Bu dokümanda, C# programlama dilinde yer alan struct (yapı) veri tiplerini detaylı bir şekilde inceliyoruz. Struct'ların ne olduğu, nasıl tanımlandığı, bellekte nasıl çalıştığı gibi temel konular sade ve anlaşılır örneklerle anlatılmıştır. Bu doküman, nesne yönelimli programlama (OOP) konularına geçmeden önce sizleri bu yapılarla tanıştırmak ve ileride göreceğiniz daha karmaşık kavramlara zemin hazırlamak amacıyla hazırlanmıştır.

STRUCT'LAR

C#'ta Struct'lara giriş

Oğuzhan Karagüzel

İçindekiler

GİRİŞ

Bu dokümanda struct konusunu işlerken bazı terimler size yabancı gelebilir. Bunun sebebi, henüz nesne yönelimli programlamaya (OOP) giriş yapmamış olmamızdır. Endişelenmeyin; bu kavramlar ilerleyen derslerde detaylıca ele alınacak.

Bu dokümanı hazırlarken önceliğim, şu anki bilgi seviyenize uygun şekilde struct konusunu açıklamak oldu. Bilmediğiniz kavramlara takılmak yerine, anlatılan konuyu genel hatlarıyla anlamaya odaklanın. İleride OOP konularını öğrendiğinizde, buradaki bilgiler daha da netleşecek ve yerli yerine oturacaktır.

Unutmayın, programlamada öğrenme süreci katman katmandır. Bu doküman da o katmanlardan sadece biri. Merak edin, araştırın ama asla bilmedikleriniz sizi yolunuzdan alıkoymasın.

Geçtiğimiz derslerde List, Dictionary, HashSet, Queue ve Stack gibi gelişmiş koleksiyon tiplerini inceledik. Bu tipler, verileri organize etme ve yönetme konusunda bize büyük esneklik sunuyor. Bugün ise C# dilinin temel yapı taşlarından biri olan **Struct (Yapı)** kavramına derinlemesine bir giriş yapacağız. Struct'lar, özellikle küçük ve basit veri gruplarını temsil etmek için kullanılan **değer tipleridir (value types)**.

Değer tipleri ve referans tiplerine oop konusuna girdiğimizde daha derin değineceğiz. Ancak değer tipinin ne olduğunu şimdi öğrenmeniz gerekiyor. Bundan dolayı bu dokümanda bu konuya sıklıkla değineceğim. Referans tiplerin ne olduğunu bilmediğinizin farkındayım. Ancak şimdilik bunları görmezden gelin ve değer tiplerini daha iyi anlamaya çalışın. Değer tiplerine daha önce değinmiştik. Gerekirse lütfen önceki dokümanlara göz atın.

STRUCT NEDİR?

Struct'lar, mantıksal olarak birbiriyle ilişkili olan farklı veri tiplerindeki değişkenleri tek bir çatı altında toplamamızı sağlayan yapılardır. Bir nevi kendi özel veri tipimizi oluşturmak gibi düşünebilirsiniz. Örneğin, bir noktanın X ve Y koordinatlarını, bir öğrencinin numarasını ve adını veya bir kitabın ISBN'sini ve başlığını bir arada tutmak için struct'ları kullanabiliriz.

Önemli Not: Struct'lar **değer tipleridir**. Bu, C#'ta şimdiye kadar gördüğünüz ve ileride öğreneceğiniz class'lardan (referans tipleri) en temel farklarından biridir.

- **Değer Tipleri (Value Types):** Değişken doğrudan veriyi tutar. Bir değer tipi başka bir değişkene atandığında, verinin bir kopyası oluşturulur. int, double, bool, char gibi temel tipler ve struct'lar değer tipleridir.
- **Referans Tipleri (Reference Types):** Değişken, verinin bellekteki adresini (referansını) tutar. Bir referans tipi başka bir değişkene atandığında, sadece referans kopyalanır, veri kopyalanmaz. Her iki değişken de bellekteki aynı veriyi işaret eder. string (özel bir durum olsa da), object, diziler (arrays) ve class'lar referans tipleridir. Bu konuya ileride değineceğiz.

NEDEN STRUCT KULLANIRIZ?

1. **Performans:** Küçük ve basit veri kümeleri için class'lara göre daha performanslı olabilirler. Çünkü değer tipleri genellikle **stack** adı verilen bellek bölgesinde saklanır, bu da erişim ve yönetim açısından daha hızlıdır. (Detaylarına gireceğiz.)

2. **Bellek Yönetimi:** Stack'te saklandıkları için, Garbage Collector (Çöp Toplayıcı) üzerinde daha az yük oluştururlar. (Heap'te saklanan referans tipleri için GC daha fazla çalışır.)
3. **Basit Veri Grupları:** Birkaç alandan (field) oluşan, karmaşık davranışları olmayan, sadece veri taşımak amacıyla kullanılan yapılar için idealdirler.
4. **Değişmezlik (Immutability) İsteği:** Struct'lar genellikle değişmez (immutable) olarak tasarlanmaya daha yatkındır. Yani bir kere oluşturulduktan sonra değerlerinin değişmemesi hedeflenir. Bu, programın takibini kolaylaştırır.

STRUCT TANIMLAMA

Bir struct tanımlamak için struct anahtar kelimesini kullanırız. İçerisinde ise bu yapıyı oluşturan değişkenleri, yani **alanları (fields)** tanımlarız.

```
// Bir struct tanımlaması
struct Nokta
{
  // Alanlar (Fields)
  public int X; // public erişim belirtecini kullanıyoruz ki dışarıdan erişilebilsin
  public int Y;
}

struct Ogrenci
{
  public int OgrenciNo;
  public string Ad; // string bir referans tipidir, ancak struct içinde kullanılabilir
  public string Soyad;
}
```

Yukarıdaki örneklerde:

- Nokta adında bir struct tanımladık. Bu struct, bir noktanın 2D uzaydaki konumunu temsil etmek için X ve Y adında iki int tipinde alan içeriyor.
- Ogrenci adında bir struct tanımladık. Bu struct, bir öğrencinin numarasını, adını ve soyadını tutmak için OgrenciNo (int), Ad (string) ve Soyad (string) alanlarını içeriyor.

Erişim Belirteçleri: Şimdilik alanlarımızı public olarak tanımlıyoruz. Bu, struct'ın dışından bu alanlara doğrudan erişebileceğimiz anlamına gelir. Erişim belirteçlerini (public, private, internal vb.) daha sonra detaylı işleyeceğiz.

