4. Derleme zamanında hangi nesnelerin hangi fonksiyonlarının kullanılacağı belli olmayabilir. Aynı şekilde nesneler arası mesajlaşmanın hangi nesneler arasında olacağı da derleme zamanında belli olmayabilir. Bu durumda nesne yönelimli programlama tekniğinin çalışma zamanında bağlama mekanizmasından faydalanılır **(late binding)**.

5. Tüm nesneler birer sınıf örneğidir. Sınıflar nesnenin özelliklerini belirler. Nesneler çalışma zamanında oluşturulabileceği gibi derleme zamanında da oluşturulabilir.

6. Kalıtım yolu ile sınıflar birbirlerinden türetilebilir. Bir sınıf diğer bir sınıftan türediği zaman, türediği sınıfın bütün özelliklerini içerir. Bunun yanında kendine has özellikleri de barındırabilir. Nesne yönelimli programlama tekniğinin en önemli kavramı kalıtım yolu ile türetmedir. Kalıtım yolu ile türetilmiş sınıflar ile hiyerarşik bir sınıf organizasyonu gerçekleştirebiliriz. Bunun en güzel örneği .NET sınıf kütüphanesidir.

7. Nesne yönelimli programlama tekniğinde nesneler çok biçimli olabilir. Çok biçimlilik **(polymorphism)** kavramı da türetme ile yakından alakalıdır ve önemli bir kavramdır. Anlamı, bir nesnenin farklı şekillerde davranabilmesidir.

Diğer Programlama Teknikleri

Programlama teknikleri geçmişten günümüze çeşitli süreçlerden geçmişlerdir. Bu teknikleri kısaca sıralayalım.

1. Nesne Tabanlı Programlama (Object-based Programming)

Bu teknikte nesneler sıklıkla kullanılır. Veriler nesneler içinde toplanmıştır. Fakat bütün nesneler birbirlerinden bağımsızdır. Ayrıca sınıf, kalıtım ve geç bağlama gibi kavramlar bu teknikte kullanılmaz. ADA programlama dili bu tekniğe en iyi örnektir.

2. Sınıf Tabanlı Programlama (Class-based Programming)

Nesne tabanlı programlamaya ek olarak bu teknikte sınıf kavramı ve sınıf örnekleri **(instances)** kavramı yer alır. Nesne tabanlı programlamada olduğu gibi türetme ve çok biçimlilik bu teknikte de yoktur.

3. Nesne Yönelimli Programlama (Object Oriented Programming)

Sınıf tabanlı programlamanın bütün özelliklerinin yanında basit kalıtım kavramını da içerir. Smalltalk dili bu tekniğe örnek olarak verilebilir.

4. İleri Nesne Yönelimli Programlama (Advanced OOP)

Nesne yönelimli programlama tekniklerinin, bütün kavramları yanısıra çoklu türetme ve nesneler arasında ilişki belirleme gibi ileri teknikleri ile birlikte kullanıldığı programlama biçimidir. C++, JAVA ve C# dilleri bu tekniğe örnek olarak verilebilir.

Nesne Kavramı

Nesneler gerçek hayatta görebildiğimiz ya da varlığını bildiğimiz eşyalardır. Nesne yönelimli programlama tekniğinde sınıflar nesnelerin biçimlerini belirler. Oluşturduğumuz nesneler sınıf türlerinden nesneler olarak da adlandırılabilir. Her bir nesne kendi içinde tutarlı bir yapıya sahiptir. Yani veriler arasında sıkı bir bağ vardır. Eğer böyle olmasa nesne yönelimli programlama tekniklerini verimli bir şekilde kullanamazdık.

C#'ta nesne yönelimli programlama yapısını sınıflar sağlar. Sınıfları kullanarak nesneler tanımlarız. Hatırlayacağınız gibi bir nesne aşağıdaki gibi oluşturuluyordu.

```
using System;

class Sınıf
{
      public int a;

      public int A()
      {
          return a;
      }
}

class MainMetodu
{
      static void Main()
      {
              Sınıf x = new Sınıf();
      }
}
```

Yukarıdaki programda Sınıf bir nesnenin şeklini belirlemektedir. Kısacası bir tür bilgisi saklamaktadır. **new** anahtar sözcüğünü kullanarak bu türden x isimli bir nesne tanımlıyoruz. Bu x nesnesinin bir tane veri taşıyan yapısı, bir tane de iş yapan metodu mevcuttur.

Sınıflar bir tür bilgisi olduğuna göre bu türü paketleyip istediğimiz programda kullanabilmemiz gerekir.

Sınıf Kütüphanesi Oluşturma

Türetme ve diğer önemli konuları incelemeye başlamadan önce C# ile bağımsız bir sınıfı nasıl oluşturabileceğimizi ve nasıl derleyebileceğimizi inceleyeceğiz. Ardından bu sınıfı istediğimiz programlarda nasıl kullanacağımızı göreceğiz.

Her zaman olduğu gibi boş bir metin dosyası açıp aşağıdaki sınıfı yazalım ve RastgeleSayi.cs adıyla kaydedelim. Bu sınıf yapısında rastgele bir değer bulunan bir özelliğe sahiptir. Ayrıca rastgele sayıyı daha sonra değiştirebilmek için bir de Degistir() isimli bir metot sağlanmıştır. Bu örnekte rastgele sayılar üretmek için System isim alanında bulunan Random sınıfı kullanılmıştır.

```
using System;

namespace CSharp
{
      public class RastgeleSayi
      {
          private Random rnd = new Random();
          private int mSayi;

          public int Sayi
          {
              get
              {
                      return mSayi;
              }
          }
          public RastgeleSayi()
          {
              mSayi = rnd.Next(0,100);
          }

          public void Degistir()
          {
              mSayi = rnd.Next(0,100);
          }
      }
}
```

Yukarıdaki kaynak kodda Main() metodunun olmadığına dikkat edin. Burada tanımladığımız sınıf kendi başına çalışabilen bir yapı değildir. Ancak bir program içinde bu sınıf türünden nesneler tanımlayabiliriz.

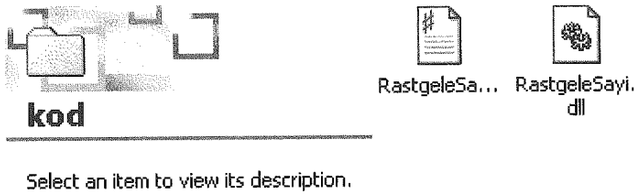
Şimdi bu sınıfın kaynak kodlarını açıkça göremeyecek şekilde, .NET platformunu destekleyen bütün dillerde kullanılmak üzere derleyeceğiz.

Komut satırında ilgili klasöre geldikten sonra aşağıdaki komutu yazın.

```
csc /t:library RastgeleSayi.cs
```

/t:library parametresi ile derleyiciye RastgeleSayi içindeki kodların bir sınıf kütüphanesi olacak şekilde derlenmesi gerektiği söyleniyor. Bu şekilde, derleyici programların başlangıç noktası olan Main() metodunu aramaz.

Program derlendikten sonra RastgeleSayi.cs dosyasının bulunduğu klasörde aşağıdaki gibi, RastgeleSayi.dll isminde bir dosyanın oluşturulduğunu görürüz.



Bu dosya da RastgeleSayi sınıfını içeren binary düzeyde bilgiler bulunur. Dosyayı açtığımızda kodlarımızı açık bir şekilde göremeyiz.

RastgeleSayi.dll dosyası elimizde olduğu sürece programlarımızda RastgeleSayi sınıfını dilediğimiz gibi kullanabiliriz.

Dosya ismi ile kaynak kodda bulunan sınıfın ismi JAVA dilinde olduğu gibi aynı olmak zorunda değildir.

Sınıf Kütüphanesini Kullanma

Oluşturduğumuz bu sınıfı programlarımızda kullanabilmek için derleyiciye dll dosyasının yolunu bildirmek gerekir. RastgeleSayi.cs dosyasının bulunduğu klasörde Program.cs isimli bir dosya açın ve aşağıdaki programı yazın.

```
using System;

class MainMetodu
{
      static void Main()
      {
          CSharp.RastgeleSayi r = new CSharp.RastgeleSayi();
          Console.WriteLine(r.Sayi);

          r.Degistir();
          Console.WriteLine(r.Sayi);
      }
}
```

Şimdi Program.cs dosyasını derlemek için komut satırına aşağıdaki komutu yazın.

```
csc /r:RastgeleSayi.dll Program.cs
```

"/r:RastgeleSayi.dll" parametresi derleyiciye RastgeleSayi.dll dosyasını tanıtmaktadır. Yani programda kullanılan türler bu dosya içinde de aranacaktır. Eğer bu parametreyi belirtmezsek RastgeleSayi sınıfı bulunmayacağı için, kod derlenemeyecektir.

Eğer dll dosyasında sınıfın bildirildiği isim alanını programa **using** anahtar sözcüğü ile dahil edersek sınıfa aşağıdaki gibi direkt olarak da erişebiliriz.

```
using System;
using CSharp;

class MainMetodu
{
      static void Main()
      {
          RastgeleSayi r = new RastgeleSayi();
          Console.WriteLine(r.Sayi);

          r.Degistir();
          Console.WriteLine(r.Sayi);
      }
}
```

Kalıtım (Inheritance)

Kalıtım nesne yönelimi programlama tekniğinin en önemli özelliğidir. Kalıtım yolu ile nesneler birbirlerinden türetilir. Türeyen sınıflar türediği sınıfın özelliklerini kalıtım yolu ile devraldığı gibi kendisi de yeni özellikler tanımlayabilir. Türetme yolu ile bir nesne kendi özelliklerini yeni nesneye aktarır. İstenirse türeyen nesne kendisine has özel özellikler de içerebilir. Türetme ile sınıflar arasında hiyerarşik bir yapı sağlanır. Örneğin bir önceki konuda gördüğümüz Stream sınıfı bir akımın sahip olabileceği minimum özelliklere sahiptir. FileStream ve NetworkStream sınıfları ise Stream sınıfından türeyerek özelleştirilmiştir. NetworkStream ve FileStream sınıfları Stream sınıfının bütün özelliklerini kalıtım yolu ile aldıkları için her iki sınıfta da ortak bir takım özellikler vardır; ancak her iki sınıfta kendi içinde kendi işlerine yönelik özel özellikler de barındırırlar. Örneğin Stream sınıfında bulunan ReadByte() metodu akımdan bir bytelık bilgi okur, aynı şekilde bu metot FileStream ve NetworkStream sınıflarında da bulunur. Bu metot kalıtım yolu ile bu sınıflara geçmiştir. Ancak her akımdan bir byte'lık bilgi farklı şekillerde elde edildiği için her sınıf kendi içinde bu metodu yeniden yazmış olabilir. Ama bu bizim için önemli değil; önemli olan arayüzün aynı olması ve her iki durumda da ilgili akımdan bir byte'lık bilginin okunmasıdır.

Sınıfları türetmeyi daha iyi anlayabilmek için şöyle bir örnek verelim: Memeliler diye bir sınıf tasarlamış olalım. Kedi, köpek, inek gibi bütün hayvanlar memeli grubuna

girmektedir. Ancak her hayvanın kendisine has özellikleri vardır. Bu hayvanları Memeli olarak tanımlayabilsek de kedi ve köpek arasındaki tek benzerlik memeli olması da olabilir. Bu durumda her hayvanı farklı bir sınıf ile ifade ederek sorunu çözebiliriz. Ancak her sınıfta memeli hayvanların ortak özelliklerini tek tek tanımlamak zor olur. Bunun yerine memeli hayvanların ortak olan özelliklerini Memeli isimli bir sınıfta toplamalıyız. Bu sınıfta her memeli hayvanda bulunması gereken özellikler tanımlanır. Diğer Kedi, Kopek gibi sınıflar ise Memeli sınıfının bir devamı olarak tasarlanır. Her özel sınıf Memeli sınıfına ait özellikleri ister gizler isterse tekrar kullanır. İsterse de memeli sınıfının özelllikleri dışında kendine yeni özellik ekler.

Gördüğünüz gibi Memeli sınıfını bir kere tanımlayarak kalıtım yolu ile diğer özellştirilmiş sınıfları Memeli sınıfından türeterek tasarlayabiliriz. Memeli sınıfının özelliklerini kalıtım yolu ile Kedi sınıfına aktarmaya kalıtım yolu ile türetme denilmektedir.

