

Visual Studio.Net -C#

7. HAFTA

**Sanal Metotlar,Özet Sınıflar
ve Arayüzler,
İstisnai Durumlar, Temsilciler
ve Olaylar, Şablon Tipler**

Sanal Metotlar

- Temel sınıf türünden bir nesneye türemiş sınıf referansı aktarılabilirdi. Bu aktarım sonrasında bazı metotların nesnelere göre seçilmesi istenebilir. Bu durumda sanal metotlar tanımlanır.
- Sanal metotlar temel sınıflar içinde bildirilmiş ve türeyen sınıflar içinde de tekrar bildirilen metotlardır. İsim saklamaya benzemesine rağmen kullanımda farklıdır.
- Sanal metotlar sayesinde temel sınıf türünden bir referansa türeyen sınıf referansı aktarıldığında, temel sınıf referansı üzerinden kendisine aktarılan türeyen sınıfın sanal metodunu çağırabilir.
- Eğer türeyen sınıf sanal metodu devre dışı bırakmamışsa temel sınıftaki sanal metot çağrılır. Çağrılan metodun hangi türe ait olduğu ise çalışma zamanında belirlenir. Hangi metodun çağrılacağıın çalışma zamanında belirlenmesine geç bağlama (late binding) olarak isimlendirilir.
- Bu şekilde aynı nesne referansı üzerinden bir çok sınıfa ait farklı versiyonlardaki metotların çağrılabilmesi çok biçimlilik (polymorphism) olarak adlandırılır.

Sanal Metotlar

- Sanal metot bildirmek için **virtual** anahtar sözcüğü kullanılır.
- Türeyaen sınıfta, temel sınıftaki sanal metodu devre dışı bırakmak için **override** anahtar sözcüğü kullanılır.
- Türeyaen sınıfta devre dışı bırakılan metotların temel sınıftaki sanal metotların ismi ile aynı olmalıdır.
- Türeyaen sınıfta devre dışı bırakılan metotların parametrik yapısı temel sınıftaki metodun parametrik yapısı ile aynı olmalıdır.
- Statik metotlar sanal olarak bildirilemez.
- Türeyaen sınıflar, temel sınıftaki sanal metotları devre dışı bırakmak zorunda değildir. Bu durumda temel sınıf referansları üzerinden temel sınıfa ait metot çağrılır.

Sanal Metotlar

```
using System;
class A
{
    public void Metot()
    {
        Console.WriteLine("A sınıfı"); }
}
class B:A
{
    public void Metot()
    {
        Console.WriteLine("B sınıfı");
    }
    static void Main()
    {
        A nesneA=new A();
        B nesneB=new B();
        nesneA=nesneB;
        nesneA.Metot();
    }
}
```

Bu program ekrana **A sınıfı** yazar.

Çünkü nesneA nesnesi A sınıfı türündendir ve bu nesne üzerinden Metot() metoduna erişilirse A sınıfına ait Metot() metodu çalıştırılır.

Şimdi böyle bir durumda B sınıfına ait Metot() metodunun çalıştırılmasını sağlayacağız.

Sanal Metotlar

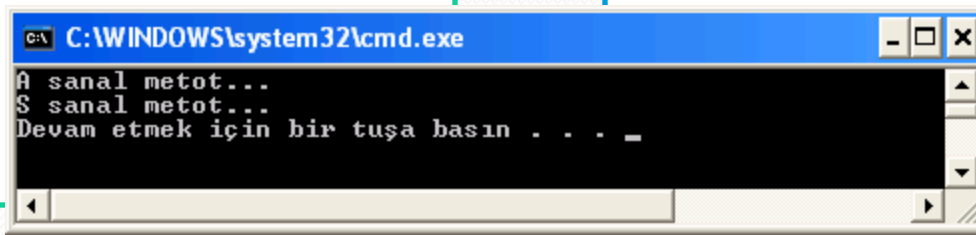
```
using System;
class A
{
    virtual public void Metot()
    {
        Console.WriteLine("A sınıfı"); }
}
class B:A
{
    override public void Metot()
    {
        Console.WriteLine("B sınıfı");
    }
    static void Main()
    {
        A nesneA=new A();
        B nesneB=new B();
        nesneA=nesneB;
        nesneA.Metot();
    }
}
```

- Bu program ekrana **B sınıfı** yazacaktır.
- Dikkat ettiyseniz bu programın öncekinden tek farkı A sınıfına ait metodun başına virtual anahtar sözcüğünün getirilmesi ve B sınıfına ait metodun başına da override anahtar sözcüğünün getirilmesi.
- Ana sınıftaki virtual anahtar sözcüğüyle ana sınıfa ait metodun bir sanal metot olmasını, türemiş sınıftaki override anahtar sözcüğüyle de ana sınıfa ait aynı adlı sanal metodun görmezden gelinmesini sağlandı. Bu sayede NesneA nesnesinin gizli türüne ait metot çalıştırıldı..

Sanal Metotlar

```
class A {  
    public A()  
    { }  
    public virtual void Metot()  
    { Console.WriteLine("A sanal metot..."); }  
}  
class T : A {  
    public T()  
    { }  
}  
class S : A {  
    public S() { }  
    public override void Metot() {  
        Console.WriteLine("S sanal metot...");  
    }  
}
```

```
class Program  
{  
    static void Main()  
    {  
        T t = new T();  
        S s = new S();  
        A a = new A();  
        a = t;  
        a.Metot();  
        a = s;  
        a.Metot();  
    }  
}
```



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window shows the output of the program: "A sanal metot...", "S sanal metot...", and "Devam etmek için bir tuşa basın . . . _". The text is displayed in a monospaced font on a black background.

Sanal Metotlar

```
using System;
class A
{ virtual public void Metot()
  { Console.WriteLine("A sınıfı"); }
}
class B:A
{ override public void Metot()
  { Console.WriteLine("B sınıfı"); }
}
class C:B
{
  public void Metot()
  { Console.WriteLine("C sınıfı"); }
  static void Main()
  {
    A nesneA=new A();
    C nesneC=new C();
    nesneA=nesneC;
    nesneA.Metot();
  }
}
```

B sınıfı yazar.

Çünkü C sınıfına ait metot override edilmediği için C'den önce gelen türeme zincirindeki son override edilen metot çalıştırılacaktır.

Sanal Metotlar

```
using System;
class A
{ virtual public int ozellik
  { set{}
    get{return 12;}
  }
}
class B:A
{ override public int ozellik
  { get{return 100;}
    set{Console.WriteLine("Bu bir denemedir");}
  }
static void Main()
{
  B nesne=new B();
  A nesne2=new A();
  nesne2=nesne;
  Console.WriteLine(nesne2.ozellik);
  nesne2.ozellik=200;
}
}
```

Ekran Çıktısı

100

Bu bir denemedir

Özet (Abstract) Sınıflar

- Nesne yönelimli programlamada sınıf hiyerarşisi oluşturulurken bazen hiyerarşinin en tepesinde bulunan sınıf türünden nesnelerin tek başına bir anlamı olmayabilir.
- Hiyerarşinin en tepesinde bulunan sınıfın kendisinden türetilecek diğer sınıflar için ortak özellikleri bir arada toplayan bir arayüz gibi davranması istenebilir.
- Bu tür sınıflara **özet sınıflar** adı verilir. Metotlar ve özellikler de özet olarak tanımlanabilir.
- Özet sınıflar **abstract** anahtar sözcüğü ile bildirilirler. Özet sınıf türünden nesneler tanımlanamaz. Özet sınıflar tek başlarına anlamlı bir nesne ifade etmezler. Kullanabilmeleri için mutlaka o sınıftan başka bir sınıf türetilmesi gerekmektedir.
- **abstract class**Sinif { }

Özet (Abstract) Sınıflar

- using System;
- abstract class A
- {
- public string Metot()
- { return "Deneme"; }
- }
- class B:A
- { static void Main()
- {
- B nesne1=new B();
- Console.WriteLine(nesne1.Metot());
- //A nesne2=new A();
- //Yukarıdaki satır olsaydı program hata verirdi. Çünkü özet sınıflar için nesne oluşturulmaz
- }
- }

Özet (Abstract) Metotlar

- Özet metotlar da yine **abstract** anahtar sözcüğüyle tanımlanır. Bu tür metotların gövdesi yoktur.
- Türeyen sınıfta mutlaka devre dışı bırakılmalıdır. Özet sınıflarda yapı itibariyle sanal oldukları için ayrıca **virtual** kullanılmaz.
- **abstract public void Metot();**
- Özet sınıflar içinde özet olmayan metotlar bildirilebilir. Fakat tersi özet olmayan sınıflarda özet metotların tanımlanması söz konusu değildir.
- Özet metotlar türeyen sınıfta devre dışı bırakılabilmeleri için **private** olarak tanımlanamazlar. **public** ya da **protected** olabilirler.

Özet (Abstract) Özellikler

- Özellikler de özet olarak bildirilebilir.
- `abstract public int X`
`{`
 `get;`
 `set;`
`}`
- Özet özellik bildiriminde kullanılan `set` veya `get` ifadelerinden hangileri kullanılmışsa türeyen sınıf bunları `override` ile mutlaka uygulamalıdır.

Özet (Abstract) Özellikler

- `using System;`
- `abstract class A`
- `{ abstract public int ozellik { set; get; } }`
- `class B:A`
- `{`
- `override public int ozellik`
- `{ get{return 100;}`
- `set{Console.WriteLine("Bu bir denemedir");}`
- `}`
- `static void Main()`
- `{ B nesne=new B();`
- `Console.WriteLine(nesne.ozellik); nesne.ozellik=200; }`
- `}`

sealed Anahtar Sözcüğü

- Bazı durumlarda sınıflardan türetme yapılması istenmeyebilir. Bunu sağlamak için sınıf tanımlamasının başına **sealed** anahtar sözcüğü eklenir.

```
sealed class Sinif  
{  
}
```

- Sınıflardan türetme yapılmaması türeyen sınıfın anlamsız olması ya da bazı üyelerin güvenliğini sağlamak olabilir. Tüm üyeleri statik olan sınıflar için de kullanılabilir.
- abstract (özet) sınıflar sealed olarak işaretlenemez.
- sealed sınıflara ek olarak static sınıflar ve yapılar da türetilmeyi desteklemez.

Arayüzler (Interfaces)

- Özet sınıfların benzeri olan bir yapı da **arayüzlerdir (interface)**, diğer sınıflar için ara yüz görevini üstlenir.
- Bütün metotları ve özellikleri özet olarak bildirilmiş sınıflardan çok fazla bir farkı yoktur. Dolayısıyla arayüzlerdeki metot ve özelliklerin gövdesi yazılamaz.
- **Arayüzler kısaca kendisini uygulayan sınıfların kesin olarak içereceği özellikleri ve metotları belirler.**
- Arayüzler, **interface** anahtar sözcüğü ile bildirilirler. Bir arayüzde özellik, metot, indeksleyici (indexer), temsilci (delegate) ve olay (event) bildirimi yapılabilir. Arayüz tanımlamalarının zorunlu olmasa da başına **“I”** harfinin eklenmesi tanımlamanın arayüz olduğunun kolayca anlaşılmasını sağlar.

