Bölüm 13

Yapı (struct)

Yapı Nedir?

C# dilinde yapılar farklı veri tiplerinden oluşan bir karma yapıdır. Sınıflara benzerler; onlar gibi tanımlanır, nesneleri onlar gibi yaratılır. Alanlar (field), metotlar, numaratörler (indexer) ve hatta başka yapıları öğe olarak içerebilirler. Ancak sınıf ile yapı arasında çok önemli bir ayrım vardır. Sınıfın nesneleri `heap` 'te tutulurken, yapı 'nın nesneleri `stack` 'ta tutulur. Böyle oluşu bazan iyidir, ama bazan da yapının öğe sayısına bağlı olarak elverişsiz olabilir. Neden öyle olduğunu birazdan anlayacağız.

C# dilinde değişkenlerin `değer (value)` ve `referans (reference)` tipler olmak üzere ikiye ayrıldığını; ana bellek içinde değer tiplerinin `stack`'ta, referans tiplerinin `heap`'te tutulduklarını biliyoruz.

`String` dışında bütün temel veri tipleri (built-in-types) `değer` tipidir; dolayısıyla `stack`'ta tutulurlar. Nesneler ve string `referans` tipidir, `heap`'te tutulurlar. `Stack`'ta tutulan öğelerin işi bitince kendiliğinden bellekten silinirler. Referans tiplerin işi bitince, çöp toplayıcı (garbage collector) onları toplayıp siler ve boşalan bellek bölgesini heap'e katar.

Java dilinde `struct` yoktur; programcı her şeyi sınıflarla yapar. C# dili sınıfa ek olarak `struct` yapısını ortaya koymuştur. Sınıftan yaratılan nesneler heap'te yer alıyordu. Bunun aksine, nesneleri stack'ta yer alacak bir alternatif oluşturmuştur.

Programcı stack'ta yer alan iki yapı kurabilir.

1. struct
2. enum

`enum` yapısını ileride göreceğiz. Bu bölümde `struct`'u inceleyeceğiz.

Yapı Bildirimi

Yapı bildirimi sınıf bildirimi gibi yapılır; class yerine struct yazılır:

```
<belirteçler> struct  <struct_adı>
{
//Struct 'un öğeleri
}
```

Örnek:

```
public struct DenekYapı
{
    public int a;
    public int b;
}
```

Bir yapı `private, public, internal` ya da `public` erişim belirteçlerinden birini alabilir.

New Operatörü ile Yapı Nesnesi Yaratma

Yapıya ait bir nesne yaratma aynen sınıflarda yapıldığı gibidir:

```
DenekYapı denek = new DenekYapı();
```

deyimi `stack` içinde `denek` adlı bir nesne yaratır. Yapının öğelerine erişim sınıflar için yapıldığı gibidir:

```
denek.a = 15;
denek.b = 20;
```

deyimleri yapı'ya ait denek nesnesinin öğelerine değer atamaktadır.
Sınıflarda olmadığı halde yapılarda var olan özeliklerden birisi de şudur:

```
DenekYapı d;
```

deyimi DenekYapı'ya ait d adlı bir nesne yaratır. Bu olgu temel veri tiplerinde yaptığımız

```
int n;
```

değişken bildirimine benziyor.
Aşağıdaki programı çözümleyelim.

Yapı01.cs

```
using System;

struct Kimlik
{
    public string ad;
    public int   yaş;


    static void Main(string[] args)
    {

        Kimlik y = new Kimlik();
        y.ad = "Dilek";
        Console.WriteLine(y.ad);
        Console.WriteLine(y.yaş);
    }
}
```

Çıktı

```
Dilek
0
```

`Kimlik` adlı struct içinde iki tane *alan (field)* tanımlıdır: `ad` ve `yaş`. Main() metodu içinde `new` opratörü ile Kimlik yapısının `y` adlı bir nesnesi yaratıldı. Bu nesne stack içindedir. `y` nesnesinin `ad` alanına "Dilek" adı atandı, ama `yaş` alanına bir değer atanmadığı için, öndeğer (default value) olarak 0 değerini tutuyor. Program derlenir, ama derleyici `yaş` alanına değer atanmadığı için öndeğer olarak 0 tuttuğu mesajını iletir. Zaten böyle olduğunu bu iki alanın değerlerini konsola yazdıran son iki deyimin çıktısından görüyoruz.