STRUCT KULLANIMI (INSTANCE OLUŞTURMA VE ALANLARA ERİŞME)

Bir struct'ı tanımladıktan sonra, onu kullanmak için bir örneğini (instance) oluşturmamız gerekir. Bu, new anahtar kelimesiyle veya doğrudan atama yaparak yapılabilir.

```
// Program.cs dosyasında Main metodu içinde
class Program
{
    static void Main(string[] args)
    {
        // 1. Yöntem: new anahtar kelimesi ile instance oluşturma
        Nokta nokta1 = new Nokta();
        nokta1.X = 10;
        nokta1.Y = 20;

        Console.WriteLine($"Nokta1: X={nokta1.X}, Y={nokta1.Y}");

        // 2. Yöntem: Alanlara değer atamadan önce new kullanmak zorunlu değil (C# 10 ve sonrası için)
        // Ancak tüm alanlara değer atanmadan kullanılamaz.
        // new Nokta() yapısı, alanlara varsayılan değerlerini (int için 0, bool için false vb.) atar.
        // Eğer new kullanmazsanız, tüm alanlara siz değer atamak zorundasınız.
        Nokta nokta2;
        nokta2.X = 5;
        nokta2.Y = 15; // Eğer Y'ye değer atamasaydık ve kullanmaya çalışsaydık derleme hatası alırdık.

        Console.WriteLine($"Nokta2: X={nokta2.X}, Y={nokta2.Y}");

        // Ogrenci struct'ı için örnek
        Ogrenci ogr1 = new Ogrenci(); // Alanlar varsayılan değerleri alır (int için 0, string için null)
        ogr1.OgrenciNo = 101;
        ogr1.Ad = "Ali";
        ogr1.Soyad = "Veli";
```

```
        Console.WriteLine($"Öğrenci: No={ogr1.OgrenciNo}, Ad Soyad={ogr1.Ad} {ogr1.Soyad}");

        Ogrenci ogr2;
        ogr2.OgrenciNo = 102;
        ogr2.Ad = "Ayşe";
        ogr2.Soyad = "Yılmaz"; // Tüm alanlara ilk değer ataması yapıldı.
        Console.WriteLine($"Öğrenci: No={ogr2.OgrenciNo}, Ad Soyad={ogr2.Ad} {ogr2.Soyad}");

        Console.ReadKey();
    }
}

// Struct tanımlamaları buraya veya ayrı bir dosyaya yazılabilir (Namespace konusunda göreceğiz)
struct Nokta
{
    public int X;
    public int Y;
}

struct Ogrenci
{
    public int OgrenciNo;
    public string Ad;
    public string Soyad;
}
```


Kod Açıklaması:

- Nokta nokta1 = new Nokta();: Nokta tipinde nokta1 adında bir değişken oluşturuyoruz ve new Nokta() ifadesiyle bellekte bu struct için yer ayrılmasını sağlıyoruz. new ile oluşturduğumuzda, X ve Y alanları varsayılan değerleri olan 0 (sıfır) ile başlar.
- nokta1.X = 10;: nokta1 örneğinin X alanına 10 değerini atıyoruz.
- Nokta nokta2;: Burada new kullanmadık. Bu durumda nokta2 için bellekte yer ayrılır, ancak alanlarına varsayılan değerler atanmaz. Tüm alanlarına siz bir değer atamadan nokta2'yi

(örneğin Console.WriteLine(nokta2.X)) kullanmaya çalışırsanız derleme hatası alırsınız: "Use of unassigned local variable 'nokta2'".

- ogr1.Ad = "Ali";: Ogrenci struct'ındaki Ad alanı string tipindedir. String'ler referans tipi olmasına rağmen struct'lar içinde sorunsuzca kullanılabilirler.

STRUCT'LAR VE BELLEK: STACK

Struct'ların en önemli özelliklerinden biri, genellikle **stack (yığın)** adı verilen bir bellek bölgesinde saklanmalarıdır.

- **Stack Nedir?**
  Stack, programınızdaki yerel değişkenlerin ve metot çağrılarının yönetildiği, son giren ilk çıkar (LIFO - Last In, First Out) prensibiyle çalışan bir bellek alanıdır. Bir metot çağrıldığında, o metodun yerel değişkenleri (struct'lar dahil) ve parametreleri için stack'te bir "çerçeve (frame)" oluşturulur. Metot sonlandığında, bu çerçeve stack'ten otomatik olarak kaldırılır.
    - **Hızlıdır:** Stack'e veri eklemek ve çıkarmak çok hızlıdır çünkü sadece bir işaretçinin (stack pointer) yeri değiştirilir.
    - **Sınırlı Boyut:** Stack'in boyutu genellikle heap'e göre daha küçüktür. Çok büyük struct'lar veya çok derin metot çağrıları (recursive) stack taşmasına (StackOverflowException) neden olabilir.
- **Heap Nedir?**
  Heap (öbek), dinamik olarak bellek ayrılan (genellikle new anahtar kelimesiyle class örnekleri oluşturulduğunda) daha büyük ve daha esnek bir bellek alanıdır. Referans tipleri (class'lar, diziler) heap'te saklanır. Heap'teki nesnelerin ne zaman silineceğine Garbage Collector (GC - Çöp Toplayıcı) karar verir. Bu, heap yönetimini stack'e göre daha karmaşık ve biraz daha yavaş hale getirir.

METOTTAN ÇIKINCA STRUCT DEĞERİNE NE OLUR?

Bir metot içinde tanımlanan bir struct değişkeni, o metot çalıştığı sürece stack'te yaşar. Metot sonlandığında, o metoda ait stack çerçevesi (içindeki tüm yerel değişkenlerle birlikte) stack'ten kaldırılır. Yani, struct değişkeninin kapladığı alan serbest bırakılır ve artık erişilemez hale gelir.