Arayüzler (Interfaces)

- using System;
- interface arayüz
- { void metot1(); }
- class sınıf1:arayüz
- { public void metot1()
• { Console.WriteLine("sınıf1'in metot elemanı"); }
• }
- class sınıf2:arayüz
- { public void metot1()
• { Console.WriteLine("sınıf2 nin metot elemanı"); }
• }
- public class start
- { static void Main()
• { arayüz a;
• sınıf1 s1=new sınıf1();
• sınıf2 s2=new sınıf2();
• a=s1;
• a.metot1();
• a=s2;
• a.metot1();
• }
• }



```
C:\Documents and Settings\SoNDuRaK\Proje...
sınıf1'in metot elemanı
sınıf2 nin metot elemanı
Press any key to continue
```


Arayüzler (Interfaces)

- Arayüz tanımlamalarında dikkat edilecek bazı kısıtlamalar vardır:
 - Arayüz elemanları statik olamaz.
 - Arayüz elemanları public yapıdadır. Ayrıca erişim belirteci ile bildirilemez.
 - Üye değişken içeremezler.
 - Yapıcı ve yıkıcı metotlar tanımlanamaz ya da bildirilemez.

```
• interface IArayuz
• {
•     int Metot1();
•     int Metot2();
•     int sahteozellik { set; get; }
•     int this[int indeks] { get; }
• }
```

Arayüzlerin Uygulanması

- Arayüzlerin uygulanması sınıf türetmeyle aynı şekilde yapılır. Örnek:
- **class** A:IArayuz { *//IArayuz arayüzündeki bütün elemanları içermeli. }*
- Sınıflar arasında çoklu türetme olmamasına rağmen arayüzler çoklu olarak uygulanabilir. Uygulanacak arayüzler virgül ile ayrılır:
- **class** A: Arayuz1, Arayuz2 { *//Hem Arayuz1 hem de Arayuz2 arayüzündeki bütün elemanları içermeli. }*
- Arayüzler de sınıf türetmeyle aynı şekilde birbirlerinden türetilebilir. Bu durumda türemiş arayüz, ana arayüzün taşıdığı bütün elemanları taşır.
- Sınıflardan farklı olarak arayüzleri birden fazla arayüzden türetebiliriz. Örnek:
- **interface** Arayuz1 { **int** Metot1(); }
- **interface** Arayuz2 { **string** Metot2(); }
- **interface** Arayuz3: **Arayuz1, Arayuz2** { **double** Metot4(); }
- Burada Arayuz3'ü kullanan bir sınıf her üç metodu da içermelidir.

Arayüzler (Interfaces)

```
using System;

interface IArayuz
{
    int Metot1();
    void Metot2();

    int Sayi
    {
        get;
        set;
    }

    int this[int indeks]
    {
        get;
    }
}

class Program
{
    static void Main()
    {
        Sinif s = new Sinif();
    }
}
```

```
class Sinif : IArayuz
{
    private int sayi;

    public int Metot1()
    {
        return 0;
    }

    public void Metot2()
    {
    }

    public int Sayi
    {
        get { return sayi; }
        set { sayi = value; }
    }

    public int this[int indeks]
    {
        get { return indeks; }
        set { }
    }
}
```

Arayüzlerin Uygulanması

- Arayüzler türetilirken **new** anahtar sözcüğü ile temel arayüzdeki elemanlar gizlenebilir. Bu şekilde aynı isimli yeni elemanlar tanımlanabilir.
- **interface** Arayuz1 { **int** Metot1(); }
- **interface** Arayuz2:Arayuz1 { **new int** Metot1(); }
- Arayuz2'yi kullanan bir sınıfta Metot1()'in bildirimi yapıldığında geçerli bir bildirim olur mu? Bunun cevabı hayırdır. Arayuz2'nin Arayuz1'e ait metodu gizlemesi sonucu değiştirmez. Bu sorunu halletmek için sınıfımızı şöyle yazabiliriz.
- **class** deneme:Arayuz2
- { **int** Arayuz1.Metot1() { ... } }
- **int** Arayuz2.Metot1() { ... } }
- Burada hem Arayuz1'in hem de Arayuz2'nin isteklerini yerine getirdik. Eğer sınıfımız Arayuz2 arayüzünü kullanmasaydı programımız hata verirdi.
- Arayüzü uygulayan sınıf, arayüz dışında da elemanlara sahip olabilir. Yani istenildiği kadar üye eleman eklenebilir

Arayüzler (Interfaces)

- Arayüzler ile referanslar (nesneler) oluşturulabilir, ama **new** anahtar sözcüğü kullanılmaz .
Arayüz referansları tek başına bir anlam ifade etmez fakat kendisini uygulayan bir sınıf nesnesinin referansı atanabilir. Bu durumda arayüz referansı ile arayüzde bulunan metot ya da özellikler hangi sınıf referansı tutuluyorsa oradan çağrılabilir.
- using System;
- interface arayuz
- { int Metot(); }
- class A:arayuz
- { public int Metot()
- {return 0;}
- static void Main()
- { arayuz a;
- A s=new A();
- a=s;
- Console.WriteLine(a.Metot());
- }
- }

Arayüzler (Interfaces)

```
using System;

interface IArayuz
{
    int Metot1();
    void Metot2();
}

class Sinif : IArayuz
{
    int IArayuz.Metot1()
    {
        Console.WriteLine("Metot1");
        return 0;
    }

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

class Program
{
    static void Main()
    {
        Sinif s = new Sinif();
        IArayuz a;

        a = s;
        a.Metot1();
    }
}
```

//veya public int Metot1() olacak

Arayüzler (Interfaces)

- C# dilinde var olan arayüzleri uygulamanın bir yolu daha vardır. Buna açık arayüz uygulama denir. Bunun avantajları şunlardır:
 - Açık arayüz uygulama yöntemiyle istenirse sınıfa ait bazı üye elemanlara erişimi sınıf nesnelerine kapatırken, aynı üye elemanlara arayüz nesnesiyle erişimi mümkün kılabiliriz.
 - Bir sınıfa birden fazla arayüz uygulandığında eğer arayüzlerde aynı isimli üye elemanlar varsa isim çakışmasının önüne geçebiliriz.

Arayüzler (Interfaces)

- `using System;`
- `interface arayuz { void Metot(); }`
- `class sinif: arayuz`
- `{ void arayuz.Metot() { Console.WriteLine("Deneme"); }`
- `}`
- `class mainMetodu`
- `{`
- `static void Main()`
- `{ sinif nesne=new sinif();`
- `((arayuz)nesne).Metot();`
- `}`
- `}`
- Burada sinif türünden olan nesne arayuz türüne dönüştürüldü ve arayuz nesnesiyle Metot() metoduna erişildi. Direkt nesne üzerinden erişilemezdi.

Arayüzler (Interfaces)

- Aynı programı şöyle de yazabilirdik:
- `using System;`
- `interface arayuz`
- `{ void Metot(); }`
- `class sinif:arayuz`
- `{`
- `void arayuz.Metot() { Console.WriteLine("Deneme"); }`
- `}`
- `class mainMetodu`
- `{`
- `static void Main()`
- `{ arayuz nesne=new sinif();`
- `nesne.Metot();`
- `}`
- `}`

Partial (Kısmi) Tipler

- Şu ana kadar tanımlanan tip bildirimleri tek bir proje dosyası içerisinde bulunmaktaydı. Oysaki bir projede birden çok dosya bulunabilir.
- Sınıf, arayüz ve yapı tanımlamaları **partial** anahtar sözcüğü ile birden fazla dosyaya dağıtılabilir. Bütün dosyadaki tanımlamalar tek bir bildirimi göstermektedir. (isim alanları tanımlarken de benzer bir durum söz konusuydu.)

Partial (Kısmi) Tipler

- `Ozellikler.cs`

- ```
partial class Sinif
{
 public int x;
 public int y;
}
```

- `Metotlar.cs`

- ```
partial class Sinif
{
    public int Topla()
    {
    }

    public void EkranaYaz()
    {
    }
}
```

Partial (Kısmi) Tipler

- Birleştirilmesini istediğimiz bütün aynı isimli türleri partial olarak bildirmeliyiz. Yalnızca birisini partial olarak bildirmek yeterli değildir.
- Sınıflar, yapılar ve arayüzler partial olarak bildirilebilir.
- Kısmi türlerden biri sealed ya da abstract anahtar sözcüğüyle belirtilmişse diğerinin de belirtilmesine gerek yoktur.
- Partial türlerin partial olarak bildirildi diye illaki başka bir dosyayla ilişkilendirilmesine gerek yoktur.
- Partial türler minimum üye eleman düzeyinde iş görürler. Yani aynı metodun gövdesinin bir kısmını bir dosyada, başka bir kısmını başka bir dosyada yazmak partial türler ile de mümkün değildir.

İstisnai Durum Yönetimi (Exception Handling)

- Program ne kadar iyi yazılırsa yazılsın hataların oluşma olasılığı bir çok durumda mümkündür.
- Özellikle çalışma anında meydana gelebilecek hatalar kod yazılırken tespit edilemeyebilir.
- Tüm hatalara karşı tek tek tedbir almak çok zor bir iştir.
- Bu yüzden modern dillerde çalışma anında meydana gelebilecek hataları yakalamak için bir hata yakalama mekanizması mevcuttur.
- Çalışma zamanında beklenmeyen bir hata sonucunda oluşturulan nesnelere **istisnai durum sınıf nesneleri** denir. .NET sınıf kütüphanesinde çok sık oluşabilecek hatalara karşı istisnai durum sınıfları tasarlanmıştır. Bu sınıflar hata ile ilgili çeşitli bilgileri tutmaktadırlar.

İstisnai Durum Yönetimi (Exception Handling)

- .NET sınıf kitaplığında istisnai durum sınıfı hiyerarşisindeki en temel sınıf System isim alanı içerisinde yer alan **System.Exception**'dir. Exception sınıfı oluşan hatalar için çok genel bilgiler verdiği için türemiş diğer bilgiler sınıflar kullanılır:
 - SystemException, ArgumentException, StackOverflowException, ArithmeticException, IOException, IndexOutOfRangeException...
- İstisnai durumları yakalamak için dört tane anahtar sözcükten faydalanılır:
- **try, catch, finally** ve **throw**
- try, catch ve finally sözcükleri birer kod bloğunu temsil ederken, throw ise bir hatanın nesnesinin oluşturulmasını sağlar.

İstisnai Durum Yönetimi (Exception Handling)

- **try** : Hatanın kontrol edileceği kod bloğunun yazılacağı kısımdır. Yani bu blokta bir hata meydana gelirse ilgili hata sınıf nesnesi oluşturulacaktır.
- **catch** : try bloğunda yakalanan hataya göre işlemlerin yapılmasını sağlayan bloktur.
- **finally** : try ve catch bloklarında açılan kaynakların kapatılması burada gerçekleşir. Kodlar hata oluştursa dahi çalıştırılır. Bu bloğu belirtme zorunluluğu yoktur.
- **throw** : try bloğu içerisinde belirli bir hata olduğunda ilgili hata sınıfı nesnesini oluşturmak için kullanılır. Eğer try içindeki nesneler zaten hata oluşturuyorsa throw'un kullanılmasına gerek olmayabilir.

