışalım. Yapısal bir dilin olanaklarını kullanmadan, örneğin, bir otomobilin markasını, modelini, rengini, lastik ebadını, motor gücünü ayrı ayrı birer değişkenle belirleyebilirsiniz. Ama, belirlemek istediğiniz bütün nitelikleri tutacak değişkenleri bir araya getiren soyut bir veri yapısı kurarsanız, o yapıyı istediğiniz her otomobilin niteliklerini belirlemek için istediğiniz kadar ve istediğiniz her yerde kullanabilirsiniz. Yapı içinde tuttuğunuz değişkenler üzerinde işlemler yapmak gerektiğinde, o işi yapacak fonksiyonları programda tanımlamanız mümkündü.

1995 yılından sonra yaygınlık kazanmaya başlayan nesne yönelimli programlama, bir adım daha ileri giderek, 1970 lerde yaratılan soyut veri yapısına değişkenler yanında, o değişkenlerle işlem yapacak fonksiyonları da ekledi. Örneğin, otomobil örneğine dönersek, onun markası, modeli, rengi gibi özeliklerini değişkenlerle belirtebiliriz. Bunun yanında, kontak anahtarını çevirince motor çalışır, gaza basınca araba hızlanır, direksiyon çevrilince otomobil döner, frene basınca durur. Bunlar otomobilin yaptığı hareketler, eylemlerdir.

Bir otomobil için marka, model, renk gibi niteliklerin her birisi otomobilin bir özeliğidir. Her özeliği bir değişkenle belirleriz. Otomobilin yürümesi, hızlanması, dönmesi, durması gibi hareketler onun eylemleridir, davranışlarıdır. Otomobilin her bir eylemini bir fonksiyonla belirleriz.

Bir varlık için, istenen özelikleri tutan değişkenleri ve eylemleri belirleyen fonksiyonları içeren soyut veri yapısı bir *sınıftır*. Burada 'varlık' somut bir şey olmak zorunda değildir. Düşünsel, soyut bir varlık da olabilir.

Bu soyut yapı (sınıf) belirlendikten sonra, Ahmet Bey'in otomobilinin niteliklerini belirlemek için, o soyut tasarımdan somut bir otomobil üretmemiz gerekir. Otomobili üretince onun modelini, rengini, döşemelerini, göstergelerini vb belirgin kılabiliriz.

Sınıf Bildirimi

Bu bölümde C# dilinde sınıf bildirimini, new operatörü ile sınıftan nesne yaratmayı, sınıfın öğelerine erişimin nasıl olduğunu öğreneceğiz.

C# dilinde sınıf bildirimi (tanımı) çok kolaydır. Örneğin,

```
class Ev
{

}
```

deyimi bir sınıf bildirir. Burada `class` anahtar sözcüktür. `Ev` sınıfın adıdır. İstediğiniz adı verebilirsiniz. { } blokuna `sınıfın gövdesi` yada (sınıf bloku) diyeceğiz.

C# derleyicisi, kaynak programdaki birden çok ardışık boşluk, tab, ve satırbaşını tek bir boşluk olarak algıladığını söylemiştik. Dolayısıyla, yukarıdaki deyimi

```
class  Ev { };
```

biçiminde de yazabiliriz. Ama kaynak programımızın kolay okunup anlgılanabilmesi için her deyimi ayrı satıra yazmaya, iç-içe blokları tab ile görünür biçime getirmeye özen göstereceğiz.

Sınıf gövdesi yukarıdaki gibi boş olabilir. Ama boş gövdeli sınıfın bir işe yaramayacağı açıktır. Onun işe yarar olabilmesi için içine sabit, değişken ve fonksiyon bildirimleri (tanımları) koyacağız. Onlara `sınıfın öğeleri` (class member) denilir.

Şimdi `Ev` sınıfımızı işe yarar hale getirmeye çalışalım. Bir ev ile ilgili bilgileri tutan değişkenleri ve o bilgilerle işlem yapan fonksiyonları `Ev` sınıfının gövdesine yerleştireceğiz. Örneğin, bir evin adresi, kapı numarası, oda sayısı, evin sahibi, emlak vergisi, fiyatı vb bilgilerden istediklerimizi sınıfın gövdesine koyabiliriz. Konunun basitliğini korumak için, başlangıçta yalnızca kapı numarasını ve sokağın adını yazalım. Her evin bir kapı numarası vardır. Bu numara bir tamsayıdır. `byte, short, int` veya `long` tipinden olabilir. `int` tipini seçelim. Sınıfımız şöyle olacaktır.

```
class Ev
{
      int kapıNo ;
      string  sokakAdı ;
}
```

Bu sınıfın içine başka öğeler koymak, `kapıNo`' yu yazmak kadar kolaydır. Ona benzer işleri ileride bolca yapacağız.

Sınıf ve Nesne

Önce yapacağımız işin gerçek yaşamda neye benzediğini bir örnekle açıklamaya çalışalım. Bir ev yapmak istiyoruz. Önce evin mimari tasarımını yaparız. O tasarım evin konumunu, odalarını, mutfağını, banyosunu, pencerelerini, nasıl ısıtılacağını, nasıl havalanacağını, suyun, elektriğin, doğal gazın nerelerden nasıl geleceğini, vb belirler. Bu tasarım, büyük bir yapı için uzmanların tasarladığı ve çizdiği karmaşık bir mimari proje olabileceği gibi, bir köylünün kendi kafasında tasarladığı basit bir klübe de olabilir. İster büyük, karmaşık bir yapı, ister bir klübe olsun, önce ortada bir tasarım vardır. İster kağıt üzerine çizilsin, ister birisinin kafasında tasarlanmış olsun, o tasarım somut bir ev, bir nesne değildir. O tasarımın içine girip salonunda oturamaz, mutfağında yemek pişiremeyiz. O işleri yapabilmemiz için, her şeyden önce, tasarımı yapılan evin inşa edilmesi gerekir.