Bu giriş bilgilerinden sonra C# dilinde kalıtım yolu ile türetmenin nasıl yapıldığını ve temel kavramları inceleyelim.

Türetmenin Yapılması ve Temel Kavramlar

Şimdi temel bir Memeli sınıfı tasarlayacağız. Her memeli hayvanda bulunan özellikler boy, ağırlık gibi fiziksel özellikler olabilir. Daha sonra Kedi sınıfını tasarlayacağız. Ancak Kedi sınıfını Memeli sınıfından türeteceğiz. Yani Memeli sınıfının tüm özelliklerini Kedi sınıfına aktaracağız. Memeli sınıfında boy ve ağırlık bilgilerini gösteren OzellikGoster() isimli bir metot olacak. Kedi sınıfında da hayvanın kedi olduğunu gösteren string türünden bir özellik olacak. Ayrıca Kedi sınıfında kedinin türünü yazdıracak bir de metot olmalıdır.

Türetme işlemi aşağıdaki gibi yapılır.

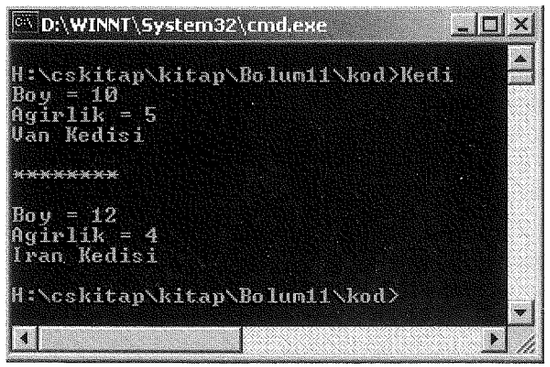
```
class Kedi : Memeli
{

}
```

Burada Kedi türemiş sınıf (derived class) Memeli ise temel sınıf (base class) olarak adlandırılır. Şimdi bu sınıfların nasıl yazıldığına bakalım.

```
using System;

class Memeli
{
      public double Boy;
      public double Agirlik;

      public void OzellikGoster()
```

```
        {
            Console.WriteLine("Boy = " + Boy);
            Console.WriteLine("Ağırlık = " + Agirlik);
        }
}

class Kedi : Memeli
{
        public string Turu;

        public void TurGoster()
        {
            Console.WriteLine(Turu + " Kedisi");
        }
}

class MainMetodu
{
        static void Main()
        {
            Kedi kedi1 = new Kedi();
            kedi1.Agirlik = 5;
            kedi1.Boy = 10;
            kedi1.Turu = "Van";

            kedi1.OzellikGoster();
            kedi1.TurGoster();

            Console.WriteLine("\n********\n");

            Kedi kedi2 = new Kedi();
            kedi2.Agirlik = 4;
            kedi2.Boy = 12;
            kedi2.Turu = "İran";

            kedi2.OzellikGoster();
            kedi2.TurGoster();
        }
}
```

İlk işimiz her zaman olduğu gibi Programın ekran çıktısına bakmaktır.



Gördüğünüz gibi Kedi sınıfı üzerinden Memeli sınıfına ait bütün özelliklere ve metotlara erişebiliyoruz. Bunu sağlayan elbette ki

```
class Kedi : Memeli
{

}
```

satırlarıdır.

Şimdi bir de aşağıdaki gibi Memeli sınıfı türünden bir nesne tanımlayalım.

```
class MainMetodu
{
        static void Main()
        {
            Memeli memeli = new Memeli();

            memeli.Agirlik = 10;
            memeli.Boy = 15;

            memeli.OzellikGoster();

            //Hatalı Kullanım!
            memeli.TurGoster();
        }
}
```

Programı derlemeye çalıştırdığımızda hata mesajı alırız; çünkü memeli nesnesi üzerinden TurGoster() metoduna erişmemiz mümkün değildir. Yani temel sınıf olan Memeli sınıfının kendisinden türeyen Kedi sınıfındaki üye elemanlardan haberi yoktur. Ancak tersi doğru değildir.

Protected Erişim Belirleyicisi

Sınıflar konusunda erişim belirleyicilerini incelerken, protected erişim belirleyicisini türetmeyi anlatırken inceleyeceğimizi söylemiştik. Kalıtım yolu ile sadece public ve protected elemanlar aktarılır. Temel sınıfta bulunan private üye elemanlara türeyen sınıf içinde ulaşamayız. Örneğin aşağıdaki kullanım geçersizdir.

```
using System;

class Memeli
{
        private double Boy;
        private double Agirlik;

        public void OzellikGoster()
        {
```

```
            Console.WriteLine("Boy = " + Boy);
            Console.WriteLine("Ağırlık = " + Agirlik);
        }
}

class Kedi : Memeli
{
        public string Turu;

        public void BoyGoster()
        {
            Console.WriteLine("Boy = " + Boy);
        }
}
```

Burada Kedi sınıfı Memeli sınıfında türemiş olmasına rağmen Kedi sınıfında bulunan BoyGoster() metodu içerisinden Memeli sınıfındaki Boy özelliğine erişemiyoruz. Bunun sebebi Memeli sınıfındaki Boy özelliğinin private olarak bildirilmesidir. private özelliklerinde kalıtım yolu ile aktarılması için her değişken için get ve set blokları tanımlanabilir. Bu durumda Memeli sınıfının private elemanlarına erişebiliriz.

Diğer sınıfların kullanımına kapalı ancak türetme ile türemiş sınıfa geçebilen özellikler protected anahtar sözcüğü kullanılarak oluşturulur. Eğer türeme söz konusu değilse protected olarak bildirilen elemanlarla private olarak bildirilen elemanlar arasında bir fark kalmaz. **protected** eleman içeren bir sınıf türetildiğinde bu elemanlar türemiş sınıflar içinde rahatlıkla kullanılabilir. **protected** elemanlar diğer sınıflar için private olarak kalmaya devam eder. Buna göre aşağıdaki sınıf bildirimleri geçerlidir.

```
using System;

class Memeli
{
        protected double Boy;
        protected double Agirlik;
        public void OzellikGoster()
        {
            Console.WriteLine("Boy = " + Boy);
            Console.WriteLine("Ağırlık = " + Agirlik);
        }
}

class Kedi : Memeli
{
        public string Turu;

        public void BoyGoster()
        {
            Console.WriteLine("Boy = " + Boy);
        }
}
```


Memeli sınıfındaki Boy ve Agirlik özellikleri protected olduğu için Main() metodu içinde aşağıdaki gibi bir kullanım geçersizdir.

```
class MainMetodu
{
        static void Main()
        {
            Kedi kedi = new Kedi();

            kedi.Boy = 10;

            Memeli memeli = new Memeli();

            memeli.Boy = 10;
        }
}
```

Görüldüğü gibi ne kedi nesnesi üzerinden ne de memeli nesnesi üzerinden Boy değişkeninin değeri değiştirilememektedir. Boy özelliğine sadece Kedi ve Memeli sınıflarının bildirildiği blok içerisinden erişebiliriz.

Yapıcı Metotların Kalıtımdaki Rolü

Sınıflar türetilirken temel sınıftaki bütün elemanların türemiş sınıfa dahil edildiğini söylemiştik. Bu elemanlara metotlar da dahildir. Ancak metotların içinde özel bir durumu olan yapıcı metotların ilginç bir yanı vardır. Bir Kedi nesnesi tanımlandığında Kedi sınıfının yapıcı metodunun çağrılacağı kesindir. Peki, Memeli sınıfının yapıcı metodu da çağrılacak mıdır? Bunu öğrenmek için Kedi ve Memeli sınıfını aşağıdaki gibi değiştirdikten sonra programı derleyip ekran çıktısına bakalım.

```
using System;

class Memeli
{
        protected double Boy;
        protected double Agirlik;

        public Memeli()
        {
            Console.WriteLine("Memeli hayvan oluşturuldu");
        }

        public void OzellikGoster()
        {
            Console.WriteLine("Boy = " + Boy);
            Console.WriteLine("Ağırlık = " + Agirlik);
        }
}
```

```
class Kedi : Memeli
{
      public string Turu;

      public Kedi()
      {
          Console.WriteLine("Kedi oluşturuldu");
      }

      public void TurGoster()
      {
          Console.WriteLine(Turu + " Kedisi");
      }
}

class MainMetodu
{
      static void Main()
      {
          Kedi kedi = new Kedi();
      }
}
```

Programın çıktısı aşağıdaki gibi olacaktır.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum11\kod>Yapicilar
Memeli hayvan olusturuldu
Kedi olusturuldu

H:\cskitap\kitap\Bolum11\kod>
```

Görüldüğü gibi hem Kedi hem de Memeli sınıfının yapıcı metodu çağrılmıştır. Ancak ilk çağrılan yapıcı metot Memeli sınıfının yapıcı metodu olmuştur. Eğer Memeli sınıfında yapıcı metot olmasaydı doğal olarak sadece Kedi sınıfının yapıcı metodu çağrılırdı. Yapıcı metotlarla ilgili en önemli konu her yapıcı metodun kendi tanımlandığı sınıfın üye elemanları ile ilgili iş yapması gerekliliğidir. Bu dilin bir zorunluluğu değildir, nesne yönelimli programlama tekniğini daha etkin kullanabilmek için gereklidir.

Kedi sınıfının aşağıdaki gibi bir yapıcı metodunun olduğunu düşünün.

```
public Kedi(string Tur, int Boy, int Agirlik)
{
}
```

Kedi sınıfının Boy ve Agirlik özelliklerinin ilk değer atamasını yaparken, Kedi sınıfının yapıcı metodu mu yoksa Memeli sınıfının yapıcı metodu mu kullanılmalıdır?

Kuşkusuz her ikisi de mümkündür. Ancak eğer ilk değer verme işlemini türeyen sınıfta yaparsak

```
Memeli memeli = new Memeli(5,10);
```

şeklinde bir nesne tanımlaması için temel sınıfta ayrıca bir yapıcı metodun daha tanımlanması gerekir. Bu yüzden her elemanın ilk değer ataması, tanımlandığı sınıfın yapıcı metodu ile yapılmalıdır. Yani Boy ve Agirlik özellikleri Memeli sınıfının yapıcı metodu ile, Turu özelliği ise Kedi sınıfının yapıcı metodu ile düzenlenmelidir. Bu durumda önümüze bir engel çıkmaktadır.

```
Kedi kedi = new Kedi("Van",5,10);
```

şeklinde bir kedi nesnesi tanımladığımızda 5 ve 10 değerlerini Memeli sınıfının yapıcı metoduna nasıl aktaracağız? Bunun için türemiş sınıf içerisinden temel sınıfın elemanlarına erişmek için kullanılan **base** anahtar sözcüğünü kullanacağız.

Buna göre kedi sınıfının yapıcı metodunu şu şekilde düzenlemeliyiz.

```
public Kedi(string turu, int boy,int agirlik):base(boy,agirlik)
{
      this.Turu = turu;
}
```

Kedi sınıfının yapıcı metodu çağrıldığı anda **base** anahtar sözcüğü ile kedi nesnesi yapılandırmak için gelen boy ve agirlik değişkenleri Memeli sınıfının yapıcı metoduna gönderiliyor. Hatırlarsanız sınıflar konusunda yapıcı metotların bu şekilde kullanıldığını görmüştük.

Aşağıda **base** anahtar sözcüğünün kullanımına Memeli ve Kedi sınıfları kullanılarak tam bir örnek verilmiştir.

```
using System;

class Memeli
{
      protected double Boy;
      protected double Agirlik;

      public Memeli(double boy,double agirlik)
      {
          this.Boy = boy;
          this.Agirlik = agirlik;
      }

      public void OzellikGoster()
      {
          Console.WriteLine("Boy = " + Boy);
```

```
            Console.WriteLine("Ağırlık = " + Agirlik);
        }
}

class Kedi : Memeli
{
        public string Turu;

        public Kedi(string turu, int boy,int agirlik):
                                base(boy,agirlik)
        {
            this.Turu = turu;
        }

        public void TurGoster()
        {
            Console.WriteLine(Turu + " Kedisi");
        }
}

class MainMetodu
{
        static void Main()
        {
            Kedi kedi = new Kedi("Van",5,10);
            kedi.OzellikGoster();
            kedi.TurGoster();

            Console.WriteLine("**********");

            Memeli memeli = new Memeli(6,11);
            memeli.OzellikGoster();
        }
}
```

Bu programı derleyip çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum11\kod>Base
Boy = 5
Agirlik = 10
Van Kedisi
**********
Boy = 6
Agirlik = 11

H:\cskitap\kitap\Bolum11\kod>
```

Görüldüğü gibi

```
base(boy,agirlik)
```

ifadesi ile çağrılan aslında Memeli sınıfının yapıcı metodudur.