İstisnai Durum Yönetimi (Exception Handling)

- İstisnai durum yakalama mekanizması aşağıda verilmiştir.

- *try*

- {

- *//hatanın fırlatıldığı bölüm*

- }

- *catch(kuraldışı_tip a)*

- {

- *//kural dışı tip için yönetici*

- }

- *finally*

- {

- *//kaynakların temizlenmesi*

İstisnai Durum Yönetimi (Exception Handling)

```
• class Program {  
•     static void Main()  
•     {  
•         int[] x = new int[5];  
•         try  
•         {  
•             x[6] = 25; }  
•         catch  
•         {  
•             Console.WriteLine("Hata Oluştı..."); }  
•         finally  
•         {  
•             Console.WriteLine("Finally Bloğu..."); }  
•     }  
• }
```

İstisnai Durum Yönetimi (Exception Handling)

- İlgili hata sınıfı nesnesi kullanıldığında hata hakkında daha ayrıntılı bilgilere sahip olunabilir.
- Eğer catch tanımlamasında bir nesne bulunursa sadece o hata meydana geldiğinde catch bloğu çalıştırılacaktır.
- try bloğu içerisinde metotlar da çağrılabilir. Bu sayede metot içinde oluşan çalışma hataları da tespit edilebilir.
- Bir try bloğu içerisinde birden fazla hata oluşabilme ihtimali vardır. Bu durumda her bir hata durumu için birer catch bloğu tanımlanabilir.

İstisnai Durum Yönetimi (Exception Handling)

- try bloğundan sonra mutlaka catch ve ya finally bloğu gelmelidir. Bu bloklar arasında başka ifadeler yer almamalıdır.
- İç içe geçmiş try blokları da kullanmak mümkündür.
- **try**
- { //A
- **try**
- { //B }
- **catch** { //C //İçteki catch bloğu }
- **finally** { //D //İçteki finally bloğu }
- }
- **catch** { //Dıştaki catch bloğu }
- **finally** { //Dıştaki finally bloğu }
- }

İstisnai durum sınıfları

- System isim uzayında tanımlı, sıkça kullanılan kural dışı durumlar aşağıda verilmiştir
 - **ArrayTypeMismatchException:** Depolanmakta olan değerin tipi dizi tipi ile uyumsuz.
 - **DivideByZeroException:** Sıfıra bölünme hatasında fırlatılır.
 - **IndexOutOfRangeException:** Dizi indexi sınıf dışına taşıldığında fırlatılır.
 - **InvalidCatsException:** Programın çalışması sırasında geçersiz tür dönüşümü yapıldığında fırlatılır.
 - **OverflowExcepton:** Aritmetik taşma meydana geldiğinde fırlatılır.
 - **NullReferanceExcepton:** Null referans üzerinde işlem yapıldığında fırlatılır.
 - **StackOverflowException:** Yığın taşmıştır.
 - **FormatException:** Metotlarda yanlış formatta parametre gönderildiğinde fırlatılır.
 - **ArithmeticExcepton:** Bu sınıf ile DivideByZeroException ve OverflowException hataları yakalanabilir. Aritmetik işlemler sonucunda oluşan istisnai durumlarda fırlatılır.
 - **OutOfMemoryExcepton:** Programın çalışması için yeterli bellek miktarı olmadığında fırlatılır.

İstisnai Durum Yönetimi (Exception Handling)

```
• using System;
• class deneme
• {
•     static void Main()
•     {
•         try
•         { int[] a=new int[2];
•           Console.WriteLine(a[3]);
•         }
•         catch(IndexOutOfRangeException)
•         { Console.WriteLine("Dizi sınırları aşıldı"); }
•     }
• }
```

```
• using System;
• class deneme
• { static void Main()
•     { for(;;)
•         { try
•             { Console.Write("Lütfen çıkmak için 0 ya
•               da 1 girin: ");
•               int a=Int32.Parse(Console.ReadLine());
•               int[] dizi=new int[2];
•               Console.WriteLine(dizi[a]);
•               break; }
•             catch { continue; } } } }
```

Lütfen çıkmak için 0 ya da 1 girin: 2
Lütfen çıkmak için 0 ya da 1 girin: 5
Lütfen çıkmak için 0 ya da 1 girin: 1
0

İstisnai Durum Yönetimi (Exception Handling)

```
• using System;
• class hatayakalama1
• { public static void Main()
• { try
• { int a=50,b=0;
• Console.WriteLine(a/b);
• }
• catch(DivideByZeroException e)
• { Console.WriteLine("Sıfıra bölünme hatası yakalandı");
• }
• finally
• { Console.WriteLine("\n program sonlandırılıyor");
• }
• }
• }
• Çıktı: Sıfıra bölünme hatası yakalandı
• program sonlandırılıyor
```

```
• using System;
• class deneme
• { static void Main()
• { try { Metot(); }
• catch(IndexOutOfRangeException nesne)
• { Console.WriteLine("Metodu kullananda
hata yakalandı"); }
• }
• static void Metot()
• { try { int[] a=new int[2];
Console.WriteLine(a[3]); }
• catch(IndexOutOfRangeException nesne)
• { Console.WriteLine("Metodun kendisinde
hata yakalandı."); }
• } }
• Çıktı: Metodun kendisinde hata yakalandı.
```

throw anahtar sözcüğü

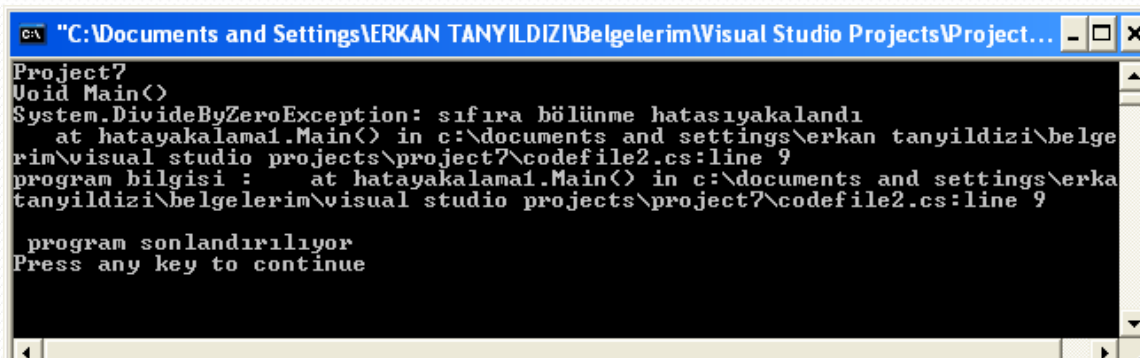
- Hatalar otomatik olarak üretilmelerine rağmen bazı özel durumlarda throw ile kendi istisnai durum nesnelerimizi oluşturabiliriz.
- throw anahtar sözcüğü istisnai durum sınıf nesnesi türünden bir nesne gönderir.
- Bir hata nesnesi throw anahtar sözcüğü yardımıyla şöyle fırlatılabilir:
 - **throw new** IndexOutOfRangeException("Dizinin sınırları aşıldı");
- veya
 - IndexOutOfRangeException nesne=**new** IndexOutOfRangeException("Dizinin sınırları aşıldı");
 - **throw** nesne;

İstisnai Durum Yönetimi (Exception Handling)

- **"exception"** Sınıfının Önemli Üye Elemanları :
 - **Message:** Hatanın özünü tarif eden bir karakter katarı içerir.
 - **Source:** catch bloğunda yakalanan istisnai durum nesnesinin gönderildiği uygulamanın ya da sınıfın adıdır.
 - **HelpLink:** Fırlatın hata ile ilgili yardım dosyasının yol bilgisini saklar.
 - **StackTrace:** İstisnai durumun fırlatıldığı metot ve programla ilgili bilgi içerir.
 - **TargetSite:** İstisnai durumu fırlatan metot ile ilgili bilgi verir
 - **InnerException:** Eğer catch bloğu içerisinde bir hata fırlatılırsa catch bloğuna gelinmesini sağlayan istisnai durumun exception nesnesidir.

İstisnai Durum Yönetimi (Exception Handling)

- using System;
- class hatayakalama1{
- public static void Main(){
- **try**
- { int a=50,b=0;
- **throw new DivideByZeroException**("sıfıra bölünme hatası yakalandı");
- Console.WriteLine(a/b);
- }
- **catch(DivideByZeroException e)**
- { Console.WriteLine(e.Source);
- Console.WriteLine(e.TargetSite);
- Console.WriteLine(e.ToString());
- Console.WriteLine("program bilgisi : "+e.StackTrace);
- }
- **finally**
- { Console.WriteLine("\n program sonlandırılıyor");
- }
- }
- }



```
C:\Documents and Settings\ERKAN TANYILDIZI\Belgelerim\Visual Studio Projects\Project...
Project7
Void Main<>
System.DivideByZeroException: sıfıra bölünme hatası yakalandı
   at hatayakalama1.Main<> in c:\documents and settings\erkan tanyildizi\belge
rim\visual studio projects\project7\codefile2.cs:line 9
program bilgisi :   at hatayakalama1.Main<> in c:\documents and settings\erka
tanyildizi\belgelerim\visual studio projects\project7\codefile2.cs:line 9

program sonlandırılıyor
Press any key to continue
```

İstisnai Durum Sınıfları Oluşturmak

- Kural dışı durumlara yönelik C#'ın sahip olduğu gücün bir parçası, programcının oluşturduğu kural dışı durumları kontrol altına alma becerisinden kaynaklanmaktadır.
- Programcı kendi oluşturduğu hataları kontrol etmek için yine kendi oluşturduğu kural dışı durumları kullanabilir. Kural dışı durum oluşturmak için yapılması gereken yalnızca Exception sınıfından türetilmiş bir sınıf tanımlamaktır.
- Oluşturulan kural dışı sınıfları Exception tarafından tanımlanan ve programcının oluşturduğu sınıfların kullanımına sunulan özelliklere ve metotlara otomatik olarak sahip olacaktır.

Örnek

```
• using System;
• class Notlar
• { private int mNot;
•   public int Not
•   { get{return mNot;}
•     set
•     { if(value>100)
•       throw new FazlaNotHatasi();
•       else if(value<0)
•       throw new DusukNotHatasi();
•       else
•       mNot=value;
•     }
•   }
•   public class FazlaNotHatasi:ApplicationException
•   {
•     override public string Message
•     {
•       get{ return "Not 100'den büyük olamaz.";}
•     }
•   }
```

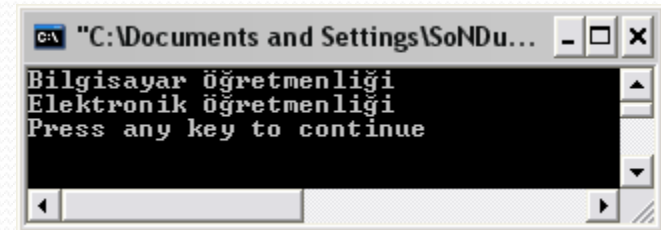
```
• public class DusukNotHatasi:ApplicationException
•   { override public string Message
•     { get{return "Not 0'dan küçük olamaz.";}
•     }
•   }
•   class Ana
•   { static void Main()
•     { try
•       { Notlar a=new Notlar();
•         Console.Write("Not girin: ");
•         int b=Int32.Parse(Console.ReadLine());
•         a.Not=b;
•         Console.WriteLine("Notu başarıyla girdiniz.");
•       }
•       catch(Exception nesne)
•       { Console.WriteLine(nesne.Message); }
•     }
•   }
```

Temsilciler

- Temsilcilerin nesneleri oluşturulabilir ve bu nesneler metotları temsil ederler. Temsilci bildirimi delegate anahtar sözcüğü ile yapılır. Bir temsilci bildirim taslağı şu şekildedir:
 - `delegate dönüşdeğeri temsilci(parametreler)`
`delegate int temsilci(int a,string b);`
- Temsilcilerin kullanılmasının amacı derleme zamanında belli olmayan metotların çalışma zamanını belirlemektir.
- Temsilciler kendi imzalarına uygun herhangi bir metodu temsil edebilirler. Bu metotlar statik de olabilir.
- `delegate` bir veri tipi olduğundan erişim belirteci ile tanımlanabilir.