Bu programda başka bir özelik göze çarpıyor. Şimdiye kadar `Main()` metodunu daima bir sınıf (class) içinde tanımladık. Bu program, `Main()` metodunun bir yapı (struct) içinde de tanımlanabileceğini gösteriyor. Zaten, yapı ile sınıf bir çok konuda benzer işlevlere sahiptirler.

Sınıf İçinde Yapı Bildirimi

İstersek, yapıları sınıf içinde de tanımlayabiliriz. Aşağıdaki program onun yapılabilirliğini gösteriyor.

Yapı02.cs

```
using System;
class Yapılar
{
    struct Kimlik
    {
        public string ad;
        public int yaş;
    }

    static void Main(string[] args)
    {
        Kimlik y = new Kimlik();
        y.yaş = 19;
        Console.WriteLine(y.ad);
        Console.WriteLine(y.yaş);
    }
}
```

Çıktı

```
19
```

`Yapılar` adlı sınıfın içine `Kimlik` adlı yapı (struct) ile `Main()` metodu yerleşmiştir. Önceki programdan farklı olarak bu kez `ad` alanına değer atanmamış, `yaş` alanına 19 değeri atanmıştır. Çıktıdan görüldüğü gibi, konsola, string tipi olan `ad` değişkeninin öndeğeri olan "" boş string yazılmıştır.

Uyarı

Bu programda yapının alanlarına verilen public nitelemesini kaldırırsanız, derleyici onlara erişemediğini belirten şu iletiyi gönderecektir:

```
Error  1   'Yapılar.Kimlik.yaş' is inaccessible due to its protection level ...
```

New Operatörü Kullanmadan Yapı Nesnesi Yaratma

Şimdi new operatörünü kullanmadan, temel veri tiplerinde yaptığımız gibi yapı tipinden değişken tanımlayalım. Bunun için yukarıdaki programı aşağıdaki gibi değiştirelim.

Yapı03.cs

```
using System;
class Yapılar
{
    struct Kimlik
    {
        public string ad;
        public int yaş;
    }

    static void Main(string[] args)
    {
        Kimlik k;
        k.ad = "Yasemin";
        Console.WriteLine(k.ad);
        Console.WriteLine(k.yaş);
    }
}
```

Programı derlemek istersek şu hata mesajını alırız:

```
Error  1  Use of possibly unassigned field 'yaş' ...
```

Bu mesajdan anlıyoruz ki, yapıyı temel veri tipi gibi kullanarak değişken tanımlarsak, onun bütün öğelerine değer atamadan değişkeni kullanamayız. Gerçekten, ikinci değişkene de değer atarsak, programın derlendiğini görebiliriz. Bunun için yukarıdaki programda son satırdan önce

```
k.yaş = 21;
```

deyimini eklemeniz yetecektir.

Sınıf değişkenlerine bildirim anında ilk değer atanabiliyordu. Ama yapı içinde bildirimi yapılan alanlara, ilk değer atanamaz. Bu gerçeği görmek için, ilk programımızda değişkenlere değer atayıp derlemeyi deneyelim.

Yapı04.cs

```
using System;

struct Kimlik
{
    public string ad = "Zeki Alasya";
    public int yaş = 70;


    static void Main(string[] args)
    {
        Kimlik y = new Kimlik();
        Console.WriteLine(y.ad);
        Console.WriteLine(y.yaş);
    }
}
```

Derleyicimiz şu hata iletisini gönderecektir.

```
Error  1   'Kimlik.ad': cannot have instance field initializers in structs ...
```

Burada karşılaştığımız engeli aşmanın bir yolu var. Daha önce kullandığımız static nitelemesini kullanmak. Eğer yapının değişkenlerine (alan, field) `static` nitelemesini verirsek, engel ortadan kalkar. Tabii, bu durumda `new` operatörü ile yapıya ait bir nesne yaratmaya gerek kalmaz.