Şimdi de birden fazla türetmenin olduğu sınıf hiyerarşilerini inceleyelim. Örneğin A sınıfı, B sınıfından; B sınıfı da C sınıfından türemiş olsun. C sınıfı türünden bir nesne tanımlandığında hangi sınıfların yapıcı metotlarının çağırıldığını görmek için aşağıdaki programı yazın.

```
using System;
vclass A
{
        public int a;
        public A()
        {
            Console.WriteLine("A sınıfının yapıcı metodu");
        }
}

class B : A
{
        public int b;

        public B()
        {
            Console.WriteLine("B sınıfının yapıcı metodu");
        }
}

class C : B
{
        public int c;

        public C()
        {
            Console.WriteLine("C sınıfının yapıcı metodu");
        }
}

class MainMetodu
{
        static void Main()
        {
            C c = new C();
        }
}
```

Bu programı derleyip çalıştırdığımızda ekrana

A sınıfının yapıcı metodu
B sınıfının yapıcı metodu
C sınıfının yapıcı metodu

yazacaktır.

Gördüğünüz gibi ilk olarak sınıf hiyerarşisinin en tepesinde bulunan A sınıfının yapıcı metodu çağrılmıştır. En son çağrılan yapıcı metot ise sınıf hiyerşisinin en altında bulunan sınıfın yapıcı metodudur.

Yukarıdaki A, B ve C sınıf hiyerarşisinde base anahtar sözcüğünün kullanımı ise aşağıdaki gibidir.

```
using System;

class A
{
      public int a;

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

class B : A
{
      public int b;

      public B(int a,int b) : base(a)
      {
          this.b = b;
      }
}

class C : B
{
      public int c;

      public C(int a,int b, int c) : base(a,b)
      {
          this.c = c;
      }
}
class MainMetodu
{
      static void Main()
      {
          C c = new C(5,6,7);
          Console.WriteLine("** C nesnesi **");
          Console.WriteLine("a=" + c.a);
          Console.WriteLine("b=" + c.b);
          Console.WriteLine("c=" + c.c);

          B b = new B(8,9);
          Console.WriteLine("** B nesnesi **");
          Console.WriteLine("a=" + b.a);
          Console.WriteLine("b=" + b.b);

          A a = new A(10);
          Console.WriteLine("** A nesnesi **");
          Console.WriteLine("a=" + a.a);
      }
}
```


Bu programda vurgulamak istediğimiz nokta, **base** anahtar söcüğünün sınıf hiyerarşisinin en tepesindeki sınıfı temsil etmediğidir. Yani C sınıfında **base** anahtar sözcüğü B sınıfı anlamına gelirken, B sınıfında **base** anahtar sözcüğü A sınıfı anlamına gelir.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum11\kod>ABC
** C nesnesi **
a=5
b=6
c=7
** B nesnesi **
a=8
b=9
** A nesnesi **
a=10

H:\cskitap\kitap\Bolum11\kod>
```

Bu programda bildirilen C sınıfı hem A sınıfının hem de B sınıfının bütün özelliklerini içermektedir.

İsim Saklama (Name Hiding)

Türemiş sınıfta, bazen temel sınıftaki bir üye elemanla aynı isimli bir eleman tanımlamak isteyebiliriz. Normal şartlarda türeyen sınıfta, temel sınıfta bulunan bir üye elemanının isminde bir eleman tanımlanabilir. Bu durumda temel sınıftaki üye elemana normal bir şekilde erişmek mümkün değildir. Çünkü türemiş sınıfta bulunan eleman temel sınıftaki elemanı gizlemiştir. İsim gizleme ile ilgili aşağıdaki örneği inceleyin.

```
using System;

class X
{
      protected int a;

      public X(int a)
      {
          this.a = a;
      }

      public X()
      {
      }

      public int A
      {
          get
          {
              Console.Write("X sınıfı");
              return a;
          }
      }
}
```

```
class Y : X
{
      protected int b;


      public Y(int a)
      {
          this.b = a;
      }

      public Y()
      {
      }

      public int A
      {
          get
          {
              Console.Write("Y sınıfı");
              return b;
          }
      }
}


class MainMetodu
{
      static void Main()
      {
          Y y = new Y(5);

          int deneme = y.A;
      }
}
```

Bu programı derlediğimizde aşağıdaki gibi bir uyarı ile karşılaşırız.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum11\kod>csc IsimGizleme.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

IsimGizleme.cs(40,13): warning CS0108: The keyword new is required on 'Y.A'
        because it hides inherited member 'X.A'
IsimGizleme.cs(16,13): (Location of symbol related to previous warning)

H:\cskitap\kitap\Bolum11\kod>
```

Bu uyarıda Y.A elemanı X.A elemanını gizlediği söylenmiştir. Bu yüzden **new** anahtar sözcüğünün kullanılmasını tavsiye ediyor. Buna rağmen programın derlendiğine dikkat edin. Derleyici bizim isim gizlemesini yanlışlıkla yaptığımızı düşünüp uyarı ver-

miştir. Ancak biz her zaman isim gizlemesi yaparken isim gizlemeyi açıkça belirtmek için temel sınıftaki bir elemanı gizlerken türemiş sınıftaki elemanın bildirimine **new** anahtar sözcüğünü eklemeliyiz. Aşağıda **new** anahtar sözcüğünün isim gizlemek için nasıl kullanıldığını görebilirsiniz.

```
class Temel
{
      public int a;
}

class Tureyen : Temel
{
      new int a;
}
```

Bu iki sınıfı içeren bir kaynak kod artık uyarı vermeden derlenecektir. İsim gizleme işlemini metotlar ve diğer üye yapılar içinde yapmak mümkündür.

Peki, isim gizlemesinin yapıldığı durumlarda temel sınıftaki üye elemana ulaşabilir miyiz? Evet, ulaşabiliriz. Bunun için **base** anahtar sözcüğü kullanılır.

Örneğin:

```
class Temel
{
      public int a;
}

class Tureyen : Temel
{
      new int a;
      base.a = 5;
}
```

sınıflarında

```
base.a = 5;
```

deyimiyle Temel sınıftaki a değişkenine erişilmiş oluyor. **base** anahtar sözcüğünün bu şekildeki kullanımı sınıflardaki this referansına benzemektedir. this referansı kendisini çağıran sınıfı temsil ederken **base** anahtar sözcüğü türetmenin yapıldığı temel sınıfı temsil eder. Aşağıda Tureyen sınıf tarafından gizlenmiş olan üye değişkenlerin ve metotların **base** anahtar sözcüğü ile nasıl kullanıldıklarına bir örnek görüyorsunuz. Tabii gizlenen her eleman için derleyicinin uyarı vermemesi için **new** anahtar sözcüğü kullanılıyor.

```
using System;

class Temel
{
      public int a;

      public void Goster()
      {
          Console.WriteLine("Temal.a = " + a);
      }
}

class Tureyen : Temel
{
      new int a;

      public Tureyen(int aTemel, int aTureyen)
      {
          a = aTureyen;
          base.a = aTemel;
      }

      new public void Goster()
      {
          base.Goster();
          Console.WriteLine("Türeyen.a = " + a);
      }
}


class MainMetodu
{
      static void Main()
      {
          Tureyen nesne = new Tureyen(5,6);

          nesne.Goster();
      }
}
```

Program derlendiğinde ekrana

Temel.a = 5
Türeyen.a = 6

yazacaktır.

Hem Temel hem de Tureyen sınıftaki Goster() metotları public olarak tanımlandığı için normal şartlarda her ikisine de erişmemiz mümkündür ancak isim benzerliğinden dolayı


```
Tureyen nesne = new Tureyen(5,6);
nesne.Goster();
```

deyimleri ile Tureyen sınıftaki Goster() metodu çağrılmaktadır.

Tureyen sınıftaki Goster() metodunu

```
new private void Goster()
{
      base.Goster();
      Console.WriteLine("Türeyen.a = " + a);
}
```

gibi public yerine private olarak bildirirsek;

```
Tureyen nesne = new Tureyen(5,6);
nesne.Goster();
```

deyimleri ile Temel sınıfın Goster() metodu çağrılacaktır. Çünkü nesne üzerinden Tureyen sınıfın private olan Goster() metoduna erişilemez.

Temel ve Türeyen Sınıf Nesneleri

C# dili tür güvenliğine (type safety) büyük önem vermektedir. Bu yüzden farklı türlerdeki nesnelerin birbirlerine özel durumların dışında atanması yasaklanmıştır. Bunun bir istisnası da temel ve türeyen sınıflarda görülmektedir. Türeyen sınıfa ilişkin bir referans temel sınıfa ilişkin bir referansa atanabilir. Bu son derece mantıklı bir yaklaşımdır. Örneğin bölümün başlarında el aldığımız Memeli ve Kedi sınıfını düşünün. Her Kedi aslında bir Memeli hayvan olduğu için Kedi nesnelerini Memeli nesnelerine atayabilmemiz son derece olağan bir durumdur.

Şimdi temel ve türeyen sınıfların birbirlerine atanmaları sonucunda oluşabilecek durumları inceleyelim. Öncelikle aşağıdaki programı yazalım.

```
using System;

class Memeli
{
      protected double Boy;
      protected double Agirlik;

      public Memeli(double boy,double agirlik)
      {
          this.Boy = boy;
          this.Agirlik = agirlik;
      }

      public void OzellikGoster()
      {
```

```
        Console.WriteLine("Boy = " + Boy);
        Console.WriteLine("Ağırlık = " + Agirlik);
    }
}

class Kedi : Memeli
{
    public string Turu;

    public Kedi(string turu, int boy,int agirlik):
                        base(boy,agirlik)
    {
        this.Turu = turu;
    }

    public void TurGoster()
    {
        Console.WriteLine(Turu + " Kedisi");
    }
}

class MainMetodu
{
    static void Main()
    {
        Kedi kedi1 = new Kedi("Van",5,10);
        Kedi kedi2;

        //Normal atama
        kedi2 = kedi1;

        kedi2.OzellikGoster();
        kedi2.TurGoster();

        Memeli memeli = new Memeli(6,10);

        memeli = kedi1;
        memeli.OzellikGoster();

        //Hatalı kullanım..
        memeli.TurGoster();
    }
}
```

Bu programda Memeli sınıfından Türeyen Kedi sınıfı türünden nesneleri Memeli türünden nesne referanslarına atayabildik. Bu tür bir atamada memeli referansı üzerinden artık kedi1 nesnesinin bütün özelliklerine erişmek mümkün değildir. Çünkü Memeli sınıfı, Kedi sınıfının üye elemanlarından haberdar değildir. Böylece memeli referans değişkenine kedi1 nesnesi atanmış olmasına rağman memeli nesnesi üze-

rinden sadece Memeli sınıfının üye elemanlarına ulaşabiliriz. Örneğin yukarıdaki memeli nesnesi üzerinden TurGoster() metoduna erişmeye çalışmak derleme zamanında hataya yol açacaktır.

Temel ve Tureyen sınıflar arasındaki bu ilişki nesne yönelimli programlama tekniğinin en önemli kavramlarından biri olan çok biçimliliğin altyapısını oluşturur. Çok biçimliliğin diğer ayrıntılarını inceleyeceğiz ancak önce Temel ve Türeyen sınıflar arasındaki bu ilişkiyi güzel bir örnekle açıklığa kavuşturalım.

Şimdi Memeli sınıfından yeni bir sınıf daha türeteceğiz. Bu sınıf da yine bir memeli hayvan olan Koyun'u temsil edecek. Koyun sınıfı da, Memeli sınıfından türeyeceği için bir memeli hayvanın sahip olacağı ortak özelliklere sahip olacaktır. Ancak kendine has özellikleri de olabilecektir. Bizim için asıl önemli olan nokta ise parametre olarak Memeli türünden bir referans alan global bir metodun Kedi ve Koyun nesnelerini nasıl parametre olarak kabul edeceğidir.