Eğer struct bir metodun geri dönüş değeri ise, metot sonlandığında struct'ın bir **kopyası** çağrı yapan metoda geri döndürülür. Orijinal struct (metot içindeki) yine stack'ten silinir.

```
    struct BasitStruct
{
   public int Deger;
}


class Program
{
   static BasitStruct StructOlustur(int val)
```

```
    {
        BasitStruct s = new BasitStruct();
        s.Deger = val;
        Console.WriteLine($"StructOlustur içinde: s.Deger = {s.Deger} (Adres benzeri bir şey: bu bilgi doğrudan alınamaz ama stack'te olduğunu biliyoruz)");
        // Bu metot bittiğinde 's' stack'ten silinecek, AMA bir kopyası geri dönecek.
        return s;
    }

    static void StructKullan(BasitStruct param)
    {
        // 'param' buraya bir KOPYA olarak gelir.
        param.Deger = 100; // Bu değişiklik sadece bu metot içindeki kopyayı etkiler.
        Console.WriteLine($"StructKullan içinde: param.Deger = {param.Deger}");
        // Bu metot bittiğinde 'param' (kopya) stack'ten silinir.
    }

    static void Main(string[] args)
    {
        BasitStruct anaStruct = StructOlustur(50);
        Console.WriteLine($"Main içinde StructOlustur'dan sonra: anaStruct.Deger = {anaStruct.Deger}");

        StructKullan(anaStruct); // anaStruct'ın bir KOPYASI StructKullan metoduna gider.
        Console.WriteLine($"Main içinde StructKullan'dan sonra: anaStruct.Deger = {anaStruct.Deger}");
// Değer hala 50 kalır!

        Console.ReadKey();
    }
}
```

Kod Açıklaması ve Çıktı Analizi:

StructOlustur içinde: s.Deger = 50

Main içinde StructOlustur'dan sonra: anaStruct.Deger = 50

StructKullan içinde: param.Deger = 100

Main içinde StructKullan'dan sonra: anaStruct.Deger = 50

StructOlustur(50) çağrıldığında:

- s adında bir BasitStruct stack'te oluşturulur, s.Deger 50 olur.
- s geri döndürülürken, s'in bir kopyası Main metodundaki anaStruct değişkenine atanır. StructOlustur metodu bittiğinde, orijinal s stack'ten silinir.

2. StructKullan(anaStruct) çağrıldığında:
   - anaStruct'ın bir kopyası param olarak StructKullan metoduna geçer. param da stack'te ayrı bir yer tutar.
   - param.Deger = 100; yapıldığında, sadece StructKullan metodundaki param kopyasının Deger alanı değişir. Main metodundaki anaStruct bundan etkilenmez.
   - StructKullan metodu bittiğinde, param stack'ten silinir.
3. Main metoduna dönüldüğünde anaStruct.Deger hala 50'dir, çünkü StructKullan metoduna yapılan değişiklikler sadece kopyası üzerinde geçerliydi. Bu, değer tiplerinin (value types) en önemli özelliğidir: **"pass-by-value" (değer ile geçirme)**.

UNSAFE KOD İLE STRUCT BOYUTUNU GÖSTERME

C#'ta normalde bellek adresleriyle veya bir tipin bellekte kapladığı kesin byte sayısıyla doğrudan ilgilenmeyiz. Ancak, struct'ların stack'te nasıl yer kapladığını anlamak için unsafe kod ve sizeof operatörünü kullanabiliriz.

UYARI: unsafe kod, C#'ın tip güvenliği ve bellek yönetimi mekanizmalarını devre dışı bırakır. Hatalı kullanıldığında programın çökmesine veya beklenmedik davranışlara yol açabilir. Sadece ne yaptığınızı çok iyi bildiğiniz durumlarda ve genellikle düşük seviyeli interoperability (diğer dillerle etkileşim) veya performansın kritik olduğu özel senaryolarda kullanılır. Bu örnek sadece **eğitim amaçlıdır**.

Bir C# projesinde unsafe kod kullanmak için proje ayarlarında "Allow unsafe code" (Güvenli olmayan koda izin ver) seçeneğini işaretlemeniz gerekir:

- Visual Studio'da: Solution Explorer -> Proje adına sağ tıkla -> Properties -> Build -> "Allow unsafe code" kutucuğunu işaretle.

sizeof operatörü, bir değer tipinin (veya unsafe blok içinde yönetilmeyen bir tipin) bellekte kaç byte yer kapladığını döndürür.

```
// unsafe kod kullanmak için proje ayarlarından "Allow unsafe code" işaretlenmeli
// ve metot veya blok unsafe olarak işaretlenmeli.

struct NoktaBasit
{
    public int X; // int genellikle 4 byte
    public int Y; // int genellikle 4 byte
```

```
}

struct NoktaKarma
{
    public int X;     // 4 byte
    public byte B;    // 1 byte
    public long L;    // 8 byte
    public short S;   // 2 byte
    // Padding (hizalama) nedeniyle beklediğimizden farklı boyutlar görebiliriz.
}

struct BosStruct { } // İçinde field olmayan struct

class Program
{
    unsafe static void Main(string[] args) // Main metodunu unsafe yaptık
    {
        Console.WriteLine("Struct Boyutları (byte cinsinden):");

        Console.WriteLine($"Boyut (NoktaBasit): {sizeof(NoktaBasit)} byte");
        // Beklenen: int (4 byte) + int (4 byte) = 8 byte

        NoktaBasit nb = new NoktaBasit();
        Console.WriteLine($"Boyut (nb değişkeni üzerinden): {sizeof(NoktaBasit)} byte"); // Aynı sonucu
verir

        Console.WriteLine($"Boyut (int): {sizeof(int)} byte");
        Console.WriteLine($"Boyut (byte): {sizeof(byte)} byte");
        Console.WriteLine($"Boyut (long): {sizeof(long)} byte");
        Console.WriteLine($"Boyut (double): {sizeof(double)} byte");
        Console.WriteLine($"Boyut (char): {sizeof(char)} byte"); // char 2 byte (UTF-16)
```

```
        Console.WriteLine($"Boyut (bool): {sizeof(bool)} byte"); // bool 1 byte


        Console.WriteLine($"Boyut (NoktaKarma): {sizeof(NoktaKarma)} byte");
        // Toplam normalde 4+1+8+2 = 15 byte. Ancak CPU'lar veriye daha hızlı erişmek için
        // verileri belirli sınırlara (genellikle 4 veya 8 byte'ın katları) hizalar.
        // Bu "padding" nedeniyle boyut 15 değil, muhtemelen 16 veya 24 byte olabilir.
        // (Sistem mimarisine göre değişebilir)


        Console.WriteLine($"Boyut (BosStruct): {sizeof(BosStruct)} byte");
        // Boş bir struct bile bellekte en az 1 byte yer kaplar.


        Console.ReadKey();
    }
}
```

Kod Açıklaması:

- unsafe static void Main...: Main metodunu unsafe olarak işaretledik, böylece içinde sizeof'u yönetilen tiplerle (struct'larımız gibi) kullanabiliriz.
- sizeof(NoktaBasit): NoktaBasit struct'ının bellekte kaç byte yer kapladığını verir. İki int alanı olduğu için (genellikle 1 int = 4 byte), toplam 8 byte bekleriz.
- sizeof(NoktaKarma): Bu struct farklı boyutlarda alanlar içerir. Teorik toplam 4 (int) + 1 (byte) + 8 (long) + 2 (short) = 15 byte. Ancak, işlemciler performansı artırmak için verileri belirli bellek sınırlarına (alignment/hizalama) göre yerleştirir. Bu, alanlar arasına "padding" (boşluk) eklenmesine neden olabilir. Bu yüzden sizeof(NoktaKarma) sonucu genellikle 15'ten büyük (örneğin 16, 24 gibi 4'ün veya 8'in katı) bir değer olacaktır. Bu, sistem mimarisine (32-bit, 64-bit) ve .NET çalışma zamanına göre değişebilir.
- sizeof(BosStruct): İçinde hiçbir alan olmayan bir struct bile bellekte 1 byte yer kaplar. Bu, farklı BosStruct örneklerinin bellekte farklı adreslere sahip olmasını sağlamak içindir.