Gerçek yaşamda, bir mimari tasarımdan bir ev inşa etmek istediğimizde, bu işi yapan bir şirkete veya ustaya başvururuz. Şirket veya usta, tasarlanan evin somut bir örneğini yapıp anahtarı bize teslim eder.

Evin mimari tasarımını, C# dilinde bir sınıfa (class) benzetebiliriz. Mimari tasarımdan inşa edilen somut bir evi de C# dilinde bir sınıftan elde edilen bir nesneye (object) benzetebiliriz. Bir sınıf bir tasarımdır. Onu somut olarak kullanamayız. Onun için sınıftan nesne(ler) kurmalıyız.

Bir mimari tasarımdan aynı tipte istediğimiz kadar ev inşa edebiliriz. Örneğin bir sitedeki veya bir apartmandaki evlerin hepsi bir tek tasarımdan üretilir. Ancak, o evlerin her birisi kendi başına somut bir varlıktır, herbiri uzayda farklı bir yer işgal eder. Evler aynı yapıya sahiptirler, ama her bir evin boyası, badanası, içindeki eşyalar, insanlar bir başka evin içindekilerden farklıdır. Bu öznitelikler, bir evi ötekinden ayırır.

Benzer olarak, bir sınıftan istediğimiz kadar nesne kurabiliriz. O nesnelerin yapıları aynıdır, ama öznitelikleri farklıdır.

C# uygulamalarının mutlaka `Main()` metodu (fonksiyon) tarafından başlatıldığını biliyoruz. O halde, sınıftan nesne kurmak istediğimizde ona başvurabiliriz. `Main()` metodu kendi başına ortalıkta duramaz, o da mutlaka öteki öğeler gibi bir sınıf içinde olmalıdır. O nedenle, `Main()` metodunu içeren bir sınıf yaratmalıyız. `Uygulama` adlı boş bir sınıf yaratalım:

```
class Uygulama
{
}
```

Sonra bu sınıfın gövdesine Main() metodunu ekleyelim.

```
static public void Main()
{
}
```

new Operatörü ile Nesne Yaratmak

`Main()` metodunun gövdesine, `Ev` sınıfından bir nesne yaratacak (kuracak) olan deyimleri koyalım.

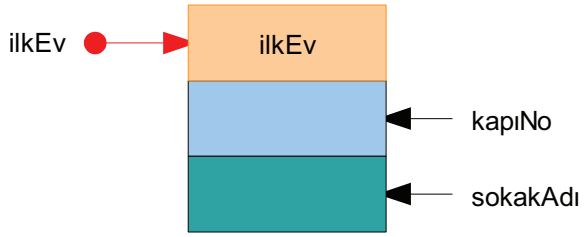
```
Ev ilkEv ;                                                    (1)
```

Bu deyim derleyiciye `Ev` sınıfına ait bir nesnenin ana bellekteki adresini gösterecek bir referans bildirimidir. Çeşitli kaynaklarda buna `pointer`, `gösterici`, `işaretçi`, `referans` gibi adlar verilir. C# dili `'reference'` sözcüğünü kullanır. Biz de ona uyarak `'referans'` veya `'işaretçi'` sözcüklerini eşanlamlı olarak kullanacağız. Bunun tek ve önemli görevi `Ev` sınıfına ait bir nesnenin bellekteki adresini işaret etmektir.

(1) bildirimi yapıldığında, henüz işaret edilecek somut bir Ev yoktur. Dolayısıyla, o aşamada, `ilkEv` işaretçisi bellekte hiç bir yeri işaret etmez. Hiç bir yeri yeri işaret etmediğini belirtmek için, işaretçiye `null` işaret ediyor deriz.

Şimdi işaretçimizin işaret edeceği somut bir Ev inşa etmeliyiz. Somut bir Ev'i inşa etmek demek, ana bellekte Ev'in bütün öğelerinin sığacağı bir yeri tahsis etmek demektir. Bu yere Ev'in bir nesnesi (object, instance) denir. Yapılan bu işe de sınıfa ait bir *nesne yaratmak (instantiate)* denilir. Bellekte sınıfa ait bir nesneyi yaratırken, bir inşaat ustası kadar uğraşmayacağız. Şu basit deyimi yazmamız yetecektir.

```
IlkEv = new Ev() ;                                            (2)
```



Bunu yapınca, ana bellekte yukarıdaki şeklin gösterdiği olaylar olur. Elbette ana bellekte böyle geometrik şekiller oluşmuyor. Ama şekiller ve resimler soyut kavramları algılamamızı kolaylaştırır. O nedenle, şeklimize bakmaya devam edelim. Ev sınıfının iki öğesini tanımlamıştık: kapıNo ve sokakAdı. Bellekte yaratılan nesneye bakarsak, orada kapıNo ve sokakAdı için ayrı ayrı yerler açılmış olduğunu görüyoruz. Bir sınıfa ait nesne (object) yaratmak demek, ana bellekte sınıfa ait static olmayan bütün öğelere birer yer ayırmak demektir. Ayrılan bu yerlere başka değerler yazılamaz; o nesne bellekte durduğu sürece, yalnızca ait oldukları öğelerin değerleri girebilir. Bunu, bellekte yer kiralama gibi düşünebiliriz. Kiralanan yerde ancak kiracı oturabilir. Henüz yaratılan nesnenin öğelerine değer atanmamıştır; yani kiracı henüz taşınmamıştır. Bize göre kapıNo ve sokakAdı değişkenleri için ayrılan hücreler boştur. Ama, C# bellekte yaratılan her değişkene kendiliğinden bir öndeğer (default value) atar. Öndeğer veri tipine göre değişir. Aşağıdaki tablo başlıca veri tiplerine atanan öndeğerleri göstermektedir.