Aşağıda Memeli, Kedi, Koyun sınıfları ile bu sınıfları test edebileceğimiz bir Main() metodu tanımlanmıştır. Önce bu programı inceleyelim, ardından nelerin olup bittiğine bakalım.

```
using System;

class Memeli
{
    protected double mBoy;
    protected double mAgirlik;

    public Memeli(double boy,double agirlik)
    {
        this.mBoy = boy;
        this.mAgirlik = agirlik;
    }
    public void OzellikGoster()
    {
        Console.WriteLine("Boy = " + mBoy);
        Console.WriteLine("Ağırlık = " + mAgirlik);
    }

    public double Boy
    {
        get{ return mBoy;}
        set{ mBoy = value;}
    }

    public double Agirlik
    {
        get{ return mAgirlik;}
        set{ mAgirlik = value;}
    }
}
```

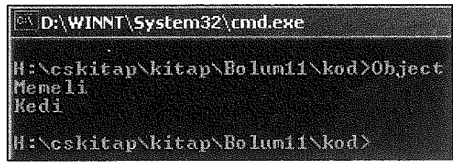
```
class Kedi : Memeli
{
      public string Turu;

      public Kedi(string turu, int boy,int agirlik):
                        base(boy,agirlik)
      {
          this.Turu = turu;
      }

      public void TurGoster()
      {
          Console.WriteLine(Turu + " Kedisi");
      }
}

class Koyun : Memeli
{
      public string Turu;
      public Koyun(string turu, int boy,int agirlik) :
                                base(boy,agirlik)
      {
          this.Turu = turu;
      }

      public void TurGoster()
      {
          Console.WriteLine(Turu + " Koyunu");
      }
}

class MainMetodu
{
      public static void Goster(Memeli memeli)
      {
          Console.WriteLine("Boy = " + memeli.Boy);
          Console.WriteLine("Ağırlık = " + memeli.
                                  Agirlik + "\n");

      }

      static void Main()
      {
          Memeli memeli1 = new Memeli(10,35);
          Goster(memeli1);

          Kedi kedi1 = new Kedi("Van", 5 , 10);
          Goster(kedi1);

          Koyun koyun1 = new Koyun("Ankara",52,70);
          Goster(koyun1);
      }
}
```


Görüldüğü gibi MainMetodu sınıfı içinde statik olarak tanımlanan Goster() metodu, parametre olarak Memeli sınıfı türünden bir referans beklemektedir. Kedi ve Koyun da birer memeli hayvan olduğu için parametre olarak Kedi ve Koyun sınıfı türünden referansları metoda parametre olarak gönderebiliyoruz. Ancak metodun içinde memeli nesnesinde Kedi ya da Koyun referansı olsa bile sadece Memeli sınıfının üye elemanlarına ulaşılabilir. Bunun sebebi daha önce de denildiği gibi Temel sınıfın Türeyen sınıfın üye elemanları ile ilgili herhangi bir bilgiye sahip olmamasıdır.

Daha önce sıkça vurgulamış olduğumuz **object** sınıfına her türden veriyi aktarabilmemizin sebebi de budur. Biz istesek de istemesek de C#'ta bütün değer ve referans türlerinin tamamı gizlice **object** sınıfından türetildiğini biliyoruz. Dolayısıyla her sınıf bir **object**'tir aslında. Nasıl String türü bir object ise Kedi ve Memeli sınıfı türünden tanımlanmış her nesne de bir object'tir. Bunu daha iyi anlamak için aşağıdaki gibi bir Main() metodunu yazarak test edin.

```
class MainMetodu
{
      static void Main()
      {
          Memeli memeli1 = new Memeli(10,35);
          Kedi kedi1 = new Kedi("Van", 5 , 10);

          object o1 = memeli1;
          object o2 = kedi1;

          Console.WriteLine(o1.ToString());
          Console.WriteLine(o2.ToString());

      }
}
```

Yukarıdaki Main() metodunu içeren bir program derlenip çalıştırıldığında yandaki ekran görüntüsü elde edilir.



Kendi tanımladığımız sınıflar da dahil olmak üzere bütün veri türlerinde bulunan ToString() metodunun aslında Object sınıfının bir üye metodu olduğunu bilmek gerekir. Diğer veri türleri de Object sınıfından türediği için her türde ToString() metodu mevcuttur. Biz kendi sınıflarımızda, **new** anahtar sözcüğünü kullanarak ya da kullanmayarak ToString() metodunu yeniden bildirebiliriz.

ToString() metodu, ilgili sınıfın string şeklindeki gösterimini sunar. Varsayılan olarak kullanıldığı nesnenin türünü string şeklinde geri döndürür.

Aşağıdaki örnekte Object sınıfının üye elemanı olan Tostring() metodunun nasıl yeniden bildirildiği görülmektedir.

```
using System;

class A
{
      int a;

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

      new public string ToString()
      {
          return "Nesnenin değeri = " + this.a;
      }
}

class MainMetodu
{
      static void Main()
      {
          A a = new A(5);

          Console.WriteLine(a.ToString());
      }
}
```

Program derlenip çalıştırıldığında ekrana

Nesnenin değeri = 5

yazacaktır.

Object türü genel bir veri türü olmasına rağmen tür güvenliği ve performans açısından kötü sonuçlara yol açabildiği için, çok gerekmediği sürece kullanmak pek tavsiye edilen bir durum değildir.

Sanal Metotlar

Şu ana kadar bir nesne üzerinden çağırdığımız metotların tamamı derleme zamanında belirgindi. Yani derleme aşamasında hangi nesne üzerinden hangi metotların çağrılabileceği bilinmekteydi. Şu ana kadar öğrendiğimiz bilgilerle bunun aksini gerçekleştirmek zaten mümkün değildir. Ancak bu kısımda göreceğimiz sanal metot kavramı ile çalışma zamanında metot seçme işinin nasıl olduğunu inceleyeceğiz.

Sanal metotlar temel sınıflar içinde bildirilmiş ve türeyen sınıflar içinde de tekrar bildirilen metotlardır. Şimdi diyebilirsiniz ki bunun isim gizleyerek bildirilen metotlardan farkı ne? Bu soruyu sormakta çok haklısınız, çünkü henüz sanal metotların özelliklerini incelemedik. Sanal metotlar nesne yönelimli programlama tekniğindeki çok biçimliliği (polimorphism) uygulayan yapılardır. Temel sınıfta bir sanal metot bildirildiğinde bu temel sınıftan türeyen sınıflar, temel sınıftaki sanal metodu devre dışı bırakarak kendilerine uygun metot gövdesi oluşturabilirler. Sanal metotların en önemli özelliği geç bağlam (late binding) özelliğini desteklemesidir.

Sanal metotlar sayesinde Temel sınıf türünden bir referansa türeyen sınıf referansları aktarıldığında, temel sınıf referansı üzerinden kendisine aktarılan türeyen sınıfın sanal metodu çağrılabilir. Eğer türeyen sınıf sanal metodu devre dışı bırakmamışsa temel sınıftaki sanal metot çağrılır. Çağrılan metodun hangi türe ait olduğu ise çalışma zamanında belirlenir. Metotların bu şekilde çalışma zamanında belirlenmesine geç bağlama **(late binding)** denilmektedir.

Sanal metotlar **virtual** anahtar sözcüğü kullanılarak bildirilir. Bu anahtar sözcük, metot bildiriminin başına eklenirse soyut metotlar bildirilmiş olur. Türeyen sınıfta, temel sınıftaki soyut metotları devre dışı bırakmak için ise **override** anahtar sözcüğü kullanılır.

Bütün bu kavramları daha iyi anlamak için bir örnek verelim. Memeli sınıfına Konus() isimli bir sanal metot eklenecektir. Bu metot çağrıldığında her hayvanın kendi konuşma biçimi ekrana yazdırılacaktır.

```
using System;

class Memeli
{
      public double Boy;
      public double Agirlik;

      public Memeli(double boy,double agirlik)
      {
          this.Boy = boy;
          this.Agirlik = agirlik;
      }

      virtual public void Konus()
      {
          Console.WriteLine("Ben konusamam");
      }
}

class Kedi : Memeli
{
      public string Turu;

      public Kedi(string turu, int boy,int agirlik):
                              base(boy,agirlik)
```

```
        {
            this.Turu = turu;
        }

        override public void Konus()
        {
            Console.WriteLine("Ben bir kediyim!");
        }
}

class Koyun : Memeli
{
        public string Turu;

        public Koyun(string turu, int boy,int agirlik) :
                                    base(boy,agirlik)
        {
            this.Turu = turu;
        }

        override public void Konus()
        {
            Console.WriteLine("Ben bir koyunum!");
        }
}

class MainMetodu
{
        static void Main()
        {
            Memeli memeli1 = new Memeli(10,35);
            Kedi kedi1 = new Kedi("Van", 5 , 10);
            Koyun koyun1 = new Koyun("Ankara",50,65);

            memeli1 = kedi1;
            memeli1.Konus();

            memeli1 = koyun1;
            memeli1.Konus();
        }
}
```

Bu programı derleyip çalıştırdığımızda ekrana

```
Ben bir kediyim!
Ben bir koyunum!
```

yazdığını görürüz. Eğer Konus() metodu sanal metot olmasaydı

```
memeli1 = kedi1;
memeli1.Konus();
```

deyimleri ile ekrana

```
Ben konusamam
```


yazısı yazacaktı. Yani Memeli sınıfındaki metot çağrılacaktı. Halbuki bu programda Memeli sınıfı türünden bir referansa Kedi ve Koyun nesnelerine ait referans aktarılmasına rağmen çalışma zamanında bu Memeli referansı üzerinden Konus() metodunu çağırdığımızda Memeli sınıfındaki Konus() metodu yerine kendisine atanan sınıflara ait Konus() metotları çağrılmıştır.

Sanal metotlar sayesinde temel sınıf referanslarına türeyen sınıf referansları atandığında, temel sınıf referansı üzerinden türeyen sınıfa ait metotları çağırabilmekteyiz. Bu yöntemle sadece temel sınıfta sanal olarak bildirilmiş metotların türeyen sınıfta devre dışı bırakılmış olan metotlara erişilebilir.

Şimdi sanal sınıflarının etkili kullanımına bir örnek daha verelim. Aşağıdaki programda statik AlanBul() metodu kendisine gönderilen Ucgen ve Dortgen nesnelerinin alanlarını ekrana yazdırmaktadır. AlanBul() metodu Sekil türünden nesneler beklemesine rağmen Sekil sınıfında tanımlanan sanal AlanBul() metodu sayesinde metodun parametresi olan Sekil referansı üzerinden AlanBul() metodu her sınıf için ayrı olacak şekilde çağrılmaktadır.

```
using System;

class Sekil
{
        public double Boy;
        public double En;

        public Sekil(double boy,double en)
        {
            this.Boy = boy;
            this.En = en;
        }

        virtual public double Alan()
        {
            return 0;
        }
}

class Dortgen : Sekil
{
        public Dortgen(int boy,int en):base(boy,en)
        {
        }

        override public double Alan()
        {
            return En * Boy;
        }
}
```

```
class Ucgen : Sekil
{
      public Ucgen(int boy,int en):base(boy,en)
      {
      }

      override public double Alan()
      {
          return En * Boy / 2;
      }
}

class MainMetodu
{
      public static void AlanBul(Sekil sekil)
      {
          Console.WriteLine("Şeklin alanı : " + sekil.Alan());
      }

      static void Main()
      {
          Ucgen ucgen = new Ucgen(10,50);
          AlanBul(ucgen);

          Dortgen dortgen = new Dortgen(10,50);
          AlanBul(dortgen);

          Sekil sekil = new Sekil(10,50);
          AlanBul(sekil);
      }
}
```

Program derlenip çalıştırıldığında ekrana aşağıdaki görüntü yansır.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum11\kod>Sekil
Seklin alani : 250
Seklin alani : 500
Seklin alani : 0

H:\cskitap\kitap\Bolum11\kod>
```

Gördüğünüz gibi statik AlanBul() metodu Sekil türünden bir nesne beklemesine rağmen Sekil türünden türeyen Ucgen ve Dortgen nesnelerini de kabul etmektedir. Üstelik bu Ucgen ve Dortgen nesneleri metodun parametresi olan Sekil nesnesine aktarılmasına rağmen Ucgen ve Dortgen sınıflarındaki Alan() metotları çağrılabilmiştir. Bunu sağlayan da şüphesiz Alan() metodunun sanal olarak bildirilmesi ve Tureyen sınıflarda bu metodunun devre dışı bırakılmasıdır.

Türeyen sınıflar, temel sınıflardaki sanal metotları devre dışı bırakmak zorunda değildir. Bu durumda eğer temel sınıf referansına, tureyen sınıf referansı aktarılırsa bu referans üzerinden temel sınıfa ait sanal metot çağrılır. Örneğin, aşağıdaki programda Dortgen sınıfı Alan() metodunu devre dışı bırakmamıştır.