SIZEOF KULLANIMININ ÖZETİ:

sizeof bize bir struct'ın veya temel bir veri tipinin stack'te ne kadar yer kapladığı hakkında bir fikir verir. Field ekledikçe veya field'ların tiplerini değiştirdikçe bu boyutun nasıl etkilendiğini görebiliriz. NoktaKarma örneğindeki gibi, alanların sırası bile padding nedeniyle toplam boyutu etkileyebilir (ama bu daha ileri bir konudur).

NEW ANAHTAR KELİMESİ VE CONSTRUCTOR (YAPICI) METOTLAR

Önceki dersimizde struct'ları tanımlamayı ve alanlarına (fields) değer atamayı gördük. Bu değer atama işlemini daha kontrollü ve merkezi bir şekilde yapmak için **constructor (yapıcı) metotları** kullanırız. Bir struct'ın örneğini (instance) oluştururken new anahtar kelimesi ile birlikte bu constructor'lar çağrılır.

CONSTRUCTOR (YAPICI) METOT NEDİR?

- **Tanım:** Constructor, bir class'ın veya struct'ın yeni bir örneği (nesnesi/instance'ı) oluşturulduğunda **otomatik olarak çağrılan** özel bir metottur.
- **Amacı:** Temel amacı, struct'ın veya class'ın alanlarına (fields) başlangıç değerlerini atamak ve nesneyi kullanıma hazır, tutarlı bir hale getirmektir. Bir nevi "ilk kurulum" veya "hazırlık" rutini gibidir.
- **Özellikleri:**
    - Constructor metodunun adı, ait olduğu **struct (veya class) ile aynı olmalıdır.**
    - Constructor'ların **geri dönüş tipi yoktur** (hatta void bile yazılmaz).
    - public, private gibi erişim belirteçleri alabilirler. Genellikle public olurlar ki dışarıdan örnek oluşturulabilsin.
    - Parametre alabilirler (parametre alan constructor'lar) veya parametresiz olabilirler (parametresiz constructor).

NEW ANAHTAR KELİMESİ NEDİR VE NE YAPAR?

new anahtar kelimesi C#'ta temel olarak iki ana amaç için kullanılır:

- **Nesne (Instance) Oluşturma:** Bir class veya struct tipinden yeni bir örnek (nesne) oluşturmak için kullanılır.
    - **Struct'lar İçin:** new kullandığınızda, derleyici struct için bellekte (genellikle stack'te) yer ayırır ve ardından belirttiğiniz constructor'ı çağırarak bu bellek bölgesindeki alanları başlatır.
    - **Class'lar İçin:** new kullandığınızda, derleyici nesne için bellekte (heap'te) yer ayırır, constructor'ı çağırır ve bu nesnenin heap'teki adresini (referansını) döndürür.
- **Tip Üyelerini Gizleme (Shadowing/Hiding):** Kalıtım konusunda göreceğimiz, bir alt class'ta üst class'taki aynı isimli bir metodu "gizlemek" için kullanılır (şimdilik bu konuya girmiyoruz).

Şu anki konumuzla ilgili olarak, new kelimesi **bir struct'ın constructor'ını çağırmak ve onun bir örneğini oluşturmak** için kullanılır.

STRUCT'LARDA CONSTRUCTOR KULLANIMI

a) Parametresiz (Varsayılan) Constructor (Default Constructor)

Her struct'ın (C# 10 öncesinde sizin tanımlayamadığınız, sistem tarafından sağlanan) **parametresiz bir constructor'ı** vardır.
new AnahtarKelimesiyleStructOlusturma() yaptığınızda bu constructor çağrılır.

- **Görevi:** Struct içindeki tüm alanları (fields) kendi tiplerinin varsayılan değerlerine ayarlar.
    - Sayısal tipler (int, double vb.) için: 0
    - bool için: false
    - char için: '\0' (null karakter)
    - Referans tipleri (string, class'lar) için: null (eğer struct içinde string bir field varsa, bu null olur.)
    - Diğer struct tipleri için: O struct'ın alanları da kendi varsayılan değerlerine ayarlanır.

```
 struct Nokta
{
   public int X;
   public int Y;
   public string Etiket;
}

class Program
{
   static void Main(string[] args)
   {
      // 'new Nokta()' çağrıldığında, Nokta struct'ının varsayılan
      // parametresiz constructor'ı çalışır.
      Nokta p1 = new Nokta();

      // p1.X otomatik olarak 0 olur
      // p1.Y otomatik olarak 0 olur
      // p1.Etiket otomatik olarak null olur
      Console.WriteLine($"P1: X={p1.X}, Y={p1.Y}, Etiket={(p1.Etiket == null ? "null" : p1.Etiket)}");
      // Çıktı: P1: X=0, Y=0, Etiket=null
```

```
        // Alanlara sonradan değer atayabiliriz:
        p1.X = 10;
        p1.Y = 20;
        p1.Etiket = "Başlangıç";
        Console.WriteLine($"P1 (güncel): X={p1.X}, Y={p1.Y}, Etiket={p1.Etiket}");
        // Çıktı: P1 (güncel): X=10, Y=20, Etiket=Başlangıç

        Console.ReadKey();
    }
}
```

ÖNEMLİ NOT (C# 10 ve Sonrası): C# 10 ile birlikte artık struct'lar için **kendi parametresiz constructor'ınızı tanımlayabilirsiniz.** Eğer kendi parametresiz constructor'ınızı tanımlarsanız, bu constructor içinde **tüm alanlara (fields) açıkça bir değer atamak zorundasınızdır.** Eğer tanımlamazsanız, yine sistemin sağladığı ve tüm alanları varsayılan değerlere ayarlayan constructor çalışır.

```
    // C# 10 ve üzeri için geçerli
struct Ogrenci
{
    public int No;
    public string Ad;

    // Kendi tanımladığımız parametresiz constructor
    public Ogrenci()
    {
        No = -1; // Tüm alanlara değer atamak zorundayız!
        Ad = "Bilinmiyor"; // Atamazsak derleme hatası alırız.
        Console.WriteLine("Ogrenci için özel parametresiz constructor çağrıldı.");
    }
}
```

```
// ... Main içinde ...
// Ogrenci ogr = new Ogrenci();
// Console.WriteLine($"No: {ogr.No}, Ad: {ogr.Ad}");
// Çıktı:
// Ogrenci için özel parametresiz constructor çağrıldı.
// No: -1, Ad: Bilinmiyor
```

b) Parametreli Constructor

Struct'ların örneklerini oluştururken alanlarına doğrudan başlangıç değerleri vermek isteyebiliriz. Bunun için parametre alan constructor'lar tanımlarız.