Yapı05.cs

```
using System;

struct Kimlik
{
    static string ad = "Zeki Alasya";
    static int yaş = 70;


    static void Main(string[] args)
    {
        Console.WriteLine(Kimlik.ad);
        Console.WriteLine(Kimlik.yaş);
    }
}
```

Bu programda `Kimlik` yapısının `static` değişkenlerine erişmek için, bir nesne adı değil, yapının adının kullanıldığına dikkat ediniz. Aynı olgunun sınıflar için de olduğunu görmüştük.

İlk programımızda aynı yapı içinde `Main()` metodunun bildirimini yaptık. İkinci programda hem yapıyı hem `Main()` metodunu bir sınıfın içine koyduk. Şimdi `Main()` metodunu yapı dışındaki bir sınıfın içine alıp sonucu görelim.

Yapı06.cs

```
using System;
struct Hesap
{
    public int a;
    public int b;
}
class Uygulama
{
    public static void Main()
    {
        Hesap yy = new Hesap();
        yy.a = 15;
        yy.b = 25;
        int toplam = yy.a + yy.b;
        Console.WriteLine("Toplam = {0}", toplam);
    }
}
```

Bu programda `Hesap` adlı yapı ile `Uygulama` adlı sınıf birbirinin dışındadırlar. `Uygulama` sınıfı içinde tanımlanan `Main()` metodu `Hesap` yapısı içindeki alanlara erişebilmektedir.

Yukarıda söylediğimiz gibi, `Hesap` yapısının alanlarından `public` nitelemesini kaldırırsak, `Main()` metodu onlara erişemez. Genel olarak, sınıflarda olduğu gibi, yapılardaki öğelere, yapı içinden engelsiz erişilir. Ama yapı dışından erişilebilmesi için yeterli bir erişim belirtecine sahip olması gerekir. Dışarıdan erişilmesini istemediğimiz değişkenleri bu izni vermeyerek onları gizlemiş (kapsüllemiş) oluruz.

İç-içe Yapılar

İç-içe yapılar kurulabilir. Aşağıdaki programda en dıştaki `Yapılar` adlı yapının içinde `Kimlik` ve `Uygulama` adlı yapılar yuvalanmıştır. `Main()` metodu `Uygulama` adlı yapının içindedir.

Yapı07.cs

```
using System;
struct Yapılar
{
    struct Kimlik
    {
        public string ad;
        public int yaş;
    }

    struct Uygulama
    {
        static void Main(string[] args)
        {
            Kimlik k;
            k.ad = "Pınar";
            k.yaş = 20;
            Console.WriteLine(k.ad);
            Console.WriteLine(k.yaş);
        }
    }
}
```

Yapılar İçinde Metotlar

Önceki programlarda yapı içinde `Main()` metodunun çalıştığını gördük. Aşağıdaki programda yapı içinde iki alan ve üç metot tanımlıdır. Yapıya ait nesnede alanlara değer atanmakta ve sonra atanan değerlerle işlem yapılmaktadır.

Yapı08.cs

```
using System;
struct Yapılar
{
    int x ;
    int y ;

    public void Ver(int i, int j)
    {
        x = i;
        y = j;
    }
```

```
    public void ToplamYaz()
    {
        int toplam = x + y;
        Console.WriteLine("Toplam = {0}", toplam);
    }

    public static void Main()
    {
        Yapılar aaa = new Yapılar();
        aaa.Ver(15, 25);
        aaa.ToplamYaz();
    }
}
```

Yapının static Değişkenleri

Sınıflarda dinamik değişkenlerin sınıfa ait her nesne içinde ayrı ayrı yaratıldığını biliyoruz. Bu demektir ki, sınıfa ait dinamik bir değişkenin herbir nesne içinde ayrı bir adresi vardır. Bir nesne içinde değişkene atanan değer öteki nesnelerin içindeki değerleri etkilemez. Ancak, statik değişkenlerde durum farklıdır. Sınıfa ait statik bir değişken nesneler içinde yaratılamaz; onun ana bellekte bir tek adresi olur. Bu adres sınıfa ait nesnelerden bağımsızdır. Dolayısıyla, herhangi bir anda statik değişkenin bir tek değeri vardır, o da en son atanan değerdir.