```
using System;

class Sekil
{
      public double Boy;
      public double En;

      public Sekil(double boy,double en)
      {
          this.Boy = boy;
          this.En = en;
      }

      virtual public double Alan()
      {
          return 0;
      }
}

class Dortgen : Sekil
{
      public Dortgen(int boy,int en):base(boy,en)
      {
      }
}

class Ucgen : Sekil
{
      public Ucgen(int boy,int en):base(boy,en)
      {
      }

      override public double Alan()
      {
          return En * Boy / 2;
      }
}

class MainMetodu
{
      public static void AlanBul(Sekil sekil)
      {
          Console.WriteLine("Şeklin alanı : " + sekil.Alan());
      }
```

```
        static void Main()
        {
            Ucgen ucgen = new Ucgen(10,50);
            AlanBul(ucgen);

            Dortgen dortgen = new Dortgen(10,50);
            AlanBul(dortgen);

            Sekil sekil = new Sekil(10,50);
            AlanBul(sekil);
        }
}
```

Programı bu şekilde derlediğimizde programın akışı

```
Dortgen dortgen = new Dortgen(10,50);
AlanBul(dortgen);
```

satırlarına geldiğinde temel sınıfın yani Sekil sınıfının Alan() metodu çağrılacaktır. Hiyerarşik sınıf yapılarında da bu kural geçerlidir. Hiyerarşik yapıdaki sınıflarda yukarıdaki gibi bir kullanım ile karşılaşılırsa sınıf hiyerarşisinde en alttan en üste doğru Alan() metodunun ilk devre dışı bırakıldığı sınıfa ait olan metot çağrılır.

Bu örneklerden de görüldüğü üzere tek bir metot ile birçok sınıfa ait farklı versiyondaki metotlar çağrılabilmektedir. Üstelik bu işlem aynı nesne üzerinden olmaktadır. Bir nesnenin bu şekilde çoklu özellik göstermesine **çok çeşitlilik (polymorphism)** denilmektedir.

Çok çeşitlilik ve metotların devre dışı bırakılması ile ilgili öğrendiklerimizi pekiştirmek gerekirse;

1. Eğer metot sanal olarak bildirilmemişse, derleyici nesnelerin tür bilgisinden faydalanarak derleme zamanında hangi metodun çağrılacağını bilir.

2. Eğer metot sanal olarak bildirilmişse, derleyici derleme aşamasında ürettiği kod ile çalışma zamanında referansın türüne göre ilgili sınıfın devre dışı bırakılmış metodunu çağırır.

3. Hangi metodun çağrılacağının çalışma zamanında belirlenmesine geç bağlama (**late binding**) denilmektedir.

4. Sanal metot bildirmek için **virtual** anahtar sözcüğü kullanılır.

5. Türeyen sınıfta, temel sınıftaki sanal metodu devre dışı bırakmak için **override** anahtar sözcüğü kullanılır.

6. Türeyen sınıfta devre dışı bırakılan metotların temel sınıftaki sanal metotların ismi ile aynı olmalıdır.

7. Türeyen sınıfta devre dışı bırakılan metotların parametrik yapısı temel sınıftaki sanal metodun parametrik yapısı ile aynı olmalıdır.

8. Statik metotlar sanal olarak bildirilemez.

9. Türeyen sınıflar, temel sınıftaki sanal metotları devre dışı bırakmak zorunda değildir. Bu durumda temel sınıf referansları üzerinden temel sınıfa ait metot çağrılır.

Özet (Abstract) Sınıflar

Nesne yönelimli programlamada sınıf hiyerarşisi oluştururken bazen hiyerarşinin en tepesinde bulunan sınıf türünden nesnelerin programcı için pek bir anlamı olmayabilir. Hiyerarşinin en tepesinde bulunan sınıfın kendisinden türetilecek diğer sınıflar için ortak özellikleri bir arada toplayan bir arayüz gibi davranmasını isteyebiliriz. Yani en tepedeki sınıf diğer sınıfların neye benzeyeceğini belirtsin isteyebiliriz. Örneğin, Memeli sınıfında tanımlanan Konus() isimli bir metot çağrıldığından yapılacak iş belli değildir. Çünkü hangi memeli hayvanın mimikleri ile konuşulacağı belli değildir. Eğer Konus() metodu Memeli sınıfında bildirilmezse bu sefer Memeli sınıfından türeyecek sınıfların (Kedi, Koyun) çok biçimliliği desteklememesi söz konusu olur. O zaman bir tek çözüm var: Memeli sınıfında Konus() metodu bildirilsin ama herhangi bir iş yapması için özelleştirilmesin, sadece Memeli sınıfından türeyecek diğer sınıflar için bir arayüz oluştursun. İşte bu tür metotlara ve sınıflara özet metot ya da özet sınıf denilmektedir.

Özet sınıflar ya da özet metotlar **abstract** anahtar sözcüğü kullanılarak bildirilir. Bir temel sınıf içinde bildirilen özet (abstract) metotların temel sınıf içerisinde gövdesi yoktur. Ancak bu temel sınıftan türeyen bütün sınıflar bu metodu devre dışı (override) bırakmalıdır. Özet metotlar içsel olarak zaten sanal oldukları için ayrıca **virtual** anahtar sözcüğünü kullanarak sanal olduklarını belirtmeye gerek yoktur.

Şimdi özet metotların nasıl bildirildiğine bakalım. Özet metotlar metot bildirimin başına abstract anahtar sözcüğünün konulması ile bildirilir. Aşağıda abstract anahtar sözcüğünün metotlar üzerindeki kullanımı görülmektedir.

```
abstract public void OzetMetot();
```

Görüldüğü gibi OzetMetot() metodunun herhangi bir gövdesi yoktur. Bir sınıf içinde

```
abstract public void OzetMetot()
{

}
```

şeklinde gövdesi boş da olsa bir metot bildirimi yapmak yasaklanmıştır.

Özet sınıflar da **abstract** anahtar sözcüğü kullanılarak bildirilir. Bir özet sınıf türünden nesneler tanımlanamaz. Özet sınıflar sadece kendilerinden türeyen sınıflar için arayüz oluştururlar. Aşağıda özet sınıfların nasıl bildirildiğini görüyorsunuz.

```
abstract class OzetSinif
{

}
```

Özet sınıflar temel sınıfın tek başına anlamlı bir nesneyi ifade etmediği durumlarda kullanılır. Özet sınıfları kullanabilmek için bu sınıflardan türeyen yeni sınıflar bildirmemiz gerekir. Örneğin, Memeli sınıfını aşağıdaki gibi özet olarak bildirirsek Memeli sınıfı türünden nesneler tanımlayamayız.

```
using System;

abstract class Memeli
{
      public double en;
      public double boy;

      public Memeli(double en, double boy)
      {
          this.en = en;
          this.boy = boy;
      }
}

class MainMetodu
{
      static void Main()
      {

          /*
           Hatalı kullanım:
           Çünkü özet sınıflar türünden
           nesneler tanımlanamaz
          */
          Memeli m = new Memeli(5,15);
      }
}
```

Buna rağmen özet bir sınıftan türeyen sınıf nesneleri üzerinden özet sınıfa ait yapıcılar kullanılarak özet sınıfın üye değişkenleri değiştirilebilir. Aşağıdaki kodda Memeli sınıfından türeyen Kedi sınıfından nesnelerin nasıl oluşturulduğu ve bu nesnelere nasıl ulaşıldığı gösterilmiştir.


```
using System;

abstract class Memeli
{
      public double en;
      public double boy;

      public Memeli(double en, double boy)
      {
          this.en = en;
          this.boy = boy;
      }
}

class Kedi : Memeli
{
      string Turu;

      public Kedi(string turu, double en,double boy):base(en,boy)
      {
          this.Turu = turu;
      }
}

class MainMetodu
{
      static void Main()
      {
          Kedi kedi = new Kedi("Van",5,15);
          Console.WriteLine("Kedinin eni: " + kedi.en);
          Console.WriteLine("Kedinin boyu: " + kedi.boy);
      }
}
```

Bu programı derlediğinizde kedi nesnesinin başarı ile oluşturulduğunu göreceksiniz.

Şimdi de özet metotların türeyen sınıflardaki durumuna bakalım. Türeyen sınıflarda temel sınıfta bulunan bütün özet metotların devre dışı bırakılması gerekir. Eğer devre dışı bırakılmayan bir metot varsa program derlenemez. Buna göre aşağıdaki sınıf bildirimlerini içeren program hata verecektir.

```
abstract class Memeli
{
      public double en;
      public double boy;

      public Memeli(double en, double boy)
      {
          this.en = en;
```

```
            this.boy = boy;
        }

        abstract public void Konus();
    }

    class Kedi : Memeli
    {
        string Turu;

        public Kedi(string turu, double en,double boy):base(en,boy)
        {
            this.Turu = turu;
        }
    }
```

Bu sınıf bildirimlerinin oluşturacağı hata " 'Kedi' does not implement inherited abstract member 'Memeli.Konus()'"dur. Yani: Kedi türediği sınıfın özet metodu olan Memeli.Konus() metodunu devre dışı bırakmamıştır.

Kuşkusuz özet sınıflar içinde özet olmayan metotlar da bildirilebilir. Ancak bir sınıf içerisinde bir özet metot bildirilmişse sınıfın da mutlaka özet olması gerekir. Örneğin aşağıdaki sınıf bildirimi geçersizdir.

```
class Sınıf
{
    abstract public int OzetMetot(double a);
}
```

Diğer bir önemli nokta da özet metotların private olarak bildirilemeyeceğidir. Çünkü eğer özet metotlar private olursa kendisinden türeyecek olan diğer sınıflar bu özet metodu devre dışı bırakamaz. Buna göre aşağıdaki sınıf bildirimi de geçersizdir.

```
class Sınıf
{
    abstract private int OzetMetot(double a);
}
```

Tahmin edebileceğiniz gibi özet metotlar public ya da protected olarak bildirilebilirler.

Metot ve sınıflar gibi özellikler de özet olarak bildirilebilir. get ve set bloğu olan bir özet özellik aşağıdaki gibi bildirilir.

```
abstract class OzetOzellik
{
    abstract public int A
    {
        get;
        set;
    }
}
```


OzetOzellik sınıfından türeyen bir sınıf A özelliğini hem get hem de set blokları ile beraber uygulamak zorundadır. Eğer temel sınıfta get ya da set bloklarından herhangi biri bildirilmiş olsaydı tureyen sınıfta sadece ilgili bloklar yazılırdı. Örneğin, hem get hem de set bloğu olan bir özet özellik içeren sınıftan aşağıdaki gibi bir türetme kullanımı geçersizdir.

```
abstract class TemelSınıf
{
    abstract public int A
    {
        get;
        set;
    }
}

class TureyenSınıf :TemelSınıf
{
 int a;

    public override int A
    {
        get{ return a;}
    }
}
```

Türeyen sınıftaki özelliğin geçerli olması için A özelliği aşağıdaki gibi bildirilmelidir.

```
public override int A
{
    get
    {
        return a;
    }
    set
    {
        a = value;
    }
}
```

Şimdi isterseniz özet sınıflar ve metotlar ile ilgili söylediklerimizi özetleyelim.

1. Özet sınıflar türünden nesneler tanımlanamaz.

2. Özet sınıflar, özet metotlar içerebilirler. Özet metotlar ancak özet sınıfların içinde bildirilebilir.

3. Özet sınıflar bir sonraki kısımda göreceğimiz **sealed** anahtar sözcüğü ile işaretlenemezler. **sealed** anahtar sözcüğü, bir sınıftan türetme yapılamayacağını belirtir.

4. Özet bir sınıftan türeyen sınıflar temel sınıftaki bütün özet metotları uygulamalıdır. Yani metodun gerçek gövdesi türeyen sınıflarda yazılmalıdır.

5. Bir özet metot ancak özet bir sınıf içinde bildirilebilir.

6. Bütün özet sınıflar içsel olarak sanaldır (virtual). Bu yüzden ayrıca virtual olduklarını belirtmeye gerek yoktur.

7. Özet metotlar türeyen sınıfta mutlaka bildirilmelidir.

8. Statik metotlar özet olarak bildirilemez. Aynı şekilde özet metot bildiriminde **override** ve **virtual** anahtar sözcükleri kullanılamaz.

9. Sınıflarda özet özellikler de bildirilebilir, türeyen sınıflarda bu özet özellikler **override** anahtar sözcüğü ile tekrar bildirilmelidir.

sealed Anahtar Sözcüğü

Şu ana kadar incelemiş olduğumuz bütün örnek sınıflardan türetme yapabilmekteyiz. Ancak bazı durumlarda oluşturduğumuz sınıfların türetilmesine engel olmak isteyebiliriz. Bunun birçok sebebi olabilir. Birincisi gerçekten ilgili sınıftan türetme yapmanın anlamsız olmasıdır. İkincisi ise güvenlik sorunları olabilir. Örneğin, donanımsal bir iş yapan sınıfın programcılara sunulduktan sonra sınıfın var olan özelliklerine ek bir takım özelliklerin eklenmesi var olan diğer metotların hatalı çalışmasına yol açabilir. Bu tür durumlarda oluşturulan sınıfların kesinlikle türetilmemesini istiyorsak sınıf bildirimlerini sealed anahtar sözcüğü ile gerçekleştirmeliyiz. Türetilmesi engellenen bir sınıf bildirimi aşağıdaki gibi yapılabilir.