Veri Tipi	Değer
byte, short, Int, long	0
float, double	0,0
bool	False

char	'\0' (null karakter)
string	"" (boş string)
nesne (object)	null

`KapıNo` değişkeni `int` tipi olduğu için onun öndeğeri 0 dır. `sokakAdı` ise `string` tipi olduğundan öndeğeri "" (boş string) dir. Olayın basitliğini korumak için, öndeğerleri şekle yazmıyoruz. Zaten, birazdan onların gerçek değerlerini atayacağız.

İstersek (1) ve (2) deyimlerini birleştirip, iki işi tek adımda yapabiliriz.

```
Ev ilkEv = new Ev();                                    (3)
```

Sonuncu deyim, ilk iki deyimin yaptıklarına denk iş yapar.

Bu deyime `nesne yaratıcı (instantiate)` diyoruz. Sözdizimine bakınca ne yaptığını anlamak mümkündür. Bu deyimde yer alan sözcüklerin işlevlerini soldan sağa doğru şöyle açıklayabiliriz:

Ev	Yukarıda tasarladığımız Ev sınıfıdır;
ilkEv	Tasarımdan üretilecek olan somut evin yerini işaret eder. Bu nedenle, ilkEv'i işaret ettiği evin adıymış gibi de düşünebiliriz. Bundan böyle işaretçi (referans) adı ile işaret ettiği nesneyi aynı adla anacağız. Söylemlerimizde, kastedilen şeyin referans mı, nesne mi olduğu belli olacaktır. Ancak çok gerektiğinde, referans oluşuna ya da nesne oluşuna vurgu yapacağız.
=	Atama operatörü
new	Sınıftan nesne yaratan operatör (nesne yaratıcı)
Ev()	Yaratılacak nesnenin tasarımı

Bir sınıftan bir nesne yaratan genel sözdizimi (syntax) şöyledir:

```
sınıf_adı   nesne_adı = new       sınıf_adı();
```

Şimdiye kadar söylediklerimizi yapmak için aşağıdaki Uygulama sınıfını yazmak yetecektir. Tabii, Ev sınıfını daha önce yazmıştık.

```
class Uygulama
{
    static public void Main(string[] args)
    {
        Ev ilkEv = new Ev();
    }
}
```

Şu ana kadar iki sınıf tanımladık. `Ev` adlı sınıf bir evin kapı numarası ile bulunduğu sokağı tutacak iki değişkene sahiptir. Ama `Ev` sınıfı bir tasarımdır, kendi başına bir iş yapamaz.

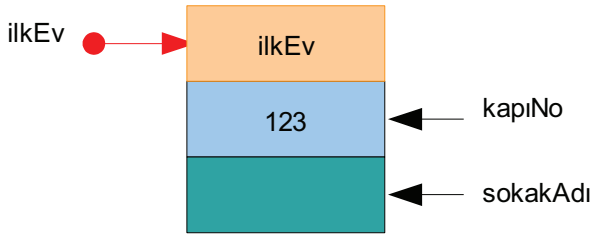
`Uygulama` adlı sınıfta `Main()` metodu tanımlıdır ve bu metot `Ev` sınıfının bir nesnesini yaratacak olan *nesne yaratıcıyı* çalıştırmaktadır. Nesne yaratıldıktan sonra, onun öğeleriyle ilgili işleri yapmaya başlayabiliriz.

Nesnenin Öğelerine Erişim

İlk işimiz nesnenin öğelerine (değişken) birer değer atamak olmalıdır. ilkEv nesnesi içindeki kapıNo ve sokakAdı bileşenleri birer değişkendir. Dolayısıyla, değişkenlerle yapılan her iş ve işlem, onlara da uygulanabilecektir. Birincisi `int` tipinden, ikincisi `string` tipinden olduğuna göre, onlara tiplerine uygun birer değer atayabiliriz:

```
ilkEv.kapıNo = 123;
```

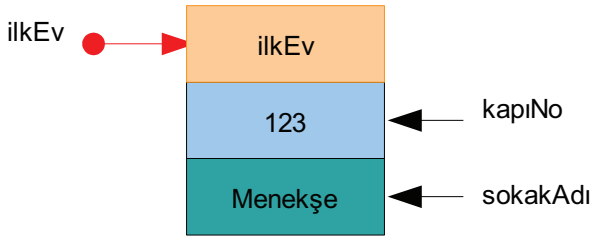
Bu atamadan sonra `ilkEv` nesnesinin durumu aşağıdaki şekildeki gibidir.



Sonra şu atamayı yapalım.

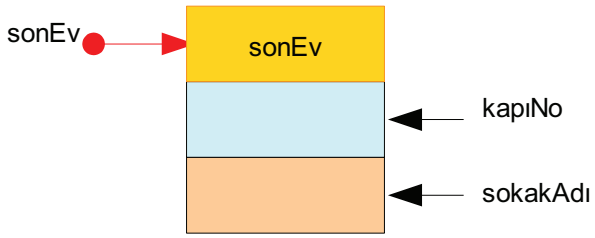
```
ilkEv.sokakAdı = "Menekşe";
```

Bu atamadan sonra `ilkEv` nesnesinin durumu aşağıdaki şekildeki gibidir.

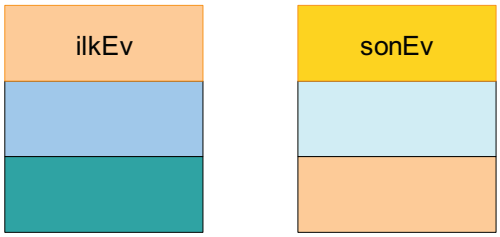


Nesne içindeki değişkenlere yapılan atama deyimlerinde, değişken adlarının önüne nesne adını yazıyoruz. Bunun nedeni açıktır. Diyelim ki,

```
Ev sonEv = new Ev();
```



nesne yaratıcısını kullanarak başka bir nesne yarattık. O zaman, bellekte iki nesne ve her birinde `kapıNo` ve `sokakAdı` bileşenleri ayrı ayrı yer alacaktır.