- Bir struct içinde birden fazla parametreli constructor olabilir (constructor overloading), yeter ki parametre imzaları (parametre sayısı veya tipleri) farklı olsun.
- **KURAL:** Eğer bir struct için parametreli bir constructor tanımlıyorsanız, o constructor'ın içinde **struct'ın TÜM alanlarına (fields) bir değer atamak zorundasınızdır.**

```
struct Kitap
{
    public string Baslik;
    public string Yazar;
    public int SayfaSayisi;
    public bool StoktaVar;

    // Parametreli Constructor
    // Bu constructor çağrıldığında tüm alanlara değer atanmalı
    public Kitap(string baslik, string yazar, int sayfaSayisi)
    {
        // this anahtar kelimesi, bu struct örneğinin kendisini ifade eder.
        // Parametre isimleri field isimleriyle aynıysa, karışıklığı önlemek için 'this' kullanılır.
        this.Baslik = baslik;
        this.Yazar = yazar;
        this.SayfaSayisi = sayfaSayisi;
        this.StoktaVar = true; // Bu alana da bir değer atamamız GEREKİR.
                               // Eğer 'StoktaVar'a atama yapmazsak derleme hatası alırız:
```

```
                    // "Field 'Kitap.StoktaVar' must be fully assigned before control is returned to the
caller"
        Console.WriteLine($"{this.Baslik} için parametreli constructor çağrıldı.");
    }

    // Başka bir parametreli constructor (Overload)
    public Kitap(string baslik)
    {
        this.Baslik = baslik;
        this.Yazar = "Bilinmiyor"; // Tüm alanlara değer ata
        this.SayfaSayisi = 0;      // Tüm alanlara değer ata
        this.StoktaVar = false;    // Tüm alanlara değer ata
        Console.WriteLine($"{this.Baslik} için SADECE BAŞLIK alan parametreli constructor çağrıldı.");
    }
}

class Program
{
    static void Main(string[] args)
    {
        // 'new' ile parametreli constructor'ı çağırıyoruz
        Kitap kitap1 = new Kitap("Sefiller", "Victor Hugo", 1500);
        Console.WriteLine($"Kitap1: {kitap1.Baslik}, Yazar: {kitap1.Yazar}, Sayfa: {kitap1.SayfaSayisi}, Stok:
{kitap1.StoktaVar}");

        Kitap kitap2 = new Kitap("Suç ve Ceza");
        Console.WriteLine($"Kitap2: {kitap2.Baslik}, Yazar: {kitap2.Yazar}, Sayfa: {kitap2.SayfaSayisi}, Stok:
{kitap2.StoktaVar}");

        // Varsayılan constructor hala kullanılabilir (eğer özel parametresiz constructor tanımlamadıysak)
        Kitap kitap3 = new Kitap(); // Alanlar varsayılan değerlerini alır (string=null, int=0, bool=false)
```

```
        Console.WriteLine($"Kitap3: Başlık='{(kitap3.Baslik == null ? "null" : kitap3.Baslik)}',
Yazar='{(kitap3.Yazar == null ? "null" : kitap3.Yazar)}', Sayfa={kitap3.SayfaSayisi},
Stok={kitap3.StoktaVar}");


        Console.ReadKey();
    }
}
```

Çıktı:

```
    Sefiller için parametreli constructor çağrıldı.
Kitap1: Sefiller, Yazar: Victor Hugo, Sayfa: 1500, Stok: True
Suç ve Ceza için SADECE BAŞLIK alan parametreli constructor çağrıldı.
Kitap2: Suç ve Ceza, Yazar: Bilinmiyor, Sayfa: 0, Stok: False
Kitap3: Başlık='null', Yazar='null', Sayfa=0, Stok=False
```

new Kullanmadan Struct Instance Oluşturma (ve Constructor Çağrılmaz!)

Daha önce de bahsettiğimiz gibi, struct'lar için new kullanmadan da değişken tanımlayabiliriz. Ancak bu durumda **hiçbir constructor çağrılmaz** ve struct'ın alanları başlangıçta **atanmamış (unassigned)** olur. Tüm alanlara siz manuel olarak değer atamadan o struct'ı (örneğin bir metodun parametresi olarak veya alanlarını okuyarak) kullanamazsınız.

```
    Nokta p2; // new kullanılmadı, constructor çağrılmaz
// Console.WriteLine(p2.X); // HATA! "Use of unassigned local variable 'p2'"

p2.X = 50;
p2.Y = 70;
p2.Etiket = "Orta Nokta"; // Tüm alanlara değer atandıktan sonra kullanılabilir.
Console.WriteLine($"P2: X={p2.X}, Y={p2.Y}, Etiket={p2.Etiket}");
```

Özet:

- **Constructor:** Bir struct'ın örneği oluşturulduğunda çalışan, alanları başlatan özel bir metottur. Adı struct adıyla aynıdır, geri dönüş tipi yoktur.
- **new Anahtar Kelimesi:** Bir struct'ın constructor'ını çağırmak ve bellekte bir örneğini (instance) oluşturmak için kullanılır.

- **Varsayılan Parametresiz Constructor:** new StructAdi() ile çağrılır. Siz özel bir parametresiz constructor yazmadıysanız (C# 10+), sistemin sağladığı tüm alanları varsayılan değerlerine ayarlar.
- **Parametreli Constructor:** new StructAdi(parametreler) ile çağrılır. İçinde struct'ın TÜM alanlarına değer atanmalıdır.
- new kullanmadan struct değişkeni tanımlanabilir, ancak bu durumda constructor çağrılmaz ve tüm alanlara manuel değer atanması gerekir. new kullanmak genellikle daha güvenli ve okunaklıdır çünkü alanların başlatılacağını garanti eder.