```
sealed class TemelSınıf
{

}
```

sealed anahtar sözcüğü kullanılarak bildirilmiş bir sınıf

```
sealed class TemelSınıf
{
}

class TureyenSınıf : TemelSınıf
{
}
```

şeklinde türetilmeye çalışıldığında " 'TureyenSınıf' : cannot inherit from sealed class 'TemelSınıf' (TureyenSınıf sealed olarak işaretlenmiş TemelSınıf' tan türetilemez) hata mesajı alınır.

sealed anahtar sözcüğü özellikle bütün metotları statik olan sınıflar için gerçekten uygun görünüyor. Bütün metotları statik olan sınıfı türetmenin çok fazla anlamlı olmadığı açıktır. Çünkü temel sınıfa ilişkin nesne referansı üzerinden herhangi bir metot zaten çağrılamamaktadır. Ayrıca, .NET sınıf kütüphanesinde kendisinden türetme yapamayacağımız birçok sınıfın olduğunu da bilmek gerekir.

Diğer bir önemli konu olan arayüzlere (interface) geçmeden önce türetme ve çok biçimlilik ile ilgili şu ana kadar gördüğümüz bütün temel konuları içeren bir örnek verelim.

```
using System;

abstract class Sekil
{
      protected int mBoy;
      protected int mEn;

      public Sekil(int boy,int en)
      {
          this.Boy = boy;
          this.En = en;
      }

      abstract public int Boy
      {
          get;
          set;
      }

      abstract public int En
      {
          get;
          set;
      }

      abstract public int Alan();
      abstract public void EkranaYaz();
}


class Dortgen : Sekil
{
      private bool mKaremi;

      public Dortgen(int boy,int en):base(boy,en)
      {
          KareKontrol();
      }
```

```
        public bool Karemi
        {
            get{ return mKaremi;}
        }

        override public int Boy
        {
            get{ return mBoy;}
            set
            {
                mBoy = value;
                KareKontrol();
            }
        }

        override public int En
        {
            get{ return mEn;}
            set
            {
                mEn = value;
                KareKontrol();
            }
        }

        private void KareKontrol()
        {
            if(mBoy == mEn)
                mKaremi = true;
            else
                mKaremi = false;
        }



        override public int Alan()
        {
            return mEn * mBoy;
        }

        override public void EkranaYaz()
        {
            Console.WriteLine("\nDortgenin eni: " + mEn);
            Console.WriteLine("Dortgenin boyu: " + mBoy );
            Console.WriteLine("Karemi : " + (mKaremi ?
                                "Evet" : "Hayır"));
            Console.WriteLine("Dortgenin alanı: " + Alan());
            Console.WriteLine();
        }
}
```



```
class Ucgen : Sekil
{
        private string UcgenTipi;
        public Ucgen(string Tur,int boy,int en):base(boy,en)
        {
            UcgenTipi = Tur;
        }

        override public int Boy
        {
            get{ return mBoy;}
            set{ mBoy = value;}
        }

        override public int En
        {
            get{ return mEn;}
            set{ mEn = value;}
        }

        override public int Alan()
        {
            return mEn * mBoy/2;
        }

        override public void EkranaYaz()
        {
            Console.WriteLine("\nÜcgenin eni: " + mEn);
            Console.WriteLine("Üçgenin boyu: " + mBoy);
            Console.WriteLine("Üçgenin tipi: " + UcgenTipi);
            Console.WriteLine("Üçgenin alanı: " + Alan() + "\n");
        }
}

class MainMetodu
{

        static void Main()
        {
            Sekil[ ] sekil = new Sekil[ 3] ;

            sekil[ 0] = new Ucgen("Eskenar",6,5);
            sekil[ 1] = new Dortgen(6,5);
            sekil[ 2] = new Dortgen(4,4);

            foreach(Sekil s in sekil)
                s.EkranaYaz();
        }
}
```

İsterseniz programı biraz yakından inceleyelim. Sekil sınıfında Boy, En özellikleri ile Alan(), EkranaYaz() metotları özet olarak bildirilmiştir. Dolayısıyla Sekil sınıfından

türeyen Ucgen ve Dortgen sınıfı bu elemanların gövdelerini uygulamalıdır. Dikkat ederseniz Dortgen sınıfının kare olup olmadığına bir metot yerine özellik tarafından erişiliyor. Bu da bazı durumlarda performans artışına sebep olur. Şöyle ki, eğer bir dortgenin kare olup olmadığını program içerisinde sıkılıkla test etmek zorunda iseniz her defasında Karemi() metodunu çağırmak yerine, Dortgenin en ve boy değerlerinin değiştiği sırada bir değişken ile dortgenin karesellik durumu değiştirilir. Zaten Dortgen sınıfının yapıcı işlevinde ve En, Boy özelliklerinin set bloklarında KareKontrol() isimli metot çağrılarak mKaremi üye değişkeni true ya da false değere getiriliyor. Programla ilgili diğer bir önemli nokta da Main() metodu içindeki

```
Sekil[] sekil = new Sekil[3];
```

satırıdır. Sekil sınıfı özet sınıf olduğu için Sekil sınıfı türünden **new** operatörü kullanılarak bir nesne oluşturulamaz. Yukarıda kullanılan **new** anahtar sözcüğü kafanızı karıştırmasın. Burada kullanılan new anahtar sözcüğü sekil dizi nesnesinin oluşturulması içindir. Bu dizi içinde 3 tane null değere atanmış Sekil referansı bulunmaktadır. Konuya biraz daha açıklık getirmek için aşağıdaki deyimin geçerli olduğunu belirtelim.

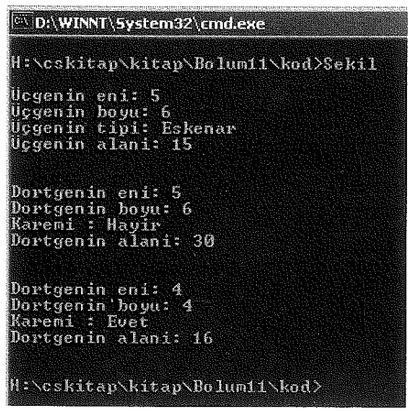
```
Sekil sekil1;
```

Burada dinamik bir nesne yaratılmıyor sadece sekil referansı yaratılıyor. Bu sekil referansına aşağıdaki gibi Ucgen ya da Dortgen referansları atanabilir.

```
Sekil sekil1;
Sekil sekil1 = new Dortgen(5,6);
```

Programdaki sekil dizisi de hem Ucgen hem de Dortgen sınıfının ortak metotlarını çok biçimlilikten faydalanarak kullanmak için tanımlanmıştır.

Programın ekran çıktısı aşağıdaki gibidir.



Main() metodunu değiştirerek programın ne şekilde çıktı ürettiğini incelemeniz konuyu daha iyi anlamanız açısından önemlidir.

Arayüzler (Interface)

Bir önceki kısımda gördüğümüz özet metotlar ile bir özet metodun bulunduğu sınıftan türeyen bir sınıfın mutlaka bu özet metodu devre dışı bırakıp kendine göre uygulaması gerekirdi. Burada yaptığımız aslında bir sınıfın neye benzediğini tespit etmekti. Özet metotlar sayesinde biliyoruz ki bir önceki örnekte Dortgen sınıfında ve türeyen bütün sınıflarda Alan() metodu uygulanmıştır. Bu da tek bir arayüzle birçok iş yapabileceğimiz anlamına gelir. Yani Sekil sınıfının bir kısım elemanları kendisinden türeyecek sınıflara arayüz niteliği taşımaktadır. Ayrı bir veri türü olan **Interface** ise tamamen diğer sınıflar için arayüz görevini taşır. Arayüzlerin bütün metotları ve özelliklerinin özet olarak bildirilmiş sınıflardan çok fazla farkı yoktur. Dolayısıyla arayüzlerdeki metotların ve özelliklerin gövdesi yazılamaz. Kısaca arayüzler, kendisini uygulayan sınıfların kesin olarak içereceği özellikleri ve metotları belirler.

Arayüzler kişisel uygulamalarda pek fazla kullanılmaz ancak özellikle birkaç firmanın ya da programcının üzerinde çalıştığı projelerde ortak bir zemin oluşturabilmek için arayüzlerden faydalanılır.

.NET sınıfı kütüphanesinde de birçok arayüz bulunmaktadır. Her arayüz çeşitli amaçlar için yazılmıştır. Örneğin, System isim alanında bulunan IDisposable arayüzü gereksiz nesne toplayıcısı (garbage collector) için gerekli olabilecek Dispose() metodunun uygulanmasını zorlar. Aynı şekilde System.Collections isim alanında bulunan IEnumarable arayüzü tanımladığımız sınıfların foreach döngü yapısı ile kullanabilmesini sağlar. IEnumarable arayüzü içindeki metotlar ilgili sınıf tarafından uygulanırsa tanımlanan bu sınıf foreach döngüsü ile kullanılabilir. Çünkü foreach döngüsü çalıştırılmaya başlanınca foreach bloğunda kullanılan sınıfın IEnumarable arayüzünü uygulayıp uygulamadığı kontrol edilir. Eğer bu arayüz uygulanmışsa IEnumarable arayüzündeki metotlar kullanılarak foreach döngüsü işletilir. Arayüzleri nasıl bildirdiğimizi inceledikten sonra IEnumerable arayüzünün kullanımına bir örnek vereceğiz.

Arayüz Bildirimi

Arayüzler, **interface** anahtar sözcüğü kullanılarak bildirilir. Bir arayüzde özellik, metot, indeksleyici (**indexer**) veya daha sonra göreceğimiz temsilci (**delegate**) ve olay (**event**) bildirimi yapılabilir. Biz şimdilik sadece metot, özellik ve indeksleyici bildirimini inceleyeceğiz. Arayüz isimleri geleneksel olarak I harfi ile başlar, ancak bu zorunlu değildir. Biz de bu geleneği devam ettireceğiz. Yani arayüzleri isimlendirirken isimleri I harfi ile başlatacağız.

Arayüz bildirimi ile ilgili çeşitli kısıtlamalar da vardır. Bunlar aşağıdaki gibi sıralanabilir.

- Arayüzdeki elemanları statik olarak bildiremeyiz.
- Arayüzdeki eleman bildirimleri içsel olarak public oldukları için ayrıca bir elemanı erişim belirleyici ile bildirmek yanlıştır.

- Arayüzler herhangi bir üye değişken içeremez.
- Arayüzlerde yapıcı ve yıkıcı metotlar tanımlanamaz ya da bildirilemez.

Bütün bu bilgiler ışığında aşağıda iki metot, bir özellik ve bir indeksleyici bildiriminin yapıldığı arayüzü görmektesiniz.

```
interface IArayuz
{
      int BirSonraki();
      void Sıfırla();

      int Deger
      {
          get;
          set;
      }

      int this[ int indeks]
      {
          get;
      }
}
```

Bu arayüzü uygulayan bir sınıfta Arayüzde bulunan bütün elemanların uygulanması gerekir. Deger özelliğinin hem get hem de set blokları olmasına karşın indeksleyicinin sadece get bloğu vardır. Buna rağmen bu arayüzü uygulayacak sınıfta bildirilen indeksleyicinin hem get hem de set bloğu ya da sadece get bloğu olabilir, fakat sadece set bloğu olamaz.

Şimdi de bazı geçersiz arayüz bildirimlerini inceleyelim.

Aşağıdaki arayüz bildiriminde metot public erişim belirleyicisi ile bildirildiği için geçersizdir.