Temsilciler

- using System;
- class delegeler
- { public delegate void temsilci();
- public static void Bilgisayar()
- { Console.WriteLine("Bilgisayar Öğretmenliği");
- }
- public static void Elektronik()
- { Console.WriteLine("Elektronik Öğretmenliği");
- }
- public static void Main()
- { temsilci nesne=new temsilci(Bilgisayar);
- nesne();
- nesne=new temsilci(Elektronik);
- nesne();
- }
- }



Çoklu temsilciler

- Bir temsilci birden fazla metodu temsil edebilir. + ve – operatörleri ile temsilciye metod ekleme ve çıkarma yapılabilir.
- + ve - metotları yerine daha pratik olması için += ve -= de kullanılabilir.
- Çoklu temsilci çağrıldığında metotlar temsilciye eklenme sırasına göre çalıştırılır.

Çoklu temsilciler

- `delegate void temsilci(int a);`
- `class deneme`
- `{ public static void Metot1(int a)`
- `{ Console.WriteLine("Metot1 çağrıldı." + a); }`
- `public static void Metot2(int a)`
- `{ Console.WriteLine("Metot2 çağrıldı." + a); }`
- `public static void Metot3(int a)`
- `{ Console.WriteLine("Metot3 çağrıldı." + a); }`
- `}`
- `class Program`
- `{ static void Main()`
- `{ temsilci nesne = null;`
- `nesne += new temsilci(deneme.Metot2); nesne += new temsilci(deneme.Metot1);`
- `nesne += new temsilci(deneme.Metot3); nesne(1);`
- `Console.WriteLine("***"); nesne -= new temsilci(deneme.Metot1);`
- `nesne(2);`
- `}`
- `}`

Ekran Çıktısı:
Metot2 Çağrıldı.1
Metot1 Çağrıldı.1
Metot3 Çağrıldı.1

Metot2 Çağrıldı.2
Metot3 Çağrıldı.2

Temsilciler

```
using System;

class Program
{
    public delegate void Temsilci(int a, int b);

    public static void Topla(int a, int b)
    {
        Console.WriteLine("Toplam:{0}", a + b);
    }

    public static void Cikar(int a, int b)
    {
        Console.WriteLine("Fark  :{0}", a - b);
    }

    static void Main()
    {
        Temsilci t = new Temsilci(Topla);
        t = t + new Temsilci(Cikar);

        t(10, 8);
    }
}
```


Delegate sınıfı

- Kendi yarattığımız temsilciler gizlice System isim alanındaki Delegate sınıfından türer. Dolayısıyla kendi yarattığımız temsilcilerin nesneleri üzerinden bu sınıfın static olmayan üye elemanlarına erişebiliriz.
- `using System;`
- `class Ana`
- `{ delegate void temsilci();`
- `static void Metot1()`
- `{ Console.WriteLine("Metot1"); }`
- `static void Metot2()`
- `{ Console.WriteLine("Metot2"); }`
- `static void Main()`
- `{ temsilci t=new temsilci(Metot1);`
- `t+=new temsilci(Metot2); t+=new temsilci(Metot3);`
- `Delegate d=t; temsilci c=(temsilci)d; // tür dönüşümüne dikkat`
- `c(); } }`

Ekran Çıktısı:
Metot1
Metot2

Delegate sınıfı

- Ayrıca Delegate sınıfı özet bir sınıf olduğu için new anahtar sözcüğüyle bu sınıf türünden nesne yaratamayız. Yalnızca kendi yarattığımız temsilci nesneleri üzerinden bu sınıfın üye elemanlarına erişebiliriz. Delegate sınıfının bazı üye elemanları:
- **GetInvocationList()** : çoklu temsilci yapısında bulunan metotları bir delegate dizisi olarak geri döndürür.
- **DynamicInvoke(object[] parametreler)** : bir temsilcinin temsil ettiği metotları tek tek çağırmak için kullanılır.
- **Combine(Delegate[] temsilciler)**: Çoklu bir temsilciye temsilciler ile belirtilen Delegate dizisindeki metotları ekler.

Delegate sınıfı

- `using System;`
- `class deneme`
- `{ delegate void temsilci();`
- `static void Metot1() { Console.WriteLine("Burası Metot1()"); }`
- `static void Metot2() { Console.WriteLine("Burası Metot2()"); }`
- `static void Main()`
- `{ temsilci nesne=null;`
- `nesne+=new temsilci(Metot1);`
- `nesne+=new temsilci(Metot2);`
- `Delegate[] dizi=nesne.GetInvocationList();`
- `dizi[0].DynamicInvoke();`
- `dizi[1].DynamicInvoke(); }`
- `}`
- Bu program alt alta Burası Metot1() ve Burası Metot2() yazacaktır.

Delegate sınıfı

- Eğer Delegate dizisindeki temsilci nesnelerinin temsil ettiği metotların parametreleri varsa bu parametreler DynamicInvoke() metoduna object dizisi olarak verilmelidir.
- **Combine(Delegate[] temsilciNesneleri)**
Combine(Delegate temsilciNesnesi1, Delegate temsilciNesnesi2)
- Birinci metot temsilciNesneleri adlı Delegate dizisindeki tüm metotları bir çoklu temsilci nesnesi olarak tutar. İkinci metot ise temsilciNesnesi1 ve temsilciNesnesi2 temsilci nesnelerindeki tüm metotları arka arkaya ekleyip bir temsilci nesnesi olarak tutar. Her iki metot da statictir. Örnek:
- *//Diğer kısımlar yukarıdaki programın aynısı. (Burası Main() metodu)*
- temsilci nesne=**null**;
- nesne+=**new** temsilci(Metot1);
- nesne+=**new** temsilci(Metot2);
- **Delegate[]** dizi=nesne.**GetInvocationList()**;
- **Delegate** nesne2=**Delegate.Combine**(dizi); nesne2.**DynamicInvoke()**;

Delegate sınıfı

- Şimdi ikinci metodu örnekleyelim:
- *//Diğer kısımlar yukarıdaki programın aynısı. (Burası Main() metodu)*
- temsilci nesne1=null;
- nesne1+=new temsilci(Metot1);
- nesne1+=new temsilci(Metot2);
- temsilci nesne2=new temsilci(Metot1);
- Delegate nesne3=Delegate.Combine(nesne1,nesne2);
- nesne3.DynamicInvoke();
- Bu program ekran çıktısı şöyle olmalıdır:
- Burası Metot1()
- Burası Metot2()
- Burası Metot1()

İsimsiz metotlar

- Şu ana kadar öğrendiklerimize göre temsilcilerin var olma nedeni metotlardı. Yani bir metot olmadan bir temsilci de olmuyordu. Ancak artık bir metot olmadan işe yarayan bir temsilci tasarlayacağız. Örnek:
- `using System;`
- `class isimsiz`
- `{ delegate double temsilci(double a,double b);`
- `static void Main()`
- `{ //temsilci nesnesine bir kod bloğu bağlanıyor.`
- `temsilci t= delegate (double a,double b)`
- `{`
- `return a+b;`
- `};`
- `//temsilci nesnesi ile kodlar çalıştırılıyor.`
- `double toplam= t(1d, 9d);`
- `Console.WriteLine(toplam);`
- `}`
- `}//Gördüğünüz gibi bir temsilci nesnesine bir metot yerine direkt kodlar atandı.`

Temsilcilerde kalıtım durumları

- `using System;`
- `delegate Ana temsilciAna();` `delegate Yavru temsilciYavru();`
- `class Ana { }`
- `class Yavru : Ana { }`
- `class program`
- `{static Ana MetotAna() {return new Ana(); } }`
- `static Yavru MetotYavru(){return new Yavru(); }`
- `static void Main()`
- `{ temsilciAna nesneAna = new temsilciAna(MetotYavru);`
- `nesneAna += new temsilciAna(MetotAna); nesneAna();`
- `temsilciYavru nesneYavru = new temsilciYavru(MetotYavru);`
- `nesneYavru();`
- `//ancak aşağıdaki kod hatalı.`
- `//nesneYavru = new temsilciYavru(MetotAna);`
- `}`
- `}`
- `// Yavru sınıfın referansı ana sınıfa atanamaz`

Olaylar (Events)

- Olaylar, temsilcilerin özel bir formudur. Olaylar ile yapılan işlemlerin tamamı temsilciler ile de yapılabilir.
- İşlemleri sadeleştirmek amacıyla **event** anahtar sözcüğü tanımlanmıştır.
- [Erişimbelirleyici] event [temsilci türü] [olay adı];

Olaylar (Events)

- **using** System;
- **delegate void** OlayYoneticisi(); *//Olay yöneticisi bildirimi*
- **class** Buton *//Olayın içinde bulunacağı sınıf bildirimi*
- { **public event** OlayYoneticisi **Click**; *//Olay bildirimi*
- **public void** Tiklandi() *//Olayı meydana getirecek metot*
- { **if**(Click!=**null**) Click(); }
- }
- **class** AnaProgram
- { **static void** Main()
- { Buton buton1=new Buton();
- buton1.Click+=new OlayYoneticisi(Click); *//Olay sonrası işletilecek metotların eklenmesi*
- buton1.Tiklandi(); *//Olayın meydana getirilmesi.*
- }
- *//Olay sonrası işletilecek metot*
- **static void** Click() *// Olaydan sonra çalıştırılacak metot ya da metotların static olma zorunluluğu yoktur.*
- { Console.WriteLine("Butona tıklandı."); }
- }*// Bu program ekrana Butona tıklandı. yazacaktır.*

Olaylar (Events)

- Şimdi programımızı derinlemesine inceleyelim. Programımızdaki en önemli satırlar:
- **delegate void** OlayYoneticisi(); ve Buton sınıfındaki **public event** OlayYoneticisi Click;
- satırıdır. Birinci satır bir temsilci, ikinci satır da bir olay bildirimidir. Bu iki satırdan anlamamız gereken Click olayı gerçekleştiğinde parametresiz ve geri dönüş tipi void olan metotların çalıştırılabileceğidir.
- Ana metottaki
- `buton1.Tiklandi();`
- satırı ile olay gerçekleştiriliyor. Tiklandi metoduna bakacak olursak;
- **public void** Tiklandi() { **if**(Click!=**null**) Click(); }
- Burada bir kontrol gerçekleştiriliyor. Burada yapılmak istenen tam olarak şu: Eğer Click olayı gerçekleştiğinde çalışacak bir metot yoksa hiçbir şey yapma. Varsa olayı gerçekleştir.
- `buton1.Click+=new OlayYoneticisi(Click);`
- Main() metodundaki bu satırla da buton1'in Click olayına bir metot ekliyoruz. Yani buton1 nesnesinin Click olayı gerçekleştiğinde Click metodu çalışacak.