O durumda, yalnızca,

```
kapıNo = 123;
sokakAdı = "Menekşe";
```

atama deyimlerini yazarsak, derleyici, bu değerleri hangi ev için atayacağını bilemez, hata iletisi verir. Bu nedenle, önce nesneyi, sonra değişkeni yazıyoruz. Tabii, aralarına (.) koymayı unutmuyoruz.

```
ilkEv.kapıNo
```

ifadesi, `ilkEv` nesnesi içindeki `kapıNo` değişkeninin adıdır. Benzer şekilde,

```
sonEv.kapıNo
```

ifadesi `sonEv` nesnesi içindeki `kapıNo` değişkeninin adıdır. Görüldüğü gibi, farklı nesneler içinde aynı adı taşıyan değişkenlere farklı değerler atanabiliyor. Değişken önüne konulan nesne adları, aynı adlı değişkenleri birbirlerinden ayırır. Zaten, bir sınıftan istediğimiz kadar farklı nesne yaratabiliyor olmamızın nedeni budur. Her nesne kendi başına bir varlıktır.

Şimdiye kadar söylediklerimizi bir program haline getirelim.

Ev.cs

```
using System;

class Ev
{
    public int kapıNo;
    public string sokakAdı;
}

class Uygulama
{
    static public void Main()
    {
        Ev ilkEv = new Ev();
        ilkEv.kapıNo = 123;
        ilkEv.sokakAdı = "Menekşe";
        Console.WriteLine("ADRES: " + ilkEv.sokakAdı + " Sokak, No: " +
ilkEv.kapıNo);
    }
}
```

Çıktı

```
ADRES: Menekşe Sokak, No: 123
```

Bu programı satır satır inceleyelim. `Ev` ve `Uygulama` ayrı ayrı iki sınıftır.

Kapsülleme (encapsulation)

Ev sınıfında dikkatimizi çekmesi gereken şey sınıfın değişkenlerine `public` sıfatının verilmiş olmasıdır.

```
class Ev
{
    public int kapıNo;
    public string sokakAdı;
}
```

Değişken bildirimindeki public nitelemesini kaldırıp programı derlemeyi deneyiniz. Derleyiciden şu hata iletisini alacaksınız.

```
Error  1    'Ev.kapıNo' is inaccessible due to its protection level      ...
```

Bu ileti bize, `Ev` sınıfının `kapıNo` adlı değişkenine erişilemediğini söylüyor. Bu olgu, nesne yönelimli programlamada adına *kapsülleme* (*encapsulation)* denen iyi bir güvenlik yöntemidir. Encapsulation kavramını basite indirerek şöyle açıklayabiliriz. Bir sınıf içindeki öğelere dışarıdan erişilememesi için sınıfın dış dünyaya kapatılması demektir. Bunu ileride daha iyi anlayacağız. Bazı durumlarda sınıfın öğelerine erişimin olması istenir. Bunu yapmak için, yukarıda kullandığımız `public` nitelemesi yeterlidir. Bunun dışında başka nitelemeler de vardır. Onları *erişim belirteçleri* bölümünde açıklayacağız.

Uygulama sınıfındaki `Main()` metodu

```
Ev ilkEv = new Ev();
```

deyimi ile `Ev` sınıfından bir nesne yaratmıştır. Başka bir deyişle, ana bellekte `kapıNo` ve `sokakAdı` değişkenleri için birer yer ayırmıştır. Bellekteki bu adreslere erişmek için `ilkEv.kapıNo` ve `ilkEv.sokakAdı` işaretçileri (referans, pointer) kullanılmaktadır. Nesne yaratıcıdan sonra gelen iki deyim, yukarıda açıklanan iki atamayı yapmaktadır. Son satır ise, atanan bu değerleri konsola yazdırır.

Aşkın Operatör (overloaded operator)

Çıktıyı konsola yazdıran `Console.WriteLine()` metoduna parametre olarak

```
("ADRES: " + ilkEv.sokakAdı + " Sokak, No: " + ilkEv.kapıNo)
```

ifadesini yazdık. Bunlardan ikisi string, ikisi değişkendir; değişken değerleri de konsola string olarak yazılacaktır. Dört stringi ard arda birleştirip yazmak için (+) operatörünü kullanıyoruz. Elbette bu eylemde (+) operatörü sayılardaki işlevinden farklı bir işlev üstlenmiş, metinleri uc uca birleştirmiştir. Buna benzer olarak, bir operatöre farklı veri tipleri üzerinde farklı işlevler yaptırılabilir. Örneğin, (+) operatörüne sayısal veri tiplerinde bilinen toplama işlemini, string veri tiplerinde metin birleştirme işlemini, tarih veri tipleri üzerinde tarih toplama işlemini yaptırırız. Farklı veri tipleri üzerinde farklı işlevler yüklenmiş operatöre *aşkın operator (overloaded operator)* denilir.