Constructor'lar, struct'larınızın her zaman geçerli bir durumda başlamasını sağlamak için çok önemlidir.

NAMESPACE KAVRAMI (İSİM ALANLARI)

Projeniz büyüdükçe, yazdığınız class, struct, interface, enum gibi tiplerin sayısı artacaktır. Farklı kütüphaneler veya projenin farklı modülleri aynı isimde tipler tanımlayabilir. Bu durum **isim çakışmalarına (naming conflicts)** yol açar. Namespace'ler bu sorunu çözmek için vardır.

NAMESPACE NEDİR?

Namespace, bir grup ilişkili tip (class, struct, interface, enum, delegate) için bir "kapsayıcı" veya "alan" tanımlar. Tıpkı dosya sisteminizdeki klasörler gibi, namespace'ler de kodunuzu organize etmenize ve aynı isme sahip tipleri birbirinden ayırmanıza olanak tanır.

Örneğin, hem sizin projenizde bir Logger class'ı olabilir, hem de kullandığınız harici bir kütüphanede Logger adında başka bir class olabilir. Namespace'ler olmadan, derleyici hangi Logger'ı kastettiğinizi anlayamazdı.

Neden Kullanılır?

1. **Organizasyon:** İlişkili tipleri bir arada gruplayarak kodun daha düzenli ve anlaşılır olmasını sağlar.
2. **İsim Çakışmalarını Önleme:** Farklı namespace'lerde aynı isimde tipler tanımlanabilir. Bir tipi kullanırken, NamespaceAdi.TipAdi şeklinde tam adını belirterek hangi namespace'deki tipi kastettiğimizi belirtiriz.
3. **Kodun Yeniden Kullanılabilirliği:** Kendi kütüphanelerinizi oluşturduğunuzda, namespace'ler sayesinde diğer projelerde bu tiplerin çakışma olmadan kullanılmasını sağlarsınız.

Namespace Tanımlama

Bir namespace tanımlamak için namespace anahtar kelimesini kullanırız.

```
namespace FirmaAdi.ProjeAdi.ModulAdi
{
    // Bu namespace içindeki tipler (class, struct, vs.)
    struct VeriPaketi
```

```
    {
        public int Id;
        public string Bilgi;
    }

    class Hesaplayici
    {
        // ...
    }
}
```

Namespace'ler iç içe de olabilir (FirmaAdi.ProjeAdi.ModulAdi örneğindeki gibi). Bu, daha granüler bir organizasyon sağlar.

BİR STRUCT'I BAŞKA BİR DOSYADA TANIMLAMA VE KULLANMA

Genellikle her struct (veya class) kendi .cs uzantılı dosyasında tanımlanır. Bu, projenin okunabilirliğini ve yönetilebilirliğini artırır.

Adım 1: Struct'ı Ayrı Bir Dosyada Oluşturma

1. Projenize yeni bir C# dosyası ekleyin. Örneğin, GeometrikSekiller.cs adında bir dosya.
2. Bu dosyanın içine Nokta struct'ımızı bir namespace ile birlikte tanımlayalım:

```
3.  // GeometrikSekiller.cs
4.
5.  namespace GeometriKutuphanesi // Kendi namespace'imizi tanımlıyoruz
6.  {
7.      public struct Nokta // struct'ı public yapıyoruz ki başka namespace'lerden erişilebilsin
8.      {
9.          public int X;
10.         public int Y;
11.
12.         // İleride buraya metotlar da ekleyebiliriz (constructor, ToString vs.)
13.         // Şimdilik sadece field'lar yeterli.
14.     }
15.
```

```
16.     public struct Dikdortgen
17.     {
18.         public Nokta SolUstKose;
19.         public int Genislik;
20.         public int Yukseklik;
21.     }
22. }
```

- namespace GeometriKutuphanesi: Nokta ve Dikdortgen struct'larını bu namespace içine aldık.
- public struct Nokta: Struct'ın public olması, GeometriKutuphanesi namespace'i dışından da erişilebilir olmasını sağlar. Eğer public yazmasaydık, varsayılan erişim belirteci internal olurdu, bu da sadece aynı assembly (proje) içinden erişilebilir demek olurdu.

ADIM 2: MAIN FONKSİYONUNDA KULLANMA VE USİNG DİREKTİFİ

Şimdi Program.cs dosyamızdaki Main metodunda bu Nokta struct'ını kullanalım.

```
// Program.cs


// Yöntem 1: Tam nitelikli isim (Fully Qualified Name) kullanmak
// Eğer using direktifi kullanmazsak, tipin tam adını namespace ile belirtmemiz gerekir.
// namespace GeometriKutuphanesi
// {
//     public struct Nokta {...}
// }


class Program_EskiYontem // İsim çakışması olmasın diye değiştirdim
{
    static void Main(string[] args)
    {
        GeometriKutuphanesi.Nokta p1 = new GeometriKutuphanesi.Nokta();
        p1.X = 10;
        p1.Y = 20;
        Console.WriteLine($"P1: ({p1.X}, {p1.Y})");
```

```
        GeometriKutuphanesi.Dikdortgen d1 = new GeometriKutuphanesi.Dikdortgen();
        d1.SolUstKose.X = 5;
        d1.SolUstKose.Y = 5;
        d1.Genislik = 100;
        d1.Yukseklik = 50;

        Console.WriteLine($"Dikdörtgen: Sol Üst=({d1.SolUstKose.X},{d1.SolUstKose.Y}),
Genişlik={d1.Genislik}, Yükseklik={d1.Yukseklik}");

        Console.ReadKey();
    }
}
```

Gördüğünüz gibi, her seferinde GeometriKutuphanesi.Nokta yazmak biraz yorucu olabilir. İşte burada using direktifi devreye girer.

USING DİREKTİFİ

using direktifi, bir namespace'i mevcut dosyanın kapsamına dahil etmenizi sağlar. Bu sayede, o namespace içindeki tipleri kullanırken namespace adını tekrar tekrar yazmak zorunda kalmazsınız. using direktifleri genellikle dosyanın en üstüne yazılır.