Olaylar (Events)-Örnek

- **using** System;
- **delegate void** OlayYoneticisi(); *//Olay yöneticisi bildirimi*
- **class** Buton *//Olayın içinde bulunacağı sınıf bildirimi*
- { **public event** OlayYoneticisi Click; *//Olay bildirimi*
- **public void** Tiklandi() *//Olayı meydana getirecek metot*
- { **if**(Click!=null) Click(); } }
- **class** Pencere { **int** a;
- **public** Pencere(**int** a) { **this.a**=a; }
- **public void** Click()
- { Console.WriteLine("Pencere sınıfındaki metot çağrıldı - "+a); } }
- **class** AnaProgram {
- **static void** Main() {
- Buton buton1=new Buton(); Pencere nesne1=new Pencere(1), nesne2=new Pencere(2);
- buton1.Click+=new OlayYoneticisi(Click); *//Olay sonrası işletilecek metotların eklenmesi*
- buton1.Click+=new OlayYoneticisi(nesne1.Click);
- buton1.Click+=new OlayYoneticisi(nesne2.Click);
- buton1.Tiklandi(); *//Olayın meydana getirilmesi.*
- } *//Olay sonrası işletilecek metot*
- **static void** Click() { Console.WriteLine("Butona tıklandı."); } }

Bu programın ekran çıktısı şöyle olacaktır:

Butona tıklandı.

Pencere sınıfındaki metot Çağrıldı - 1

Pencere sınıfındaki metot Çağrıldı - 2

Olaylar (Events)

- Olay yöneticisine yeni metotlar ekleme ve çıkarma anında kontroller yapmak için set- get yapısının benzeri bir yapı olan **add** ve **remove** blokları kullanılabilir.
- **+= add** bloğunu, **-=** ise **remove** bloğunu çalıştırır.
- add ve remove bloklarının her ikisi de mutlaka aynı anda tanımlanmalıdır. Delege olay sayısı kadar dizi tanımlanmalıdır.
- Olaylar da bir tür olduğu için arayüzlerde bildirilebilir. Ayrıca sanal (virtual) ve özet (abstract) olarak da bildirilebilirler.
- Yalnız add ve remove içeren olaylar özet olarak bildirilemez.
- Olayların en büyük kullanım alanı nesneler arası minimum bağlantıyla mesajlaşmadır.

Olaylar (Events)

- Örnek program:
- **using** System;
- **class** deneme {
- **delegate void** OlayYoneticisi();
- **event** OlayYoneticisi Olay
- {
- **add** { Console.WriteLine("Olaya metot eklendi."); }
- **remove** { Console.WriteLine("Olaydan metot çıkarıldı."); }
- }
- **static void** Main()
- { deneme d=new deneme();
- d.Olay+=new OlayYoneticisi(d.Metot1);
- d.Olay+=new OlayYoneticisi(d.Metot2);
- d.Olay-=new OlayYoneticisi(d.Metot1); }
- **void** Metot1(){} **void** Metot2(){} }

Bu programın ekran çıktısı şöyle olur:
Olaya metot eklendi.
Olaya metot eklendi.
Olaydan metot Çıkarıldı.

Örnek

```
• using System;
• delegate void OlayYoneticisi();
• class deneme
• { OlayYoneticisi [ ] evnt = new OlayYoneticisi[3];
•     public event OlayYoneticisi Olay
•     { add
•         { for (int i = 0; i < 3; i++)
•             { if (evnt[i] == null)
•                 { evnt[i] = value;
•                     Console.WriteLine((i + 1) + ". Olay"); }
•                 if (i == 3)
•                     Console.WriteLine("Olay listesi Dolu");
•             }
•         }
•     }
•     remove
•     { for (int i = 0; i < 3; i++)
•         { if (evnt[i] == value)
•             { evnt[i] = null;
•                 Console.WriteLine(" Olay çıkarıldı.");
•             }
•             if (i == 3)
•                 Console.WriteLine("Olay bulunamadı");
•         }
•     }
• }
```

```
• // olayları ateşlemek için
• public void Olayim()
•     { for (int i = 0; i < 3; i++)
•         { if (evnt[i] != null) evnt[i]();
•         }
•     }
• }
• class Program {
•     static void Metot1() {Console.WriteLine("Metot1");}
•     static void Metot2() {Console.WriteLine("Metot2");}
•     static void Main() {
•         deneme d = new deneme();
•         d.Olay += new OlayYoneticisi(Metot1);
•         d.Olay += new OlayYoneticisi(Metot2);
•         d.Olay -= new OlayYoneticisi(Metot1);
•     }
• }
```

Şablon tipler

- Şimdiye kadar bir sınıftaki tür bilgileri sınıf bildirimi esnasında belliydi. Ancak artık şablon tipler sayesinde sınıftaki istediğimiz türlerin nesne yaratımı sırasında belli olmasını sağlayacağız. Üstelik her nesne için söz konusu türler değişebilecek. Şablon tipler sayesinde türden bağımsız işlemler yapabileceğiz. Şimdi şablon tipleri bir örnek üzerinde görelim:
- `using System;`
- `class Sinif<SablonTip> { public SablonTip Ozellik; }`
- `class AnaProgram {`
- `static void Main() {`
- `Sinif<int> s1=new Sinif<int>(); s1.Ozellik=4;`
- `Sinif<string> s2=new Sinif<string>(); s2.Ozellik="deneme";`
- `Console.WriteLine(s1.Ozellik+" "+s2.Ozellik); }`
- `}`
- Bu programda s1 nesnesinin Ozellik özelliğinin geri dönüş tipi int, ancak s2 nesnesinin Ozellik özelliğinin geri dönüş tipi stringdir.
- Sınıflar, arayüzler, yapılar, temsilciler ve metotlar şablon tip olarak bildirilebilir.

Şablon tipler-Sınıf aşırı yükleme

- Bir sınıf (veya yapı, arayüz vs.) birden fazla şablon tip alabilir.
- Örnek:
- `class Sinif<SablonTip1, SablonTip2>`
- Bu durumda bu sınıf türünden nesne şunun gibi yaratılır:
- `Sinif<int,string> s=new Sinif<int,string>();`
- Aynı isim alanında isimleri aynı olsa bile farklı sayıda şablon tipi olan sınıflar bildirilebilir. Buna sınıf aşırı yükleme denir.

Şablon tipler arasında türeme

- Şablon tipler arasında türeme ile ilgili çok fazla kombinasyon düşünebiliriz. Ancak biz burada karşımıza en çok çıkacak iki kombinasyondan bahsedeceğiz:
- `class AnaSinif<T>`
- `{ //... }`
- `class YavruSinif<T, Y>:AnaSinif<T> {`
- `//Gördüğünüz gibi yavru sınıf en az ana sınıfın şablon tipini içerdi. }`
- `class YavruSinif2<T>:AnaSinif<int>`
- `{ /*Gördüğünüz gibi yavru sınıf ana sınıfın şablon tipine belirli bir tür koydu. Böylelikle yavru sınıftaki ana sınıftan gelen T harfleri int olarak değiştirilecektir.*/ }`

Şablon tipler ve arayüzler

- **Arayüzlerin de şablon tipli versiyonları yazılabilir.** Örneğin System.Collections.Generic isim alanında birçok şablon tipli arayüz bulunmaktadır. Bunlardan birisi de IEnumerable arayüzünün şablon tipli versiyonudur. IEnumerable arayüzünün şablon tipli versiyonu aşağıdaki gibidir.
- **interface** IEnumerable<T>:IEnumerable
- { IEnumerator<T> GetEnumerator(); }
- Buradan anlıyoruz ki bu arayüzü uygulayan sınıf geri dönüş tipi IEnumerator<T> olan GetEnumerator() metodunu ve IEnumerable arayüzünün içerdiği tüm üyeleri içermeli. Bu arayüzü bir sınıfta şöyle uygulayabiliriz:
- **class** Sinif:IEnumerable<int>
- { IEnumerator IEnumerable.GetEnumerator() { //... }
- IEnumerator<int> IEnumerable<int>.GetEnumerator() { //... }
- }
- Burada hem IEnumerable hem de IEnumerable<T> arayüzlerinin metotları uygulandı. Çünkü IEnumerable<T> arayüzü, IEnumerable arayüzünden türemiştir.

Şablon tiplerin metotlara etkisi

- Bildiğiniz gibi sınıf düzeyinde bir şablon tip belirlediğimizde bu şablon tip metotların geri dönüş tipine, parametre tiplerine koyulabilir. Ancak bunun sonucunda bazen bir çakışma oluşabilir. Örneğin:
- `class Sinif<T>`
- `{ public int Metot(T a) { return 0; } }`
- `public int Metot(int a) { return 1; } }`
- Bu sınıf türünden nesneyi şöyle tanımlayıp kullanırsak;
- `Sinif<int> a=new Sinif<int>();`
- `Console.WriteLine(a.Metot(10));`
- Bu gibi bir durumda `Metot(int a)` metodu çalışır. Yani normal tipli versiyon şablon tipli versiyonu gizler.
- Şablon tipler metotlarla kullanıldığı gibi özellik, indeksleyici ve olaylarla da kullanılabilir

Şablon tipler-default operatörü

- Bildiğimiz gibi şablon tipler herhangi bir tipi temsil ederler. Bu yüzden C#, yalnızca bazı türlere özgü olan operatörler (+, -, *, ...) kullanılmasına izin vermediği gibi şablon tip türünden bir nesne yaratılıp bu nesneye bir değer verilmesine engel olur.
 - Örnekler:
 - `class Sinif<T> { public int Metot(T a) { return a+2; } }`
 - **Bu sınıf derlenmez.** Çünkü T tipinin hangi tip olduğunu bilmiyoruz. + operatörü bu tip için aşırı yüklenmiş olmayabilir. Bu yüzden C# bu gibi tehlikeli durumları önlemek için bu tür bir kullanımı engeller. Başka bir kullanım:
 - `class Sinif<T> { public int Metot(T a) { T nesne=0; } }`
 - **Yine burada da bir hata söz konusudur.** Çünkü her tipten nesneye 0 atanmayabilir. Benzer şekilde yapı nesnelere de null değer atanamaz.
 - **default** operatörü bir şablon tipin varsayılan değerini elde etmek için kullanılır. Örnek:
 - `class Sinif<T> { public void Metot() { T nesne=default(T); } }`
 - Varsayılan değer bazı türler için 0 (int, short, float vs.) bazı türler için null (tüm sınıflar) bool türü için de false'tur.