Yer Tutucu Operatör

Çıktıyı konsola yazdırırken, çoğu kez, yukarıdaki gibi (+) aşkın operatörünü kullanırız. Ama çok işlevsel bir operatörümüz daha var: Yer Tutucu operatör {}. Daha önce bu parantezleri programdaki blokları belirlemek için kullandık. Bunun başka işlevleri olduğunu göreceğiz. Onlardan önemli birisi, konsol çıktılarında değişkene yer tutmaktır. Bunu bir örnekle açıklayalım. Yukarıdaki örnekte çıktıyı

```
Console.WriteLine("ADRES: " + ilkEv.sokakAdı + " Sokak, No: " +
ilkEv.kapıNo);
```

deyimi ile yazdırdık. Aynı işi

```
Console.WriteLine("ADRES: {0} Sokak, No: {1} " , ilkEv.sokakAdı ,
ilkEv.kapıNo  );
```

deyimi de yapar. "ADRES: {0} Sokak, No: {1} " stringi içerisine iki tane yer tutucu operatör konulmuştur: {0} ve {1}. Stringden sonra virgülle ayrılmış iki değişken adı vardır. C# onlardan ilkini; yani ilkEv.sokakAdı değişkenini 0-ıncı değişken olarak sayar ve onun değerini {0} yer tutucusunun bulunduğu yere koyar. Benzer olarak, ilkEv.kapıNo değişkenini 1-inci değişken olarak sayar ve onun değerini {1} yer tutucusunun bulunduğu yere koyar. Genel olarak {n} yer tutucusu, saymayı sıfırdan başlatırsak, n-inci değişkene yer ayırır. Yer tutucu operatörle, çıktıyı nasıl biçemleyebileceğimizi ileride göreceğiz. Şimdilik, aynı çıktıyı,

```
Console.WriteLine("ADRES: {1} Sokak, No: {0} " , ilkEv.kapıNo   ,
ilkEv.sokakAdı);
```

deyimiyle de yapabileceğimizi apaçık görebiliriz.

İpucu

Yukarıdaki programda 123 ve "Menekşe" değerleri ilkEv adlı nesne içinde atanmış değerlerdir. Ev sınıfından başka nesneler yaratılabilir. Yaratılacak başka nesnelerde, bu atamalar geçerli değildir. Her nesnenin öğelerine ayrı ayrı atamalar yapılmalıdır.

Başka bir ev nesnesi yaratılırsa, onun kapı numarasının ya da sokakAdı adının farklı olması zorunlu mu? sorusu akla geliyor. Tabii gerçek hayatta, aynı sokakta kapıNo'ları eşit iki evin olması postacıyı yanıltabilir. Ama, işaretçileri (adları) farklı olduğu sürece nesneye ait değişkenlerin aynı değeri almalarına hiç bir engel yoktur. Farklı nesnelerde aynı adı taşıyan değişkenler farklı ya da aynı değeri alabilirler. Örneğin Menekşe sokakta birden çok ev olabilir. O evlerin sokakAdı adları aynı olacaktır.

Bunu görmek için, ikinciEv ve üçüncüEv adlı iki nesne daha yaratalım ve programı aşağıdaki gibi değiştirelim.

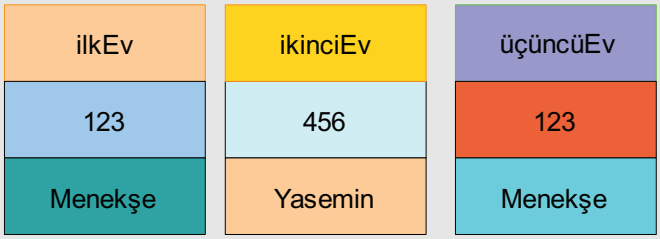
Ev02.cs

```
using System;

namespace Sınıflar
{
    class Ev
    {
        public int kapıNo;
        public string sokakAdı;
    }

    class Uygulama
    {
        static public void Main()
        {
            Ev ilkEv = new Ev();
            ilkEv.kapıNo = 123;
            ilkEv.sokakAdı = "Menekşe";

            Ev ikinciEv = new Ev();
            ikinciEv.kapıNo = 456;
            ikinciEv.sokakAdı = "Yasemin";

            Ev üçüncüEv = new Ev();
            üçüncüEv.kapıNo = 123;
            üçüncüEv.sokakAdı = "Menekşe";

            Console.WriteLine("ADRES: {1} Sokak, No: {0} ", ilkEv.kapıNo,
ilkEv.sokakAdı);
            Console.WriteLine("ADRES: {1} Sokak, No: {0} ",
ikinciEv.kapıNo, ikinciEv.sokakAdı);
            Console.WriteLine("ADRES: {1} Sokak, No: {0} ",
```

```
            üçüncüEv.kapıNo, üçüncüEv.sokakAdı);
        }
    }
}
```

Çıktı

```
ADRES: Menekşe Sokak, No: 123
ADRES: Yasemin Sokak, No: 456
ADRES: Menekşe Sokak, No: 123
```



Buradan şu kuralı çıkarabiliriz.

Kural

Bir sınıftan *nesne yaratıcı* ile istediğimiz kadar nesne yaratabiliriz. Her nesneye bellekte ayrı yerler ayrılır. Dolayısıyla, birisinin değişkenlerine atanan değerler, öteki nesneleri etkilemez.

Ama, istenirse bir nesnenin değerleri başkasına aktarılabilir. Örneğin, yukarıdaki programda, konsola yazdıran iki satırdan önce

```
            Ev yazlıkEv = new Ev();
            yazlıkEv = ilkEv;
            ilkEv = ikinciEv;
            ikinciEv = yazlıkEv;
```

deyimlerini ekleyelim. Çıktıda `ilkEv` ile `ikinciEv`'in değişkenlerine verilen değerlerin birbirleriyle yer değiştirdiğini görebiliriz.

```
ADRES: Yasemin Sokak, No: 456
ADRES: Menekşe Sokak, No: 123
ADRES: Menekşe Sokak, No: 123
```

Yukarıdaki takas işlemine biraz yakından bakalım. `yazlıkEv = ilkEv` atamasında olduğu gibi, bir nesne aynı sınıftan başka bir nesneye değer olarak atanabildi.

İpucu

Bir nesne aynı sınıftan başka bir nesneye değer olarak atanabilir.

Genkurucu (default constructor)

Şimdi şunu deneyelim. Nesne yaratıcısını etkisiz kılalım; yani bir nesne yaratılmasın. Onun yerine `Ev` sınıfının değişkenlerini `public static` sıfatlarıyla niteleyelim.