Şimdi Program.cs dosyamızı using direktifi ile güncelleyelim:

```
    // Program.cs

using System; // Console sınıfı için bu zaten genelde ekli olur
using GeometriKutuphanesi; // Kendi namespace'imizi using ile ekliyoruz

class Program
{
    static void Main(string[] args)
    {
        // using GeometriKutuphanesi; sayesinde artık sadece Nokta yazabiliriz.
        Nokta p1 = new Nokta();
        p1.X = 100;
        p1.Y = 200;
```

```
        Console.WriteLine($"P1 (using ile): ({p1.X}, {p1.Y})");


        Dikdortgen d1 = new Dikdortgen();
        d1.SolUstKose.X = 15; // SolUstKose de bir Nokta struct'ı!
        d1.SolUstKose.Y = 25;
        d1.Genislik = 150;
        d1.Yukseklik = 75;


        // Nokta struct'ı içindeki X ve Y'ye erişim
        Console.WriteLine($"Dikdörtgen (using ile): Sol Üst=({d1.SolUstKose.X},{d1.SolUstKose.Y}),
Genişlik={d1.Genislik}, Yükseklik={d1.Yukseklik}");


        Console.ReadKey();
    }
}
```

Kod Açıklaması:

1. GeometrikSekiller.cs dosyasında GeometriKutuphanesi adında bir namespace tanımladık ve içine Nokta ve Dikdortgen struct'larını yerleştirdik. Struct'ların public olmasına dikkat ettik.
2. Program.cs dosyasının en başına using GeometriKutuphanesi; satırını ekledik.
3. Artık Main metodu içinde Nokta ve Dikdortgen tiplerini doğrudan isimleriyle kullanabiliyoruz (GeometriKutuphanesi.Nokta yazmamıza gerek kalmadı).
4. Dikdortgen struct'ı, SolUstKose alanı olarak başka bir struct'ı (Nokta) içerebilir. Bu, karmaşık veri yapılarının struct'lar kullanılarak nasıl oluşturulabileceğine bir örnektir. d1.SolUstKose.X gibi iç içe alanlara erişebiliriz.

Eğer iki farklı namespace'de aynı isimde bir tip varsa (örneğin Ns1.A ve Ns2.A) ve her iki namespace'i de using ile eklerseniz, sadece A yazdığınızda derleyici hangi A'yı kastettiğinizi anlayamaz ve hata verir (ambiguity error). Bu durumda ya tam nitelikli isim kullanırsınız (Ns1.A) ya da using alias (takma ad) kullanırsınız:
using TakmaAd = Ns1.A;
Sonrasında TakmaAd olarak kullanabilirsiniz.

ÇOK BİLİNEN VE SIK KULLANILAN STRUCT'LAR (.NET İÇİNDE GELENLER)

.NET Framework / .NET Core, günlük programlama ihtiyaçlarınız için birçok hazır struct sunar. İşte en sık karşılaşacağınız ve kullanacağınız bazıları:

TEMEL VERİ TİPLERİ:

Aslında C#'ta gördüğünüz birçok temel veri tipi (primitive types) .NET altında bir struct olarak tanımlanmıştır!

- System.Int32 (takma adı: int)
- System.Int64 (takma adı: long)
- System.Boolean (takma adı: bool)
- System.Char (takma adı: char)
- System.Double (takma adı: double)
- System.Single (takma adı: float)
- System.Decimal (takma adı: decimal)
- System.Byte (takma adı: byte)
- System.SByte (takma adı: sbyte)
- System.Int16 (takma adı: short)
- System.UInt16 (takma adı: ushort)
- System.UInt32 (takma adı: uint)
- System.UInt64 (takma adı: ulong)

Bu tiplerin hepsi System namespace'i altında bulunur ve değer tipleridir (yani struct'tır).

SYSTEM.DATETIME

Amaç: Bir tarih ve zaman bilgisini temsil eder. (Örn: 17 Ekim 2023, Saat 10:30:15.500)

- **Özellikler/Kullanım:**
  - DateTime.Now: Mevcut sistem tarih ve saatini alır.
  - DateTime.UtcNow: Mevcut UTC (Eş Güdümlü Evrensel Zaman) tarih ve saatini alır.
  - DateTime.Today: Mevcut günün tarihini (saat kısmı 00:00:00) alır.
  - new DateTime(year, month, day, hour, minute, second): Belirli bir tarih ve saat oluşturur.
  - Year, Month, Day, Hour, Minute, Second, Millisecond, DayOfWeek gibi birçok özelliği vardır.
  - AddDays(), AddHours(), ToString() gibi metotları vardır.
- **Örnek:**
- ```
  DateTime simdi = DateTime.Now;
  ```
- ```
  Console.WriteLine($"Şu an: {simdi}"); // Formatlı bir şekilde yazar
  ```

- Console.WriteLine($"Yıl: {simdi.Year}, Ay: {simdi.Month}, Gün: {simdi.DayOfWeek}");
-
- DateTime dogumGunu = new DateTime(1990, 5, 15);
- Console.WriteLine($"Doğum Günü: {dogumGunu.ToShortDateString()}"); // Sadece tarih

SYSTEM.TIMESPAN

- **Amaç:** Bir zaman aralığını temsil eder. (Örn: 2 saat, 30 dakika, 15 saniye)
- **Özellikler/Kullanım:**
  - İki DateTime arasındaki farkı hesapladığınızda sonuç bir TimeSpan olur.
  - new TimeSpan(days, hours, minutes, seconds, milliseconds): Belirli bir zaman aralığı oluşturur.
  - FromDays(), FromHours(), FromMinutes() gibi statik metotlarla kolayca oluşturulabilir.
  - TotalDays, TotalHours, Minutes, Seconds gibi özellikleri vardır.
  - Add(), Subtract() gibi metotları vardır.
- **Örnek:**
- DateTime baslangic = DateTime.Now;
- // ... bir işlem yap ...
- System.Threading.Thread.Sleep(1500); // 1.5 saniye bekle (örnek amaçlı)
- DateTime bitis = DateTime.Now;
-
- TimeSpan gecenSure = bitis - baslangic; // İki DateTime farkı TimeSpan verir
- Console.WriteLine($"Geçen Süre (Toplam Milisaniye): {gecenSure.TotalMilliseconds}");
- Console.WriteLine($"Geçen Süre: {gecenSure.Hours} saat, {gecenSure.Minutes} dakika, {gecenSure.Seconds} saniye");
-
- TimeSpan onDakika = TimeSpan.FromMinutes(10);
- DateTime onDakikaSonra = DateTime.Now + onDakika; // DateTime + TimeSpan = DateTime
- Console.WriteLine($"On dakika sonra: {onDakikaSonra}");

SYSTEM.DRAWING.POINT (Genellikle Windows Forms, WPF veya görüntü işleme ile ilgili projelerde kullanılır)

- **Amaç:** 2D uzayda bir noktanın x ve y tamsayı koordinatlarını temsil eder.
- **Alanlar/Özellikler:** X, Y.
- IsEmpty özelliği vardır (koordinatlar 0,0 ise ve özel bir şekilde oluşturulmadıysa).
- Offset() metodu ile noktayı kaydırabilirsiniz.
- **Not:** Konsol uygulamalarında doğrudan kullanmak için System.Drawing.Common NuGet paketini projenize eklemeniz gerekebilir (özellikle .NET Core/.NET 5+). Alternatif olarak kendi Nokta struct'ınızı kullanabilirsiniz.
- **Örnek:**
- // System.Drawing.Point kullanmak için System.Drawing.Common NuGet paketi gerekebilir.
- // Veya projeniz Windows Forms/WPF ise zaten referanslıdır.
- System.Drawing.Point p = new System.Drawing.Point(10, 20);
- Console.WriteLine($"Nokta: ({p.X}, {p.Y})");

SYSTEM.DRAWING.POINTF

- Point struct'ının float (kesirli sayı) koordinatları kullanan versiyonudur.