Şablon tipler-KISITLAR

- Şablon tür nesneleriyle herhangi bir üye de (metot, özellik vs.) çalıştıramayız. Çünkü şablon tipin ne olduğunu bilmediğimiz için olmayan bir metodu çalıştırmaya çalışıyor olabiliriz. Tüm bunların sebebi aslında şablon sınıf türünden nesne yaratırken şablonu herhangi bir tür yapabilmemizdi. Eğer şablona koyulabilecek türleri kısıtlarsak bunları yapabiliriz.
- C#'ta şu kısıtlar bulunmaktadır:
- **struct** şablon tip yalnızca yapılar olabilir.
- **class** şablon tip yalnızca sınıflar olabilir.
- **new()** şablon tip yalnızca nesnesi yaratılabilen tiplerden olabilir. (tür abstract, static, vb. olamaz)
- **türetme** şablon tip mutlaka belirtilen bir türden türemiş olmalıdır.
- **interface** şablon tip mutlaka belirtilen bir arayüzü uygulamalıdır.
- Kısıtlar **where** anahtar sözcüğüyle yapılmaktadır. Şimdi türetme kısıtına bir örnek verelim:

Şablon tipler-KISITLAR

- `class A{}`
 - `class B:A{}`
 - `class C:A{}`
 - `class D<T> where T:A{}`
 - `class AnaProgram {`
 - `static void Main() {`
 - `D nesne1=new D(); D<C> nesne2=new D<C>();`
 - *//Aşağıdaki olmaz.*
 - *//D<int> nesne3=new D<int>(); }*
 - `}`
-
- Türeme kısıtı sayesinde şablon tip nesnesiyle bir metodu çağırabiliriz. Bunun içinde ana sınıfa o metodu koyarız, bu sınıftan türeyen yavru sınıflarda o metodu override ederiz. Son olarak şablon tipe ana sınıftan türeme zorunluluğu getiririz. Ana sınıftan türeyen bütün sınıflar söz konusu metodu içereceği için artık C#, bu metodu şablon tip nesneleri üzerinden çağırmamıza izin verecektir.

Şablon tipler-KISITLAR

- Ancak halen new operatörüyle şablon tip türünden nesne oluşturamayız. Bunun için şablon tipe new() kısıtını eklemeliyiz. Yani şablon tipe yalnızca nesnesi oluşturulabilen türler koyulabilecek.
- Örnek:
- `class Sinif<T> where T:new()`
- `{ ... }`
- Artık bu sınıfın içinde T türünden nesneleri new operatörüyle oluşturabiliriz. Tabii ki C# şablon tipe nesnesi oluşturulamayan bir tür koyulmasını engelleyecektir. Başka bir önemli kısıt ise arayüz kısıtıdır. Arayüz kısıtı sayesinde şablon tipe koyulan tipin mutlaka belirli bir arayüzü kullanmış olmasını sağlarız.

Şablon tipler-KISITLAR

- Örnek program:
- `using System;`
- `class Karsilastirici<T> where T:IComparable<T>`
- `{ public static int Karsilastir(T a,T b)`
- `{ return a.CompareTo(b); } }`
- `class Program {`
- `static void Main() {`
- `int s1=Karsilastirici<int>.Karsilastir(4,5); int`
`s2=Karsilastirici<float>.Karsilastir(2.3f,2.3f); int`
`s3=Karsilastirici<string>.Karsilastir("Ali","Veli"); int`
`s4=Karsilastirici<DateTime>.Karsilastir(DateTime.Now,DateTime.Now.AddDays(1));`
`Console.WriteLine("{0} {1} {2} {3}",s1,s2,s3,s4); } }`
- Bu program ekrana -1 0 -1 -1 çıktısını verecektir. .Net Framework kütüphanesindeki IComparable arayüzünde CompareTo() metodu bulunmaktadır. Dolayısıyla bu arayüzü uygulayan her sınıfta da bu metot bulunur. Bu sayede de C# şablon tip nesnesinden ilgili metodun çağrılmasına izin vermiştir.

Şablon tipler-KISITLAR

- Diğer önemli iki kısıt ise şablon tipe yalnızca bir sınıf koyulabilmesini sağlayan class ve aynı şeyi yapılar için yapan struct kısıtlarıdır. Örnekler:
- `class Sinif1<T> where T:class { }`
- `class Sinif2<T> where T:struct { }`
- Tabii ki bir şablon tipe birden fazla kısıt eklenebilir. Bu durumda kısıtlar virgülle ayrılır. Örnek:
- `class Sinif<T> where T:class, IComparable, new() { }`
- Kısıtlarda diğerlerinin sırası önemli değildir. Ancak -varsa- new() kısıtı en sonda olmalıdır. Bir sınıfa eklenen birden fazla şablon tip varsa her biri için ayrı ayrı kısıtlar koyulabilir. Örnek:
- `class Sinif<T,S> where T:IComparable,IEnumerable where S:AnaSinif`
- **NOT:** Main() metodu mutlaka şablon tipli olmayan bir sınıfın içinde olmalıdır. Ayrıca Main() metodu ya bir sınıfın içinde ya da bir yapının içinde olmalıdır. Diğer bir deyişle çalışabilir her program en az bir sınıf ya da yapı içermelidir.

Şablon tipli metotlar

- Şablon tipler metot ve temsilci düzeyinde de tanımlanabilir.
- Örnek:
- `using System;`
- `class karsilastirma {`
- `static void Main() {`
- `Console.WriteLine(EnBuyuk<int>(4,5));`
- `Console.WriteLine(EnBuyuk<string>("Ali","Veli"));`
- `}`
- `static T EnBuyuk<T>(T p1,T p2) where T:IComparable`
- `{ T geridonus=p2;`
- `if(p2.CompareTo(p1)<0) geridonus=p1;`
- `return geridonus; } }`
- Bu program alt alta 5 ve Veli yazacaktır.

Şablon tipi çıkarsama

- Az önceki örneğin sadece Main() metodunu alalım:
- **static void** Main()
- { Console.WriteLine(EnBuyuk<int>(4,5));
Console.WriteLine(EnBuyuk<string>("Ali","Veli"));
- }
- Burada parametrelerin türleri belli olduğu hâlde ayrıca int ve string türlerini de belirttik. İstersek bunu şöyle de yazabilirdik:
- **static void** Main()
- { Console.WriteLine(EnBuyuk(4,5));
Console.WriteLine(EnBuyuk("Ali","Veli")); }
- Bu programda metot şöyle düşünecektir: "Benim parametrelerim T türünden, o hâlde yalnızca parametreye bakarak T'nin hangi tür olduğunu bulabilirim." Gördüğünüz gibi metotların aşırı yükleyerek saatlerce uğraşarak yazabileceğimiz programları şablon tipli metotlar kullanarak birkaç dakikada yazabiliyoruz.

Şablon tipli temsilciler

- Temsilciler de şablon tip alabilirler. Bu sayede temsilcinin temsil edebileceği metod miktarını artırabiliriz. Örnek:
- **using** System;
- **delegate** T Temsilci<T>(T s1,T s2);
- **class** deneme {
- **static int** Metot1(**int** a,**int** b) {**return** 0;}
- **static string** Metot2(**string** a,**string** b){**return** null;}
- **static void** Main() {
- Temsilci<**int**> nesne1=new Temsilci<**int**>(Metot1);
- Temsilci<**string**> nesne2=new Temsilci<**string**>(Metot2);
- Console.**WriteLine**(nesne1(1,2)); Console.**WriteLine**(nesne2("w","q")); }
- }
- Temsilci şablon tipleri de kısıt alabilirler. Örnek:
- **delegate** T Temsilci<T>(T s1,T s2) **where** T:**struct**
- Burada T yalnızca bir yapı olabilir. Yani bu temsilcinin temsil edeceği metodun parametreleri ve geri dönüş tipi yalnızca bir yapı olabilir.

null değer alabilen yapı nesneleri

- Bildiğiniz gibi yapı nesneleri null değer alamaz. Örneğin şu kod hatalıdır:
- `int a=null;`
- Ancak System isim alanındaki `Nullable<T>` yapısı sayesinde yapı nesnelerinin de null değer alabilmesini sağlayabiliriz. `System.Nullable<T>` yapısı şu gibidir:
- `public struct Nullable<T> where T:struct`
- `{ private T value;`
- `private bool hasValue;`
- `public T Value{get {...} }`
- `public bool HasValue{get {...} }`
- `public T GetValueOrDefault(){...}`
- `}`
- Bu yapıya göre null değer alabilen yapı nesneleri şöyle oluşturulur:
- `Nullable<int> a=new Nullable<int>();`
- `a=5; a=null;`
- `Nullable<double> b=new Nullable<double>(2.3);`
- `Console.WriteLine("{0}{1}",a,b);`

? İşareti

- ? takısı kısa yoldan nullable tipte nesne oluşturmak için kullanılır. Örneğin:
- `Nullable<double> d=10;` ile `double? d=5;`
- satırları birbirine denktir.
- Nullable nesneleri normal nesnelere tür dönüştürme operatörünü kullanarak dönüştürebiliriz. Ancak nullable nesnenin değeri null ise çalışma zamanı hatası alırız. Örnek: `int? a=5; int b=(int)a;`
- Benzer şekilde tür dönüşüm kurallarına uymak şartıyla farklı dönüşüm kombinasyonları da mümkündür:
- `int? a=5; double b=(double)a;`
- Normal ve nullable nesneler arasında ters dönüşüm de mümkündür. Normal nesneler nullable nesnelere bilinçsiz olarak dönüşebilir. Örnek:
- `int a=5; int? b=a;`
- Nullable nesneler operatörler ile kullanılabilir. Örnek:
- `int? a=5; int? b=10; int c=(int)(a+b);`
- Burada `a+b` ifadesinin ürettiği değer yine `int ?` türünden olduğu için tür dönüşüm operatörü kullanıldı.

? İşareti

```
• class NullableExample
• { static void Main()
• { int? num = null;
•     Nullable<int> a = new Nullable<int>();
•     a = 5; a = null;
•     if (num.HasValue == true)
•     { Console.WriteLine("num = " + num.Value);}
•     else
•     { Console.WriteLine("num = Null");}
•     //y değerine 0 atanıyor
•     int y = num.GetValueOrDefault();
•     // num.Value throws an InvalidOperationException if num.HasValue is false
•     try
•     { y = num.Value; }
•     catch (InvalidOperationException e)
•     { Console.WriteLine(e.Message); }
• }
• }
```

?? Operatörü

- ?? operatörü `Nullable<T>` yapısındaki `GetValueOrDefault()` metoduna benzer şekilde çalışır. Örnek:
- `int? a=null; int b=a??0;`
- Burada eğer `a` null ise ?? operatörü 0 değerini döndürür.
- ?? operatörünün döndürdüğü değer normal (nullable olmayan) tiptedir.
- Eğer ilgili nullable nesne null değilse olduğu değeri döndürür. Başka bir örnek:
- `int? a=null; int b=a??50;`
- Burada ise eğer `a` null ise 50 döndürülür. Yani ?? operatöründe `GetValueOrDefault()` metodundan farklı olarak ilgili nesne null olduğunda döndürülecek değeri belirleyebiliyoruz.