GenKurucu01.cs

```
using System;

namespace Sınıflar
```

```
{
    class Uygulama
    {
        class Ev
        {
            public static int kapıNo;
            public static string sokakAdı;
        }

        static public void Main()
        {
            Ev.kapıNo = 123;
            Ev.sokakAdı = "Menekşe";
            Console.WriteLine("ADRES: {1} Sokak, No: {0} ", Ev.kapıNo,
Ev.sokakAdı);
        }
    }
}
```

Çıktı

ADRES: Menekşe Sokak, No: 123

Bu programın nasıl çalıştığını açıklamak için kurucu (constructor) kavramına biraz daha eğilmemiz gerekiyor. Önceki programlarda `Ev` sınıfının bir kaç nesnesini yarattık, o nesnelerin değişkenlerine değerler atadık. Ama yukarıdaki programda apaçık bir nesne yaratmadık. Buna karşın,

```
Ev.kapıNo = 123;
Ev.sokakAdı = "Menekşe";
```

atamalarını yaptık. Oysa, daha önce söylediklerimize göre `Ev` sınıfı bir tasarımdır; ona bir şey yaptırılamazdı; ancak o sınıftan yaratılacak nesnelere bir iş yaptırılabilirdi. O zaman, nesne yaratılmadan yapılan yukarıdaki atamaları nasıl açıklayacağız?

Yukarıdaki sorunun basit bir yanıtı var. Daha önce öğrendiğimiz kural bozulmadı. Yalnızca bizim apaçık istemediğimiz bir işi derleyici kendiliğinden yaptı. Ev sınıfı için kendiliğinden bir nesne yarattı. O nesneye sınıfın adını verdi. Böylece, `Ev.kapıNo = 123;` deyiminin başındaki `Ev` adı sınıfın değil, bizden habersiz yaratılan ve aynı adı taşıyan nesnenin adıdır. Bunu kendiliğinden yapan nesne kurucuya `genkurucu (default constructor)` denilir.

İpucu

Program bir sınıfa ait hiç bir nesne yaratmıyorsa, genkurucu kendiliğinden o sınıfa ait bir nesne yaratır; yarattığı nesneye o sınıfın adını verir.

Genkurucu birden çok nesne yaratmaz.

Nesne yaratıcı ile bir nesne yaratılıyorsa, genkurucu nesne yaratmaz.

Sınıf değişkenlerine verilen değerleri istediğimizde değiştirebiliriz. Aşağıdaki örnek bunu göstermektedir.

GenKurucu02.cs

```
using System;

namespace Sınıflar
{
    class Uygulama
    {
```

```
        class Ev
        {
            public static int kapıNo = 444;
            public static string sokakAdı = "Kardelen";
        }

        static public void Main()
        {
            Console.WriteLine("ADRES: {1} Sokak, {0} " , Ev.kapıNo,
Ev.sokakAdı);
            Ev.kapıNo = 123;
            Ev.sokakAdı = "Menekşe";
            Console.WriteLine("ADRES: {1} Sokak, {0} ", Ev.kapıNo,
Ev.sokakAdı);
        }
    }
}
```

Çıktı

ADRES: Kardelen Sokak, 444

ADRES: Menekşe Sokak, 123

İpucu

Bir sınıfın static öğelerine nesne işaretçisi ile erişilemez.

Aşağıdaki programda Ev sınıfının `static` nitelemeli `sokakAdı` değişkenini, `ilkEv` işaretçisi göremiyor; derleme hatası doğuyor.

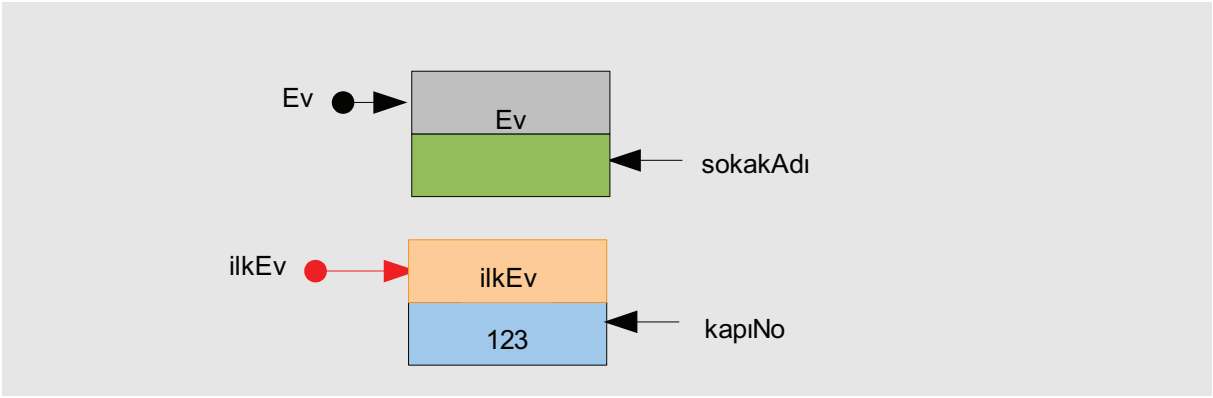
GenKurucu03.cs

```
using System;

namespace Sınıflar
{
    class Uygulama
    {
        static public void Main()
        {
            Ev ilkEv = new Ev();
            ilkEv.kapıNo = 123;
            ilkEv.sokakAdı = "Menekşe";
        }
    }

    class Ev
    {
        public int kapıNo;
        public static string sokakAdı;
    }
}
```

Error 1 Member 'Sınıflar.Uygulama.Ev.sokakAdı' cannot be accessed with an instance reference; qualify it with a type name instead ...