```
interface IArayuz
{
      public void Metot();
}
```

Aşağıdaki arayüz bildiriminde statik metot bildirimi yapıldığı için geçersizdir.

```
interface IArayuz
{
      static void Metot();
}
```

Aşağıdaki arayüz bildiriminde ise üye değişkenin bildirilmesi arayüzü geçersiz kılmıştır.


```
interface IArayuz1
{
      int a;

      int Metot();

      int Ozellik
      {
          get;
      }
}
```

Arayüzlerin Uygulanması

Şimdi bir sınıfın belirlediğimiz bir arayüzü nasıl uyguladığını inceleyeceğiz. Arayüzlerin uygulanması, sınıfların türetilmesi ile aynı şekilde yapılır. Arayüz uygulamada, arayüzler türetmedeki temel sınıf yerine geçer. Buna göre IArayuz'ü uygulayan Sayılar sınıfı aşağıdaki gibi bildirilir.

```
class Sayılar : IArayuz
{
      //Uygulanacak elemanlar
}
```

Sayılar sınıfı bu şekilde IArayuz'den türetildiğinde IArayuz'deki bütün üye elemanları uygulamalıdır.

Buna göre Sayılar sınıfı IArayuz'ü ile aşağıdaki gibi bildirilebilir.

```
interface IArayuz
{
      int BirSonraki();
      void Sıfırla();

      int Deger
      {
          get;
          set;
      }

      int this[ int indeks]
      {
          get;
      }
}

class Sayılar : IArayuz
{
```

```
        public int BirSonraki()
        {
            //Metot gövdesi
        }

        public void Sıfırla()
        {
            //Metot gövdesi
        }

        public int Deger
        {
            get
            {
                //get bloğu
            }
            set
            {
                //set bloğu
            }
        }

        public int this[ int indeks]
        {
            get
            {
                //get bloğu
            }
            set
            {
                //set bloğu
            }
        }
}
```

Sınıflar arasında çoklu türetme olmamasına rağmen sınıflar birden fazla arayüzü de uygulayabilir. Uygulanacak arayüzler virgül ile belirtilir. Örneğin IDisposable ve System.Collecitons isim alanında bulunan IEnumarable arayüzünü uygulayan sayılar sınıfı aşağıdaki gibi bildirilebilir.

```
using System;
using System.Collections;

class Deneme : IDisposable,IEnumerable
{
        //Sınıf elemanları
}
```


Bir arayüzdeki elemanlar içsel olarak public olduğu için arayüzü uygulayan sınıfın arayüzdeki elemanları public olarak bildirilmemelidir.

Bir arayüzü uygulayan sınıfın, sadece arayüzdeki elemanlara sahip olabileceği anlamına gelmez. Aynen türetmede olduğu gibi arayüzdeki elemanların dışında istenildiği kadar yeni üye eleman eklenebilir.

Sınıflar birbirlerinden nasıl türetilebiliyorsa Arayüzler de birbirlerinden türetilebilir. Temel arayüzdeki bütün elemanlar türeyen arayüze aktarılır. Böylece istediğimiz kadar eski arayüzü kullanarak daha geniş arayüzleri oluşturabiliriz. Aşağıda iki arayüzün türetilmesine ilişkin bir örnek verilmiştir.

```
interface TemelArayuz
{
        void Metot1();
}

interface TureyenArayuz : TemelArayuz
{
        void Metot2();
}
```

Bu kullanım ile TemelArayuz'ü uygulayan bir sınıf sadece Metot1'i uygulamalı iken TureyenArayuz'ü uygulayan bir sınıf hem Metot1() hem de Metot2()'yi uygulamalıdır. Aksi halde derleme hatası alınır. Sınıflardan farklı olarak arayüzleri birden fazla arayüzden türetebiliriz. Örneğin, aşağıdaki Arayuz3 bildirimi tamamen geçerlidir.

```
interface Arayuz1
{
        void Metot1();
}

interface Arayuz2
{
        void Metot2();
}

interface Arayuz3 : Arayuz1,Arayuz1
{
        void Metot3();
}
```

Arayüzleri türetirken **new** anahtar sözcüğü kullanılarak temel arayüzdeki bir eleman gizlenebilir. Bu şekilde türeyen arayüzde temel arayüzdeki bir elemanla aynı isimli eleman bildirilebilir. Örneğin

```
interface Arayuz1
{
        void Metot1();
```

```
}
interface Arayuz2 : Arayuz1
{
      new void Metot1();
}
```

Şimdi aklınıza şöyle bir soru gelmiş olabilir. Arayuz2'yi uygulayan bir sınıfta Metot1() metodu uygulandığında geçerli bir bildirim olur mu? Cevabımız hayır. Çünkü her ne kadar Arayuz1'deki Metot1() gizlenmiş olsa da kalıtım yolu ile Arayuz2'ye aktarılmıştır. Bu yüzden Arayuz2'yi uygulayan bir sınıf hem Arayuz1.Metot1() hem de Arayuz2.Metot1() metodunu uygulamalıdır. Bunun için '.' operatörü ile hangi arayüzdeki metodu uyguladığımızı bildirmeliyiz. Örneğin;

```
class Deneme : Arayuz2
{
      void Arayuz1.Metot1()
      {

      }

      void Arayuz2.Metot1()
      {

      }
}
```

Peki Deneme sınıfından bir nesne üzerinden Metot1() metoduna erişmemiz mümkün müdür? Cevabımız tabi ki hayır, çünkü

```
deneme d = new deneme();
d.Metot1();
```

şeklindeki bir deyim ile hangi arayüze ilişkin metodun çağrıldığı belli değildir. Bu yüzden yukarıdaki gibi bir kullanım hatalıdır. O halde Arayuz1.Metot1() ve Arayuz2.Metot1() metotları sadece Deneme sınıfında tam yolu aşağıdaki gibi açıkça belirtilerek kullanılır.

```
void Arayuz2.Metot1()
{
      Arayuz1.Metot1();
}
```

Arayüz Referansları

Arayüzler ile referanslar oluşturabilir. Evet yanlış duymadınız arayüz referansları oluşturulabilir. Peki bir arayüz referansı ne anlama gelir? Açıkça söylemek gerekirse tek başına bir arayüz referansı herhangi bir anlam ifade etmez. Ancak arayüz referanslarına kendisini uygulayan herhangi bir sınıf nesnesinin referansı atanabilir. Bu durumda arayüz referansı ile arayüzde bulunan metot ya da özellikler hangi sınıf referansı tutuluyorsa o sınıf türünden bir nesne için çağrılabilir. Şimdi bu çarpıcı özel-

liğe bir örnek verelim. Aşağıda ICSharp arayüzünü uygulayan iki sınıfın bildirimini görüyorsunuz. Main() metodu içinde ise ICSharp referansına bu iki sınıfın nesne referanslarının nasıl aktarıldığını ve bu referans üzerinden sınıflardaki metotların nasıl çağrıldığını görüyorsunuz.

```
using System;

interface CSharp
{
      void Metot1();
}

class Sınıf1 : CSharp
{
      public void Metot1()
      {
          Console.WriteLine("Sınıf1.Metot()");
      }
}

class Sınıf2 : CSharp
{
      public void Metot1()
      {
          Console.WriteLine("Sınıf2.Metot1()");
      }
}

public class MainMetodu
{
      public static void Main()
      {
          CSharp a;

          Sınıf1 sınıf1 = new Sınıf1();
          Sınıf2 sınıf2 = new Sınıf2();

          a = sınıf1;
          a.Metot1();

          a = sınıf2;
          a.Metot1();
      }
}
```

Programı derleyip çalıştırdığınızda ekrana

```
Sınıf1.Metot1()
Sınıf2.Metot1()
```

yazdığını göreceksiniz. Yukarıdaki işlem türetmede gördüğümüz temel sınıf referansına türeyen sınıf referansı aktarma işlemi ile aynıdır.

Açık (Explicit) Arayüz Uygulama

C# dilinde arayüzleri uygulamanın bir yolu daha vardır: **Explicit Interface Implementation**. Bu yöntem daha önce bahsedilen yöntemlerin aşağıdaki kısıtlarından kaynaklanmaktadır.

- Açıkça arayüz uygulama yöntemi ile istenirse türeyen sınıflarda arayüzde bulunan üye elemanlar açık bir şekilde nesneler tarafından erişilemez hale getirilebilir. İlgili üye elemanlarına sadece arayüz referansları ile erişilmesi sağlanır.
- Birden fazla arayüz uygulandığı durumlarda eğer aynı isimli üye elemanlar varsa isim çakışmasının önüne bu yöntemle geçilebilmektedir.

Açık arayüz uygulaması yapmak için arayüzün üye elemanları arayüz isimleri ile beraber belirtilir. Aşağıda buna bir örnek verilmiştir.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace explicitinterfaceimplemetantion
{
    class Program
    {
        static void Main(string[] args)
        {
            Kedi k = new Kedi();

            ((IMemeli)k).Konus();
        }
    }

    interface IMemeli
    {
        void Konus();
    }

    class Kedi : Memeli
    {
        void IMemeli.Konus()
        {
            Console.WriteLine("miyav");
        }
    }
}
```

Yukarıdaki örnekte de görüldüğü üzere **Konus()** metodunun çağrılabilmesi için **IMemeli** arayüzü referansına çevrilmesi gerekiyor. Direkt Kedi nesneleri üzerinden **Konus()** metodu çağrılamayacaktır. Böylece sınıf nesneleri için arayüz üye elemanları private gibi davranmaktadır. Normal yöntemle bunu yapmak mümkün değildir.

Arayüzlerle ilgili bölümün sonuna geldik. Arayüzlerin kullanımı, bildirimi, türetilmesi gibi temel konuları inceledik ama arayüzlerin gerçek bir programda kullanılmasına

örnek vermedik. Şimdi arayüzleri anlatmaya başladığımızda söylediğimiz IEnumerable arayüzünü uygulayan bir örnek yapacağız. IEnumerable arayüzünü uygulayan bir sınıf foreach döngü yapısı ile kullanılabildiğini söylemiştik. Tabi foreach döngü yapısı ile kullanabileceğimiz bir sınıfın yapısında dizi ya da daha sonra göreceğimiz koleksiyon tabanlı bir yapının bulunması mantıklı olur. Önce programı yazalım, ardından programda neler olup bittiğini inceleriz.

```
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] );
            }
        }

    }
}

public class MainMetodu
{
    public 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 + " ");
    }
}
```

Bu programı derlediğinizde ekrana

```
1 2 3 8 6 9 7
```

yazıldığını göreceksiniz.

Programda neler olup bitiyor bir göz atalım: Koleksiyon isimli sınıfın yapısındaki Dizinin eleman değerlerini foreach döngüsü ile elde etmek için Koleksiyon sınıfı System. Collections isim alanında bulunan IEnumerable arayüzünü uyguluyor. Dolayısıyla IEnumerable arayüzündeki bütün metotları uygulamalıdır. IEnumerable arayüzünde sadece

```
IEnumerator GetEnumerator();
```

metodu bildirilmiştir. Bu metot dizinin içerisinde dolaşmak için gerekli olan IEnumerator referansına geri dönmektedir. Bu yüzden bu metot aşağıdaki gibi yazılmıştır.

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


GetEnumarator() metodunun geri dönüş değeri için IEnumerator arayüzünden türetilmiş ENumaralandırma sınıfı bildirilmiştir. IEnumerator arayüzündeki metot ve özellikler Koleksiyon nesnesi içindeki dizinin elemanlarına erişmek için kullanılacaktır. IEnumerator arayüzünde bulunan üye elemanlar aşağıdaki gibidir.

```
object Current { get;}
bool MoveNext();
void Reset();
```

Bu üç elemanın da ENumaralandırma sınıfında bildirilmesi gerekir. Current özelliği koleksiyon nesnesinin herhangi bir andaki değerini gösterir. Sadece get bloğunu bildirmek yeterlidir. MoveNext() metodu dizi koleksiyon dizisi içinde bir sonraki elemana geçmek için kullanılır. Eğer bir sonraki elemana geçilemiyorsa false değerine geri döner. Reset() metodu ise koleksiyon dizisini herhangi bir andaki değerini ilk elemandan önceki elemana çeker. Yani dizinin indeksi –1 olur.