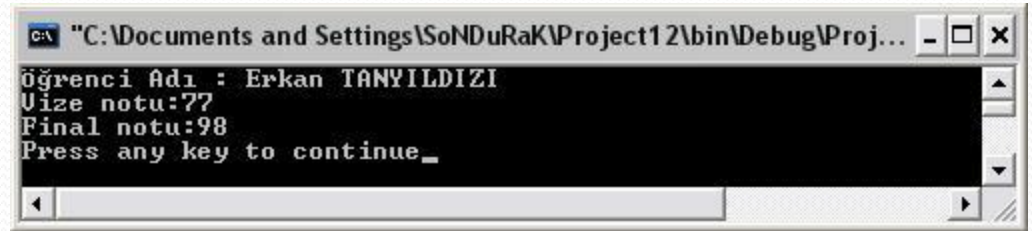
Visual Studio.Net -C#

8. HAFTA

Göstericiler, Ön işlemciler, Temel Giriş Çıkış İşlemleri (I/O)

Özet (Abstract) Sınıflar-Örnek

```
• using System;
• abstract class Ogr_not {
•     public int vize;
•     public int final;
•     public Ogr_not(int v,int f) {
•         vize=v;   final=f;    }
• }
•
• class ogr:Ogr_not {
•     public string str;
•     public ogr(string ad,int vize,int final):base(vize,final)
•     {   str=ad; }
•     public void ad_göster()
•     {
•         Console.WriteLine("Öğrenci Adı : "+str);
•         Console.WriteLine("Vize notu:"+vize+"\n"+"Final notu:" +final);
•     }
• }
• class Ana_sınıf
• {
•     static void Main()
•     {
•         ogr d=new ogr("Erkan TANYILDIZI",77,98);
•         d.ad_göster();
•     }
• }
```



```
C:\Documents and Settings\SoNDuRaK\Project12\bin\Debug\Proj...
Öğrenci Adı : Erkan TANYILDIZI
Vize notu:77
Final notu:98
Press any key to continue_
```

Özet (Abstract) Sınıflar-Örnek

```
using System;

abstract class A
{
    public int x;
    abstract public int y
    {
        set;
        get;
    }

    public A(int x)
    {
        this.x = x;
    }

    abstract public void Metot();
}

class S : A
{
    int z;

    public S(int x):base(x)
    {
    }
}
```

```
    public override int y
    {
        get
        {
            return z;
        }
        set
        {
            z = value;
        }
    }

    public override void Metot()
    {
        Console.WriteLine(x);
        Console.WriteLine(y);
    }
}

class Program
{
    static void Main()
    {
        S s = new S(5);
        s.y = 2 * s.x;
        s.Metot();
    }
}
```

Örnekler

```
using System;
class A
{
    public void Metot1()
    {    Metot2(); }
    public void Metot2()
    {    Console.WriteLine("A sınıfı"); }
}
class B:A
{
    public new void Metot2()
    {    Console.WriteLine("B sınıfı"); }
}
class Ana
{
    static void Main()
    {
        B b=new B();
        b.Metot1();
    }
}
```

Bu programda ekrana A sınıfı yazılır.

```
using System;
class A
{    public void Metot1()
    {    Metot2(); }
    public void Metot2()
    {    Console.WriteLine("A sınıfı"); }
}
class B:A
{    public new void Metot1()
    {    Metot2(); }
    public new void Metot2()
    {    Console.WriteLine("B sınıfı"); }
}
class Ana
{    static void Main()
    {    B b=new B();
        b.Metot1();
    }
}
```

Bu programda ekrana B sınıfı yazılır.

Örnekler

```
using System;
class A
{
    public void Metot1()
    {    Metot2(); }
    virtual public void Metot2()
    {    Console.WriteLine("A sınıfı"); }
}
class B:A
{
    override public void Metot2()
    {    Console.WriteLine("B sınıfı"); }
}
class Ana
{
    static void Main()
    {
        B b=new B();
        b.Metot1();
    }
}
```

Bu programda ekrana B sınıfı yazılır.

```
using System;
class A
{    public int OzellikA;
    public A(int a)
    {    OzellikA=a; }
}
class B:A
{    public int OzellikB;
    public B(int b,int a):base(a)
    {    OzellikB=b; }
}
class C:B
{    public int OzellikC;
    public C(int c,int b,int a):base(b,a)
    {    OzellikC=c; }
    static void Main()
    {    C nesne=new C(12,56,23);
        Console.WriteLine(nesne.OzellikA+"
"+nesne.OzellikB+" "+nesne.OzellikC);
    }
}
```

Örnekler

```
using System;
using System.Collections;
class Koleksiyon:IEnumerable
{ int[] Dizi;
  public Koleksiyon(int[] dizi)
  {   this.Dizi=dizi;   }
  IEnumerator IEnumerable.GetEnumerator()
  {   return new ENumaralandırma(this);   }
  class ENumaralandırma:IEnumerator
  { int indeks;
    Koleksiyon koleksiyon;
    public ENumaralandırma(Koleksiyon
koleksiyon)
    { this.koleksiyon=koleksiyon;
      indeks=-1;
    }
    public void Reset()
    {   indeks=-1;   }
```

```
public bool MoveNext()
{   indeks++;
  if(indeks<koleksiyon.Dizi.Length)
    return true;
  else    return false;
}
object IEnumerator.Current
{   get   { return(koleksiyon.Dizi[indeks]); }
}
}
class MainMetodu
{   static void Main()
  {   int[] dizi={1,2,3,8,6,9,7};
    Koleksiyon k=new Koleksiyon(dizi);
    foreach(int i in k)
      Console.Write(i+" ");
  }
}
```

Örnekler

Bu program kendi tanımladığımız herhangi bir sınıfın foreach döngüsüyle kullanılabilmesini sağlıyor. Yani foreach(tur1 nesne1 in nesne2){} deyiminde hem nesne1'in türünü (tur1) hem de nesne2'nin türünü kendi oluşturduğumuz türlerden (sınıf, yapı, vb.) biri yapabiliriz. Bu programda nesne2'nin kendi türümüzde olmasını sağlayacağız. Bu örnekten hemen sonraki örnekte de nesne1'in kendi türümüzden olmasını sağlayacağız. Programımıza dönecek olursak foreach döngüsünde nesne2'yi kendi türümüzden yapabilmemiz için bazı metotlar oluşturmamız gerekiyor. System.Collections isim alanındaki IEnumerable ve IEnumerator arayüzleri de bize bu metotları oluşturmaya zorluyor. Ayrıca programımızın ana hattına dikkat ederseniz iki sınıf iç içe geçmiş. C#'ta bu tür bir kullanım mümkündür. Bu programımızı satır satır incelemeye başlayalım.

```
using System;
using System.Collections;

//Bu satırlarla System ve System.Collections isim
alanlarındaki türlere direkt erişim hakkı elde ettik.

class Koleksiyon:IEnumerable {
    // Bu satırlarla System.Collections isim alanındaki
    IEnumerable arayüzünü kullanan bir sınıf başlattık.

    int[] Dizi;
    // Bu satırla sınıfımıza (Koleksiyon) ait bir özellik bildirdik.
    public Koleksiyon(int[] dizi)
    { this.Dizi=dizi; }

    // Bu satırlarla sınıfımızın (Koleksiyon) yapıcı metodunu
    bildirdik. Yapıcı metodumuz int[] türünden bir parametre
    alıyor ve bu aldığı veriyi sınıfın bir özelliği olan Dizi'ye
    aktarıyor. Burada this anahtar sözcüğünün kullanılması
    zorunlu değildir ancak okunurluğu artırır.
```

Örnekler

```
IEnumerator IEnumerable.GetEnumerator()  
{ return new ENumaralandırma(this); }
```

// Bu satırlarla sınıfımıza, IEnumerable arayüzündeki GetEnumerator() metodunu açık arayüz uygulama yöntemiyle geçiriyoruz. Metodun geri dönüş tipi IEnumerator arayüzü. Metodun gövdesinde ise ENumaralandırma sınıfının yapıcı metodu kullanılarak ENumaralandırma türünden bir nesne döndürülüyor. Buradan ENumaralandırma sınıfının yapıcı metodunun Koleksiyon sınıfı türünden bir nesne aldığını anlayabiliyoruz. Çünkü buradaki this anahtar sözcüğü bu GetEnumerator metoduna hangi nesne üzerinden erişildiğini temsil ediyor. Bu metot Koleksiyon sınıfında olduğuna göre bu metoda da bir Koleksiyon nesnesi üzerinden erişilmelidir. Burada bir terslik varmış gibi gözüküyor. O da metodun geri dönüş tipiyle geri döndürülülen verinin tipinin birbirine uymaması. Ancak programımızın sonraki kodlarına bakacak olursanız ENumaralandırma sınıfının IEnumerator arayüzünü kullandığını göreceksiniz. Dolayısıyla da bir IEnumerator tipinden nesneye bir ENumaralandırma nesnesi atanabilecek.

```
class ENumaralandırma:IEnumerator {  
    // Burada Koleksiyon sınıfının içinde ENumaralandırma  
    // sınıfını oluşturuyoruz ve bu sınıf da System.Collections  
    // isim alanındaki IEnumerator arayüzünü kullanıyor.  
  
    int indeks; Koleksiyon koleksiyon;  
    // Burada sınıfımıza (ENumaralandırma) iki tane özellik  
    // ekledik.  
  
    public ENumaralandırma(Koleksiyon koleksiyon)  
    { this.koleksiyon=koleksiyon; indeks=-1; }  
  
    // Burada ENumaralandırma sınıfının yapıcı metodunu  
    // hazırladık. Yeri gelmişken belirtmek istiyorum. İç içe  
    // sınıflar iç içe gözükselerde aslında birbirinden  
    // bağımsızdır. İç içe sınıfların normal sınıflardan tek farkı  
    // içteki sınıfa dıştaki sınıfın dışından erişilmek istendiğinde  
    // görülür. Bu durumda içteki sınıfa DisSinif.IcSinif yazarak  
    // erişilebilir. Ancak tabii ki bu erişimin mümkün olabilmesi  
    // için iç sınıfın public olarak belirtilmesi gerekir.
```


Örnekler

```
public void Reset() { indeks=-1; }
// Burada sınıfımıza bir metot ekledik. Bu metodu
sınıfımızın kullandığı IEnumerator arayüzü
gerektiriyordu.
public bool MoveNext()
{ indeks++; if(indeks<koleksiyon.Dizi.Length)
return true;
else return false; }
// Burada sınıfımıza bir metot daha ekledik. Yine bu
metodu da IEnumerator arayüzü gerektiriyordu.
object IEnumerator.Current
{
get { return(koleksiyon.Dizi[indeks]); }
}
// Burada IEnumerator arayüzünün gerektirdiği bir
sahte özelliği açık arayüz uygulama yöntemiyle
hazırladık.
class MainMetodu {
static void Main() {
// Artık programımızın çalışmaya başlayacağı kısmı
yazmaya başlıyoruz.
int[] dizi={1,2,3,8,6,9,7};
Koleksiyon k=new Koleksiyon(dizi);
// Dizimizi ve yeni bir Koleksiyon nesnesi oluşturduk.
```

```
foreach(int i in k) Console.Write(i+" ");
//Ve başardık. nesne2'yi kendi türümüz yaptık.
```

Şimdi nesne1'i kendi türümüz yapacak programı yazalım:

```
using System;
class A
{ public int Ozellik;
public A(int a) { Ozellik=a; }
}
class Ana {
static void Main()
{ A[] dizi=new A[3]; dizi[0]=new A(10);
dizi[1]=new A(20); dizi[2]=new A(50);
foreach(A i in dizi)
Console.WriteLine(i.Ozellik);
}
}
```

// Gördüğünüz gibi kendi oluşturduğumuz sınıf türünden nesnelerle diziler oluşturabiliyoruz. Nasıl ki int[] ile int türündeki veriler bir araya geliyorsa bizim örneğimizde de A türünden nesneler bir araya geldi.