Bu olgu yapılar için de aynen geçerlidir. `static` nitelemeli değişkenler yapıya ait nesneler içinde yaratılmaz. Yapıya ait kaç nesne yaratılırsa yaratılsın, statik değişkenin bir tek adresi ve dolayısıyla herhangi bir anda bir tek değeri vardır, o da en son atanan değerdir. Aşağıdaki program bu gerçeği göstermektedir.

Yapı09.cs

```
using System;
struct Yapılar
{
    static int x = 250;
    static int y = 500;

    public void Ver(int i, int j)
    {
        x = i;
        y = j;
    }

    public void Ver(int i)
    {
        x = i;
        y = i;
    }

    public static void Main()
    {
        Console.WriteLine("x = {0} , y = {1} " ,x,y);
            Yapılar aaa = new Yapılar();
            aaa.Ver(10, 20);
            Console.WriteLine("x = {0} , y = {1} " ,x,y);
        Yapılar bbb = new Yapılar();
```

```
            bbb.Ver(1000);
            Console.WriteLine("x = {0} , y = {1} " ,x,y);


        }
}
```

Çıktı

```
x = 250 , y = 500
x = 10 , y = 20
x = 1000 , y = 1000
```

Programı çözümleyelim.

Yapının `x` ve `y` adlı iki `alanı (field)` var. Bu alanlar `static` nitelemeli olduğu için, bildirim anında, sırasıyla 250 ve 500 ilk değerleri atanabilmiştir. `Main()` metodunun ilk deyimi bu değerleri konsola yazdırıyor: x = 250 , y = 500.

Sonra `Main()` içindeki

```
Yapılar aaa = new Yapılar();
aaa.Ver(10, 20);
Console.WriteLine("x = {0} , y = {1} " ,x,y);
```

deyimleri çalışınca `aaa` adlı bir nesne yaratılıyor. Bu nesne içindeki `Ver` metodu `(10,20)` parametreleriyle çağrılıyor. Bu metot static değişkenlere x=10 ve y=20 atamalarını yapıyor. Bu yeni atamalar, statik değişkenlerin ilk değerlerinin yerine geçiyor. Böylece, son satır konsola bu yeni değerleri yazıyor: x = 10 , y = 20.

En son olarak `Main()` metodunun

```
Yapılar bbb = new Yapılar();
bbb.Ver(1000);
Console.WriteLine("x = {0} , y = {1} " ,x,y);
```

deyimleri çalışınca `bbb` adlı başka bir nesne yaratılıyor. Bu nesne içindeki `Ver` metodu `(1000)` parametresiyle çağrılıyor. Bu metot static değişkenlere x=1000 ve y=1000 atamalarını yapıyor. Bu yeni atamalar, statik değişkenlerin önceki değerlerinin yerine geçiyor. Böylece, son satır konsola bu yeni değerleri yazıyor: x = 1000 , y = 1000.

Bu programda dikkatimizi çeken başka bir şey daha var:

```
void Ver(int i, int j)
void Ver(int i)
```

metotları aynı ad ve aynı değer kümesine (void) sahiptirler. Böyle metotlara aşkın (overloaded) metotlar diyorduk. Bu program bize gösteriyor ki, yapılar içinde de aşkın metotlar tanımlanabilir.

Yapılar ve Kurucular

Sınıfların olduğu gibi yapıların da kurucuları tanımlanabilir. Parametresiz kurucular *genkurucu*'durlar (default constructor), onları ayrıca tanımlamaya gerek yoktur. Parametreli kurucular *aşkın* (overloaded) olabilirler. Aşağıdaki program bu gerçeği sergiler.