`Main()` metodundaki kodların yazılış sırası ile hata iletisini karşılaştırırsak şunu görebiliriz: `Ev` sınıfının `ilkEv` adlı bir nesnesi yaratılmıştır. `ilkEv.kapıNo = 123` ataması ile `ilkEv` nesnesinin `kapıNo`

adlı bileşenine `123` değeri atanmıştır. Ama `ilkEv.sokakAdı = "Menekşe"` atamasına gelindiğinde, derleyici hata vermektedir. Çünkü, Ev sınıfının `static` nitelemeli `sokakAdı` adlı öğesi bir nesne içine gitmez. Ona ana bellekte `ilkEv` nesnesi dışında bir yer ayrılır. Dolayısıyla nesne içinde `ilkEv.sokakAdı` adlı bir değişken yoktur. Bunu aşağıdaki gibi bir şekille temsil edebiliriz.



`Ev` sınıfını `Uygulama` sınıfının içine alsak bile bu durum değişmez. Böyle olduğunu görmek için, programımızı aşağıdaki gibi değiştirelim. Derleyicimizin aynı hata iletisini verdiğini göreceğiz.

GenKurucu04.cs

```
using System;

namespace Sınıflar
{
    class Uygulama
    {
        static public void Main()
        {
            Ev ilkEv = new Ev();
            ilkEv.kapıNo = 123;
            ilkEv.sokakAdı = "Menekşe";
        }

        class Ev
        {
            public int kapıNo;
            public static string sokakAdı;
        }
    }

}
```

```
Error  1  Member 'Sınıflar.Uygulama.Ev.sokakAdı' cannot be accessed with an instance reference;
```

Bir sınıfın bazı öğeleri static, bazıları değilse, *static öğelere* sınıf işaretçisi ile, *static olmayan öğelere* de nesne işaretçisi ile erişilebilir.

GenKurucu05.cs

```
using System;

namespace Metotlar
{
```

```
    class Uygulama
    {
        static public void Main()
        {
            Ev.kapıNo = 777;
            Ev eskiEv = new Ev();
            eskiEv.sokakAdı = "Yasemin";
            Console.WriteLine("ADRES: {1} Sokak, {0} ", Ev.kapıNo,
eskiEv.sokakAdı);
        }
    }

    class Ev
    {
        public static int kapıNo;
        public string sokakAdı;
    }
}
```

Çıktı

```
ADRES: Yasemin Sokak, 777
```

this *anahtarı*

Her zaman söylediğimizi anımsatarak başlayalım. C# dilinde her şey sınıflar içinde tanımlanır. Dolayısıyla, bazı dillerde önemli rol oynayan *global değişkenler* ve *global fonksiyonlar* C# dilinde yoktur. Onun yerine *sınıf öğeleri* (class member) dediğimiz değişkenler ve metotlar vardır. Sınıfın bir öğesine 'global' rol oynatmak istediğimizde, onu *public* erişim belirteci ile niteleriz; yani public nitelemeli sınıf öğelerine başka sınıflardan erişilebilir. Metotların içinde tanımlanan değişkenler o metodun yerel değişkenleridir. Yerel değişkenlere ancak ait olduğu metot içinden erişilir; kendi sınıfından veya başka sınıflardan erişilemez.

Bazı durumlarda, yerel değişken adları ile sınıf değişkenlerinin adları aynı olabilir. Bu durumda, metotlar kendi yerel değişkenlerine öncelik tanırlar. Bunu bir örnekle açıklayalım. Örneğin dairenin alanını bulan bir sınıfta `yarıÇap` adlı bir değişken hem sınıf öğesi olarak hem de bir metodun yerel değişkeni olarak bildirilmiş olsun. C#, bunları farklı iki değişken olarak yorumlar. Dolayısıyla, her ikisine ana bellekte ayrı ayrı yerler ayırır. Öyleyse, derleyici açısından onlar birbirlerinden farklı iki değişkendir.

Metotlar sınıf öğelerini görebildiğine göre, metot içinde `yarıÇap` değişkenine bir atama yapılsa, o değer hangisinin adresine yazılacak? Sınıf değişkeninin adresine mi, yoksa yerel değişkenin adresine mi? Benzer olarak, `yarıÇap` değeri okutulmak istense hangi adresteki değeri okuyacak? Derleyici bu konuda hiç tereddüde düşmez, aksi söylenmedikçe öncelik daima yerel değişkenlerindir. Böyle olduğunu aşağıdaki örnekten görebiliriz.

Daire.cs

```
using System;
class Daire
{
    public const double PI = 3.14159;
    public double yarıÇap;
    public double alan;

    public double AlanBul(double r)
    {
        double yarıÇap = r;
        double alan;
        alan = PI * r * r;
        return alan;
```

```
    }

    public void Yaz()
    {
        Console.WriteLine("Dairenin yarıçapı = {0} ", yarıÇap);
        Console.WriteLine("Dairenin alanı    = {0} ", alan);
    }
}

class Uygulama
{
    static void Main(string[] args)
    {
        Daire d = new Daire();
        Console.WriteLine(d.AlanBul(2.5));
        d.Yaz();
    }
}
```

Çıktı

```
19,6349375
Dairenin yarıçapı = 0
Dairenin alanı    = 0
```

Çıktıya bakarak bu programı çözümleyelim. `Daire` sınıfı içinde double tipinden public nitelemeli `yarıÇap` ve `alan` adlı iki *sınıf değişkeni* bildirimi yapılmıştır. Sınıfa ait metotlar ve başka sınıflara ait metotlar bu değişkenlere erişebilirler. Nitekim, `Daire` sınıfı içinde tanımlanan `Yaz()` metodu bu değişkenlere erişmekte ve onların değerlerini konsola yazmaktadır.