Try-catch-throw Örnekler

- using System;
- class notlar {
- private int final;
- private int vize;
- public notlar(int f, int v)
- {
- if(v>100)
- { throw new hatalınot(v,"yapıcı metot"); }
- final=f;
- vize=v;
- }
- public void degerver(int f,int v)
- {
- if(v>100)
- throw new hatalınot (v,"degerver metodu");
- final=f;
- vize=v;
- }
- }

İstisnai Durum Yönetimi (Exception Handling)

```
• public int Final
•     {
•         get
•         { return final;}
•         set
•         { final=value;}
•     }
• public int Vize
•     {
•         get
•         { return vize;}
•         set
•         { if (value>100)
•             throw new hatalınot (value,"Vize Giriş Hatası");
•             vize=value;
•         }
•     }
• 
```

İstisnai Durum Yönetimi (Exception Handling)

- `public class hatalınot:ApplicationException`
- `{`
- `private int hatalıvize;`
- `private string hataKaynağı;`
- `public hatalınot(int hatalıvize, string hataKaynağı)`
- `{`
- `this.hataKaynağı=hataKaynağı;`
- `this.hatalıvize=hatalıvize;`
- `}`
- `public int Hatalıvize`
- `{`
- `get`
- `{ return hatalıvize;}`
- `}`
- `public string HataKaynağı`
- `{`
- `get`
- `{ return hataKaynağı; }`
- `}`

İstisnai Durum Yönetimi (Exception Handling)

- **public override string ToString()**
- { string str1="Hata Kaynağı: " + hataKaynagi+"\n";
- string str2="Hata Değeri: " + hatalivize;
- return str1+str2;
- }
- }
- }
- class hatatesti {
- public static void Main()
- { notlar a=new notlar(50,0);
- try
- { a.Vize=120; }
- catch(notlar.hatalinot e)
- { Console.WriteLine(e.ToString()); }
- }
- }



```
C:\Documents and Settings\SoNDuRaK\Project12\bin\Debug\P...  
Hata Kaynağı: Uize Giriş Hatası  
Hata Değeri: 120  
Press any key to continue
```

Örnekler

- **using** System;
 - **class** deneme
 - { **static void** Main()
 - { **for**(;;)
 - { **try**
 - { Console.**Write**("Lütfen çıkmak için 0 ya da 1 girin: ");
 - **string** a=Console.**ReadLine**();
 - **if**(a=="0" || a=="1") **break**;
 - **else**
 - **throw new** IndexOutOfRangeException("Devam ediliyor");
 - }
 - **catch**(IndexOutOfRangeException nesne)
 - { Console.**WriteLine**(nesne.**Message**); **continue**; }
 - } }
-
- Lütfen çıkmak için 0 ya da 1 girin: 5
 - Devam ediliyor
 - Lütfen çıkmak için 0 ya da 1 girin: 1

- **Başka bir yol ile**
- **using** System;
- **class** deneme
- { **static void** Main()
- { **for**(;;) {
- **try**
- { Console.**Write**("Lütfen çıkmak için 0 ya da 1 girin: ");
- **string** a=Console.**ReadLine**();
- **if**(a=="0" || a=="1") **break**;
- **else**
- { **IndexOutOfRangeException** nesne=new **IndexOutOfRangeException**("Başa dönüldü");
- nesne.**HelpLink**="http://tr.wikibooks.org";
- *//Gördüğünüz gibi bu yöntemle nesnenin özelliklerini değiştirebiliyoruz.*
- **throw nesne**; }
- }
- **catch**(IndexOutOfRangeException nesne) {
- Console.**WriteLine**(nesne.**Message**); **continue**; }
- }

Temsilciler-Örnek

- **using** System;
- **class** Temsilciler
- { **public delegate void** KomutMetodu();
//Geri dönüş tipi void olan ve parametre almayan bir temsilci bildirdik.
- **public struct** KomutYapisi { *//KomutYapisi isminde bir yapı başlattık.*
- **public** KomutMetodu KomutMetot;
//Yapımıza geri dönüş tipi KomutMetodu temsilcisi olan bir özellik ekledik.
- **public string** Komut; *//Yapımıza bir özellik daha ekledik. }*
- **public static void** Komut1() { *//Sınıfımıza bir metod ekledik.*
- Console.WriteLine("Komut1 çalıştı."); }
- **public static void** Komut2() { *//Sınıfımıza bir metod daha ekledik.*
- Console.WriteLine("Komut2 çalıştı."); }
- **public static void** Komut3() *//Sınıfımıza bir metod ekledik. {*
Console.WriteLine("Komut3 çalıştı."); }
- **public static void** Komut4() *//Sınıfımıza bir metod ekledik.*
- { Console.WriteLine("Komut4 çalıştı."); }
- **static void** Main()
- { KomutYapisi[] komutlar=new
KomutYapisi[4]; *//KomutYapisi nesnelerinden oluşan bir dizi oluşturduk. (4 elemanlı)*
- komutlar[0].Komut="komut1";
//komutlar[0] nesnesinin Komut özelliğine değer atadık.
komutlar[0].KomutMetot=new
KomutMetodu(Komut1); *//Artık komutlar[0] nesnesinin KomutMetot özelliği Komut1'i temsil ediyor. Aynı durumlar diğer komutlar için de geçerli.*

Temsilciler-Örnek

- `komutlar[1].Komut="komut2";`
`komutlar[1].KomutMetot=new`
`KomutMetodu(Komut2);`
`komutlar[2].Komut="komut3";`
`komutlar[2].KomutMetot=new`
`KomutMetodu(Komut3);`
`komutlar[3].Komut="komut4";`
`komutlar[3].KomutMetot=new`
`KomutMetodu(Komut4);`
`Console.Write("Komut girin: "); string`
`GirilenKomut=Console.ReadLine(); for(int`
`i=0;i<komutlar.Length;i++)//komutlar dizisi`
içinde dolaşmaya çıkıyoruz.
`if(GirilenKomut==komutlar[i].Komut)`
`komutlar[i].KomutMetot(); } }`

- Ekran Çıktısı:
- Komut girin: komut3
- Komut3 çalıştı.

Olaylar (Events)-Örnek

- **using** System;
- **delegate void** OlayYoneticisi(); *//Olay yöneticisi bildirimi*
- **class** AnaProgram {
- **static void** Main()
- {
- AnaProgram nesne=new AnaProgram();
- nesne.Olay+=new OlayYoneticisi(Metot); *//Olay sonrası işletilecek metotların eklenmesi*
- nesne.Olay(); *//Olayın gerçekleştirilmesi* }
- *//Olay sonrası işletilecek metot*
- **static void** Metot() { Console.WriteLine("Butona tıklandı."); }
- **event** OlayYoneticisi Olay; *//Olay bildirimi*
- }
- Burada bütün üye elemanlar aynı sınıfta bildirildi. Bu örneğimizde Olay olayını herhangi bir metot üzerinden değil, direkt çağırıldı.

Olaylar (Events)-Örnek

- using System;
- **public delegate int OlayYoneticisi();**
- interface IArayuz
- { int Metot1(); int Metot2();
- int sahteozeellik { set; get; }
- int this[int indeks] { get; }
- **event OlayYoneticisi Olay;**
- }
- class deneme:IArayuz
- { public int Metot1() { return 1; }
- public int Metot2() { return 2; }
- public int sahteozeellik
- { set{}
- get{return 3;}
- }

- public int this[int indeks]
- { get{return indeks;} }
- public event OlayYoneticisi Olay;
- static void Main() {
- deneme nesne=new deneme();
- **nesne.Olay+=new**
- **OlayYoneticisi(nesne.Metot1);**
- Console.WriteLine(nesne.Olay());
- }
- }

Örnekler

- using System;
- **delegate void OlayYoneticisi();**
- class Buton
- { OlayYoneticisi[] olay=new OlayYoneticisi[2];
//Olay yöneticimiz türünden iki elemanlı bir dizi oluşturduk.
- **public event OlayYoneticisi ButonKlik**
- { **add**
- { int i;
- for(i=0;i<2;++i)
- if(olay[i]==null)
- //Buradaki value olaya eklenen metottur.
- { olay[i]=value; break; }
- if(i==2) Console.WriteLine("Olaya en fazla iki metot eklenebilir.");
- }
- **remove**
- { int i;

- for(i=0;i<2;++i)
- if(olay[i]==value)
- { olay[i]=null; break;
- }
- if(i==2)
- Console.WriteLine("Metot bulunamadı");
- }
- }
- public void Kliklendi()
- {
- for(int i=0;i<2;++i)
- if(olay[i]!=null)
- olay[i]();
- }
- }
- class Pencere
- { int PencereNo;
- public Pencere(int no)
- { PencereNo=no; }

Örnekler

```
• public void ButonKlik()  
• { Console.WriteLine("{0} nolu pencere olayı  
algıladı.",PencereNo);  
• }  
• }  
• public class OlayTest  
• { static void Main()  
• {  
• Buton buton=new Buton();  
• Pencere p1=new Pencere(1);  
• Pencere p2=new Pencere(2);  
• //Geçerli ekleme:  
• buton.ButonKlik+=new OlayYoneticisi(ButonKlik);  
• buton.Kliklendi();  
• Console.WriteLine();  
• //Geçerli ekleme:  
• buton.ButonKlik+=new OlayYoneticisi(p1.ButonKlik);  
• buton.Kliklendi();  
• Console.WriteLine();
```

```
• //Geçersiz ekleme (Olay dolu):  
• buton.ButonKlik+=new OlayYoneticisi(p2.ButonKlik);  
• buton.Kliklendi();  
• Console.WriteLine();  
• buton.ButonKlik-=new OlayYoneticisi(p1.ButonKlik);  
• buton.Kliklendi();  
• Console.WriteLine();  
• buton.ButonKlik-=new OlayYoneticisi(ButonKlik);  
• buton.Kliklendi();  
• Console.WriteLine();  
• //Geçersiz çıkarma (metot yok):  
• buton.ButonKlik-=new OlayYoneticisi(ButonKlik);  
• buton.Kliklendi(); }  
• public static void ButonKlik()  
• { Console.WriteLine("Buton kliklendi"); }  
• }
```

Bu programın ekran çıktısı şöyle olmalıdır:

Buton kliklendi

Buton kliklendi 1 nolu pencere olayı algıladı.
Olaya en fazla iki metod eklenebilir.

Buton kliklendi 1 nolu pencere olayı algıladı.
Buton kliklendi Metod bulunamadı