Yapı10.cs

```
using System;
struct Yapılar
{
```

```
    int x;
    int y;

    public Yapılar(int i, int j)
    { x = i; y = j; }

    public Yapılar(int i)
    { x = y = i; }

    public void AlanYaz()
    { Console.WriteLine("x = {0} , y = {1}", x, y); }
}
struct Uygulama
{
    public static void Main()
    {
        Yapılar aaa = new Yapılar(100, 200);
        Yapılar bbb = new Yapılar(300);
        aaa.AlanYaz();
        bbb.AlanYaz();
    }
}
```

Çıktı

```
x = 100 , y = 200
x = 300 , y = 300
```

Yapının içinde iki tane dinamik değişken ile iki tane aşkın kurucu tanımlanmıştır.

`Main()` metodu iki parametreli olan `Yapılar(100, 200)` kurucusu ile `aaa` adlı nesneyi kuruyor ve parametre değerlerini, sırasıyla `x` ve `y` dinamik değişkenlerine atıyor. `aaa.AlanYaz()` metodu konsola `x = 100 , y = 200` yazıyor.

Sonra `Main()` metodu tek parametreli olan `Yapılar(300)` kurucusu ile `bbb` adlı nesneyi kuruyor ve parametre değerini hem `x` hem `y` dinamik değişkenlerine atıyor. `bbb.AlanYaz()` metodu konsola `x = 100 , y = 200` yazıyor.

Struct ve Kalıtım

`struct` yapısı `System.ValueType` 'dan, `sınıflar` ise `System.Object`'den türerler. `struct` başka bir sınıf veya struct'tan türeyemez, türetilemez. Bunun yerine struct'un oluşturduğu (implemet) bir kaç önemli arayüzü vardır.

`struct`'u arayüz olarak kullanmak istersek, istemsiz (implicit) olarak kutulanır (boxed), çünkü arayüzler yalnızca referans tipleriyle iş yapabilirler. Örneğin,

Yapı11.cs

```
using System;
interface Iarayüz
{
    void birMetot();
}
struct Karma : Iarayüz
{
```

```
    public void birMetot()
    {
        Console.WriteLine("Struct Metodu çağrıldı");
    }
}
class Uygulama
{
    public static void Main()
    {
        Karma krm = new Karma();
        krm.birMetot();
    }
}
```

programında

```
Karma krm = new Karma();
```

deyimi Karma'nın bir nesnesini yaratır ve kutular (boxed). Arayüzün bütün metotları kutulanan bu nesne üzerinde işlevlerini yaparlar.

`class` ve `struct` yapısını bildiğimize göre, ne zaman `struct` kullanmak avantajlıdır, ne zaman değildir? sorusuna yanıt verebiliriz.

Ne zaman struct kullanmalı?

- Yaratılacak veri tipinin temel veri tipleri gibi davranması isteniyorsa.
- Çok sayıda nesne yaratılıp, hemen işi bitenler bellekten silinecekse. Örneğin, bir döngü içinde böyle bir durumla karşılaşılabilir.
- Yaratılan nesneler çok kullanılmıyor ve bellekte kalmaları gerekmiyorsa.
- Onu başka tiplerden türetmiyor ve ondan başka tipler türetilmiyorsa.
- Tutulan verinin başkalarına değer olarak aktarılabilmesi (pass by value) isteniyorsa.

Ne zaman struct kullanılmamalı?

- Struct'un öğelerinin bellekte işgal edecekleri bölge büyükse. (Microsoft bu büyüklüğü 16 byte ile sınırlandırmayı tavsiye ediyor).
- Yaratılan nesneler bir loleksiyona dahil ediliyor ve orada değiştiriliyor ya da onlarla döngü yapılıyorsa. Her işlemde boxing/unboxing yapılacağı için performans düşecektir. (Boxing/unboxing konusu ileride anlatılacaktır.

Sonuç

C# programcıya kendi veri tipini yaratmak için `struct` yapısı ile yeni bir araç sunmuştur. Her aracın iyi kullanılabileceği ve kullanılamayacağı yerler vardır. `Sınıf`'ın ve `struct`'un niteliklerini bilen programcı, yaratacağı veri tipini ne zaman sınıf ile ne zaman struct ile temsil edeceğine doğru karar vermelidir. Bazan birisi ötekinden daha etkin olabilir.