2. **System.Drawing.Size** (Point gibi, genellikle UI veya görüntü işleme ile ilgili)
   - **Amaç:** Bir genişlik ve yüksekliği (tamsayı olarak) temsil eder.
   - **Alanlar/Özellikler:** Width, Height.
   - **Örnek:**
   - System.Drawing.Size boyut = new System.Drawing.Size(800, 600);
   - Console.WriteLine($"Boyut: Genişlik={boyut.Width}, Yükseklik={boyut.Height}");

SYSTEM.DRAWING.SIZEF

- Size struct'ının float (kesirli sayı) genişlik ve yükseklik kullanan versiyonudur.

SYSTEM.GUID **(Globally Unique Identifier)**

- **Amaç:** Evrensel olarak benzersiz bir tanımlayıcı oluşturur. 128-bitlik bir sayıdır. Veritabanlarında kayıtların birincil anahtarları olarak, dağıtık sistemlerde nesneleri eindeutig (benzersiz bir şekilde) tanımlamak için vb. sıkça kullanılır.
- **Özellikler/Kullanım:**
  - Guid.NewGuid(): Yeni, rastgele bir GUID oluşturur.
  - ToString(): GUID'ı string formatında (örn: "a9d1cc40-9a97-4611-9615-0b03e7a20bf3") verir.

- Guid.Empty: Tüm bitleri sıfır olan özel bir GUID değeridir.

- **Örnek:**
- Guid yeniId = Guid.NewGuid();
- Console.WriteLine($"Yeni Benzersiz ID: {yeniId}");
-
- string guidString = "f47ac10b-58cc-4372-a567-0e02b2c3d479";
- Guid mevcutId = new Guid(guidString); // String'den Guid oluşturma
- Console.WriteLine($"String'den GUID: {mevcutId}");

SYSTEM.NULLABLE<T> **(takma adı: T?)**

- **Amaç:** Normalde null değer alamayan değer tiplerinin (int, bool, DateTime gibi struct'lar) null değer alabilmesini sağlar. T burada herhangi bir değer tipini temsil eder.
- **Özellikler/Kullanım:**
  - int? sayi = null; veya Nullable<int> sayi = null;
  - HasValue: Nullable<T> örneğinin bir değer içerip içermediğini (null olup olmadığını) kontrol eder.
  - Value: Eğer HasValue true ise, içerdiği değeri döndürür. Eğer null ise ve Value özelliğine erişmeye çalışırsanız InvalidOperationException fırlatır.
  - Null birleştirme operatörü (??) ve null koşullu operatör (?.) ile sıkça kullanılır.
- **Örnek:**
- int? opsiyonelSayi = null;
- Console.WriteLine($"Opsiyonel Sayı (null mu?): {!opsiyonelSayi.HasValue}"); // True
-
- opsiyonelSayi = 100;
- if (opsiyonelSayi.HasValue)
- {
- Console.WriteLine($"Opsiyonel Sayı Değeri: {opsiyonelSayi.Value}");
- }
-
- // Null birleştirme operatörü: Eğer opsiyonelSayi null ise 0 ata, değilse kendi değerini ata.
- int gercekSayi = opsiyonelSayi ?? 0;

- Nullable<T> kendisi bir struct'tır! İçinde bool hasValue ve T value alanlarını barındıran akıllı bir yapıdır.

SYSTEM.COLLECTİONS.GENERIC.KEYVALUEPAIR<TKEY, TVALUE>

- **Amaç:** Bir anahtar (key) ve bir değer (value) çiftini tutmak için kullanılır. En çok Dictionary<TKey, TValue> koleksiyonunun elemanlarını temsil ederken karşımıza çıkar.
- **Özellikler:** Key, Value (her ikisi de sadece okunabilir - readonly).
- **Örnek:**
- KeyValuePair<string, int> urunFiyati = new KeyValuePair<string, int>("Elma", 5);
- Console.WriteLine($"Ürün: {urunFiyati.Key}, Fiyat: {urunFiyati.Value} TL");
-
- // Dictionary ile kullanım
- Dictionary<string, string> baskentler = new Dictionary<string, string>();
- baskentler.Add("Türkiye", "Ankara");
- baskentler.Add("Fransa", "Paris");
-
- foreach (KeyValuePair<string, string> kvp in baskentler)
- {
- Console.WriteLine($"Ülke: {kvp.Key}, Başkent: {kvp.Value}");
- }

Bu struct'lar, .NET'in sunduğu zengin kütüphanenin sadece küçük bir parçası. İhtiyaç duydukça Microsoft'un resmi dokümantasyonundan (docs.microsoft.com) daha fazlasını keşfedebilirsiniz.

ÖZET VE SONRAKİ ADIMLAR

Bugün struct'ların ne olduğunu, neden ve nasıl kullanıldığını öğrendik. En önemlisi:

- Struct'lar **değer tipleridir** ve genellikle **stack**'te saklanırlar.
- Metotlara parametre olarak geçildiğinde veya metotlardan geri döndürüldüğünde **kopyalanırlar**.
- Küçük, basit veri grupları için class'lara göre **performans avantajı** sunabilirler.
- unsafe kod ve sizeof ile bellekteki boyutlarını (eğitim amaçlı) gördük.
- namespace'ler kodumuzu organize etmemize ve isim çakışmalarını önlememize yardımcı olur.
- using direktifi ile başka namespace'lerdeki tipleri kolayca kullanabiliriz.

- .NET'in bize sunduğu DateTime, TimeSpan, Guid gibi birçok faydalı struct olduğunu gördük.

Bundan sonraki derslerimizde struct'ların constructor (yapıcı metot), metotlar, özellikler (properties) gibi daha gelişmiş yönlerine ve class'lar ile olan farklarına daha detaylı değineceğiz.