Gene `Daire` sınıfı içinde tanımlı olan `AlanBul()` metodu, çağrılırken aldığı parametreyi kullanarak dairenin alanını bulmaktadır. Bu metodun yerel değişkenleri, `Daire` sınıfının değişkenleri ile aynı adları taşımaktadır. `AlanBul()` metodu hesapladığı daire alanının değerini `alan` adlı yerel değişkeninin adresine yazar; çünkü öncelik kendi yerel değişkenlerinindir. Sonuçta sınıf değişkenlerine program içinde hiçbir değer atanmamıştır.

`Main()` metodu önce `Daire` sınıfına ait `d` adlı bir nesne yaratıyor. Sonra `AlanBul()` metodunun değerini olan `19,6349375` sayısını konsola yazdırıyor. En sonunda `Yaz()` metodunu çağırıyor. `Yaz()` metodu `AlanBul()` metodunun yerel değişkenlerine erişemez; o ancak sınıfın değişkenlerine erişiyor ve onların öndeğerleri (default value) olan 0 değerlerini konsola yazıyor.

Peki ama biz yerel değişkeni değil, sınıf değişkenini kullanmak istiyorsak ne yapmalıyız? O zaman `this` anahtar sözcüğü yardıma koşacaktır. Aşağıdaki örnekler bunun nasıl olduğunu gösteriyor.

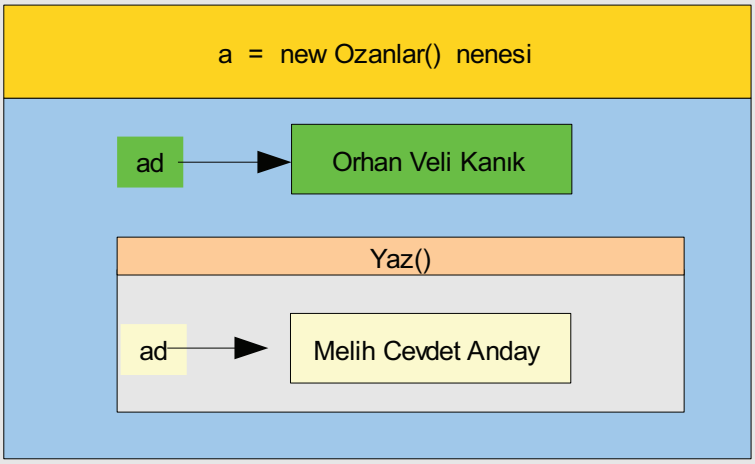
this anahtarının metot içinde kullanımı

Ozanlar.cs

```
using System;

    public class Ozanlar
    {
        public string ad  ;

        public void Yaz()
```

```
        {
            string ad = "Melih Cevdet Anday";
            this.ad   = "Orhan Veli Kanık";
            Console.WriteLine(ad) ;
            Console.WriteLine(this.ad );
        }
    }

class Uygulama
{
    public static void Main(string[] args)
    {
        Ozanlar a = new Ozanlar();
        a.Yaz();
    }
}
```

`Ozanlar` sınıfını çözümleyelim. Sınıfın içinde `ad` adlı string tipinden bir sınıf değişkeni tanımlıdır. O sınıfın bir öğesi olan `Yaz()` metodu içinde de aynı adla bir yerel değişken tanımlıdır. Derleyici her ikisine ayrı ayrı bellek adresleri ayırır.



Sınıf değişkeni ile yerel değişkene ayrı ayrı adresler tahsis edilir

```
string ad = "Melih Cevdet Anday";
```

deyimi yerel ad değişkenine "Melih Cevdet Anday" değerini atar.

```
this.ad   = "Orhan Veli Kanık";
```

deyimi ise sınıf öğesi olan ad değişkenine "Orhan Veli Kanık" değerini atar.

```
Console.WriteLine(ad) ;
```

deyimi yerel değişken değerini konsola yazar.

```
Console.WriteLine(this.ad );
```

deyimi ise sınıf değişkeninin değerini konsola yazar.

this Anahtarının Kurucu İçinde Kullanımı

Aşağıdaki programda `Ozan` sınıfı içinde tanımlanan public `Ozan()` kurucusundan sınıf değişkenine nasıl erişildiğini göstermektedir.

Ozan.cs

```
using System;
```

```
public class Ozan
{
    public string ad;

    public Ozan() //kurucu
    {
        this.ad = "Orhan Veli Kanık";
    }
}

class Uygulama
{
    public static void Main(string[] args)
    {
        Ozan obj = new Ozan();
        Console.WriteLine(obj.ad);
    }
}
```

`Ozan` sınıfı içinde tanımlanan public `Ozan()` kurucusu sınıfın ad değişkenine erişmek için `this.ad` ifadesini kullanmaktadır:

```
            this.ad    = "Orhan Veli Kanık";
```

Eğer bunlar yerine, örneğin

```
            string ad    = "Orhan Veli Kanık";
```

gibi yerel değişkenler koyarak programı

```
using System;

public class Ozan
{
    public string ad;

    public Ozan() //kurucu
    {
        string ad = "Orhan Veli Kanık";
    }
}

class Uygulama
{
    public static void Main(string[] args)
    {
        Ozan obj = new Ozan();
        Console.WriteLine(obj.ad);
    }
}
```

biçimine dönüştürürsek, program derlenir; yani sözdizimi hatası yoktur, ama çıktısı “” boş stringdir. Çünkü, `Ozan` sınıfının `ad` değişkenine değer atanmamıştır. Öndeğeri (default value) “” boş string 'dir; konsola öndeğeri gider.

Gezegen.cs

```
using System;

class Gezegen
{
    public int yarıÇap;
```

```
    public double yerÇekim;
    private string ad;
}

class Uygulama
{
    public static void Main()
    {
        Gezegen dünya = new Gezegen();
        dünya.yerÇekim = 9.81;
        dünya.yarıÇap = 6378;
        Console.WriteLine("Dünyanın yarıÇapı  = " + dünya.yarıÇap  );
        Console.WriteLine("Dünyanın yerÇekimi = " + dünya.yerÇekim );
    }
}
```