ENumaralandırma sınıfının bir tane yapıcı metodu vardır. Bu yapıcı metot ile üzerinde işlem yapılacak Koleksiyon nesnesi alınmaktadır. Yapıcı metot ile aynı zamanda ENumaralandırma sınıfındaki indeks değişkeni de –1 değerine çekiliyor. Reset() metodunda indeks değişkeni tekrar –1 değerine çekiliyor. MoveNext() metodunda ise indeks değişkeni bir artırılıyor. Eğer indeks değişkeninin yeni değeri dizinin boyutundan büyükse false değerine geri dönüyor. Yani burada dizinin elemanlarının bitiş noktası tespit ediliyor. Current özelliği ise dizinin herhangi bir andaki indeks ile belirtilmiş elemana geri dönmektedir. Current özelliğinin object olduğuna dikkat edin. Eğer bu özellik object türünden olmasaydı mümkün olabilecek tür dönüşümlerini hesaba katmazsak bütün bu işlemleri sadece Current özelliği ile aynı türden olan diziler üzerinde yapabilirdik.

Partial (Kısmi) Tipler

Dikkat ettiyseniz şu ana kadar gördüğümüz bütün tip bildirimleri (sınıf, struct, arayüz) tek bir *.cs* dosyası içerisinde tanımlandı. (Bu elbette bir zorunluluk değildir. Birden fazla tip bildirimini de tek bir dosyada yapabiliriz. Ancak şimdiye kadar tek bir tip bildirimini birden fazla dosyada yapmak ile ilgili bir bilgi edinmedik.) Bu durum gayet normaldir. Çünkü birbiri ile mantıksal bir bütünlük oluşturan üye elemanların tek bir kod dosyasında tanımlanması iyi bir tekniktir. C# 2.0'dan itibaren yeni bir tip bildirim biçimi ile karşılaşmaktayız.

C# 2.0'dan itibaren tip bildirimleri (sınıf, struct ya da interface bildirimleri) tek bir dosya içinde olmak zorunda değildir. Bir sınıfa ait üye elemanları farklı kod dosyalarında yazma imkanına sahibiz. Bir sınıfın üye elemanlarını farklı dosyalarda bildirmek için C# diline **partial** adından yeni bir anahtar sözcük eklenmiştir. partial olarak bildirilen sınıflar birden fazla kod dosyasında yer alabilir. Tabi bir sınıfın birden fazla kod dosyasında yer alması o sınıfın derleme zamanında birden fazla örneği olması

anlamına gelmiyor. Derleme aşamasında partial olarak bildirilen bütün aynı isimli sınıflar yine eskiden olduğu gibi tek bir çatı altında birleştirilmektedir. Yani sınıflar konusunda değindiğimiz metot aşırı yükleme ve yapıcı metot aşırı yüklemede olan kurallar tamamen geçerlidir. Kısaca, imzası aynı olan iki metot kısmi tip içinde tanımlanırsa derleme hatası meydana gelecektir.

Önemli Not: Derleme zamanında bütün aynı isimli kısmi türler birleştirilerek tek bir tür olarak ele alınırlar.

Örneğin, **Button** isimli bir sınıfı aşağıdaki gibi iki farklı *.cs* kod dosyası içinde tanımlayabiliriz.

Button1.Ozellikler.cs

```
partial class Button
{
    public int Uzunluk;
    public int Genislik;
    public string Ad;
}
```

Button1.Metotlar.cs

```
partial class Button
{
    public void Surukle(){...};
    public int Tıkla(){...};
    public void Kaldır(){...};
}
```

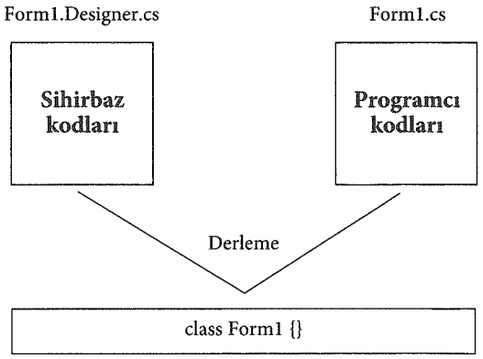
Yukarıdaki metotların içi boş bırakılmıştır. Siz içi dolu metotlarda test işleminizi yapabilirsiniz.

Yukarıdaki kod bloklarını derlediğimizde herhangi bir **Main** metodunda aşağıdaki gibi button nesnesi yarattığımızda hem özelliklere hem de metotlara aynı nesne üzerinden eriştiğimizi göreceksiniz.

```
static void Main(string[] args)
{
    Buttun b = new Button();
    b.Uzunluk = 3;
    b.Ad = "Dugme 1";
    b.Tıkla();
}
```


Önemli Not: İsimleri aynı olsa bile bütün kısmi türleri partial ile bildirmek zorunludur.

Aşağıdaki şema kısmı türlerin çalışma prensibini açıkça göstermektedir.



Kısmı türlerin C# diline eklenmesinde temel olarak iki neden vardır. Birincisi, özellikle windows ve web uygulamalarında sıkça karşılaşılan **designer** dediğimiz araçların ürettiği kodların (sürükle bırak ya da sihirbazlarla) programcıların yazdığı kodlarla karışmaması için designer araçlarının otomatik ürettiği kodlar farklı bir kod dosyasında **partial** olarak bildirilmektedir. Bu yeni özellik sayesinde hem programcılar otomatik üretilen kodlara daha az müdahale ediyor hale geldi hem de ortamdaki kod kalabalığı ortadan kalktı. Kısmi tiplerin diğer bir faydası ise takım halinde çalışan programcıların aynı anda aynı fiziksel dosya üzerinde işlem yapamaması durumunda sınıfı birden fazla dosyaya ayırarak esnekliği ve birlikte çalışabilmeyi kolaylaştırmasıdır. Örneğin, Oğuz isimli programcı ile Burak isimli programcı mantıksal olarak aynı sınıfta olması gereken ama farklı işlemler yapan iki metodu iki farklı dosyada tanımlayıp ayrı ayrı geliştirme yapabilirler. Böylece derleme aşamasında her iki programcının yazacağı metotlar tek bir sınıf içinde birleştirilecektir.

C# öğrenmeyi bitirip büyük uygulamalar geliştirmeye başladığınızda kısmi türlerin faydasını daha iyi anlayacağınızı rahatlıkla söyleyebilirim. Dolayısıyla şimdilik bütün tiplerinizi tek bir fiziksel dosyada tutmanızda sakınca görmüyorum.

Kısmi tip tanımlama biçimi sınıflar, arayüzler ve struct bildirimlerinde kullanılabilir.

Kısmi olarak tanımlanan tiplerin herhangi birinde **abstract** ya da **sealed** tanımı yapıldığında diğerlerinde aynı tanımı yapmaya gerek yoktur. Herhangi birinde yapılan sealed ya da asbtract tanımı sınıfın tamamı için geçerli olacaktır.

12

İleri C# Konuları

- İstisnai Durum Yönetimi (Exception Handling)
  - İstisnai Durum Sınıfları
  - İstisnai Durum Yakalama
  - Standart İstisnai Durum Sınıfları
  - İstisnai Durum Sınıfları Oluşturmak
- Temsilciler (delegates) ve Olaylar (events)
  - Temsilciler (delegates)
  - Çoklu Temsilciler (Multicast Delegate)
  - Delegate ve Multicast Delegate Sınıfları
  - İsimsiz Metotlar (Anonymous Methods) ve Yan Etkileri
  - Olaylar (Events)
  - .NET ve Olaylar
- Önişlemci Komutları
  - Önişlemci
- C#'ta Gösterici (Pointer) Kullanımı ve Emniyetsiz (unsafe) Kod
  - Gösterici Bildirimi ve Gösterici Operatörleri
  - Gösterici Aritmetiği ve Dizi İşlemleri
  - Göstericiler ile Dizi İşlemleri
  - Yapı (struct) Türünden Göstericiler
- Yansıma ve Nitelikler
  - Assembly Kavramı ve Assembly Sınıfı
  - Type Sınıfı ve Yansıma Kavramı
  - Çalışma Zamanında Metot Çağırma
  - Nitelikler (Attributes)
  - Özel Nitelik Bildirimi ve Niteliklerin Elde Edilmesi
  - Plugin (Eklenti) Tabanlı Programlama ve Yansıma
- C# Dilinin Dinamik Özelliği (dynamic ve DLR)
  - Statik ve Dinamik Dillerin Karşılaştırılması
  - dynamic ile Tanımlanan Tipler
  - Çalışma Sorusu

Bu bölümde, C# dilinin çok önemli beş konusunu inceleyeceğiz. Bunlardan birincisi, çalışma zamanında meydana gelebilecek hataları tespit etmek ve hatanın çeşidine göre programın akışını yönlendirmek için kullanılan istisnai durum yakalama (exception handling) mekanizmasıdır. İkinci konumuz içinde metot referansı tutan yeni bir veri yapısı olan temsilciler (delegates) ve temsilcilerle yakından ilgisi olan olaylar (events) olacaktır. Üçüncü konu, C# önişlemci komutları ve bunların kullanımı olacaktır. Dördüncü konu, programlama dünyasının en önemli konusu haline gelen göstericilerin (pointers) C# ile nasıl kullanılabileceği, göstericilerin yararları ve zararları olacaktır. Ardından çalışma zamanı tip bilgilerinin elde edilmesi olan yansıma ve nitelik gibi kavramları System.Reflection isim alanı ile beraber inceleyeceğiz ve son olarak C# 4.0 ile birlikte gelen yeni dinamiklik özelliklerini inceleyerek bölümü bitireceğiz.

İstisnai Durum Yönetimi (Exception Handling)

Programlama diline ne kadar hakim olursanız olun uygulama geliştirirken sizin dışınızda hatalara yol açabilecek çeşitli etkenler vardır. İşin kötüsü sizin dışınızda gelişen bu hataların çoğu çalışma zamanında ortaya çıkmaktadır. Dolayısıyla programda çıkabilecek olası bütün hatalara karşı tek tek önlem almak gerçekten çok zahmetli bir iştir. Hele istisnai durum yönetimi gibi hata yakalamayı otomatik hale getiren bir sistemin olmadığı dillerde bu iş daha da zordur. Ancak bütün modern dillerde (C++, JAVA, C#) çalışma zamanında meydana gelebilecek bu tür hataları yakalamak için istisnai durum yakalama mekanizması vardır. Bu mekanizma çalışma zamanında oluşacak hataları sistematik bir şekilde yakalamamızı ve bu hatalara müdahale edebilmemizi sağlar. Zira, C ve benzeri dillerde çalışma zamanında hataları yakalamak için herhangi bir mekanizma yoktur. Bu dillerde bu tür hataları bulmak ve önlem almak için fonksiyonların geri dönüş değerleri kontrol edilirdi. Her bir geri dönüş değeri farklı hata kaynağını temsil etmekteydi. Dolayısıyla fonksiyon her çağrıldığında fonksiyonun geri dönüş değeri kontrol edilerek hata durumunda ya program sonlandırılırdı ya da başka işlemler yapılırdı. C#, C++ ve JAVA dillerinde ise, tanımladığınız istisnai durum sınıfları ile bu hataları yakalamak daha kolay hale getirilmiştir.

İstisnai Durum Sınıfları

Çalışma zamanında beklenmedik bir hatanın oluşumu sonrasında oluşturulan nesnelere istisnai durum sınıf nesneleri denilmektedir. Buradan bütün istisnai durumların aslında birer sınıf ile temsil edildiğini anlıyoruz. Bu sınıflar oluşan hata ile ilgili çeşitli bilgileri tutmak için gerekli özelliklere sahiptir. .NET sınıf kütüphanesinde çok sık oluşabilecek hatalara karşı çeşitli istisnai durum sınıfları tasarlanmıştır. Bu hazır sınıflar genellikle bize yetecek ancak istersek kendi istisnai durum sınıflarımızı da oluşturabiliriz.

.NET sınıf kütüphanesindeki istisnai durum sınıflarının hiyerarşisini biraz inceleyelim. Öncelikle diğer bütün türlerde olduğu gibi istisnai durum sınıfları da Object sınıfından türetilmiştir. Object sınıfından türeyen Exception sınıfı istisnai durum sınıf hiyerarşisinin en tepesinde bulunur. System isim alanında bulunan Exception sınıfı oluşan hata ile ilgili çok genel bir bilgi verdiği için genellikle kullanılmaz. .NET çalışma zamanı (run time) tarafından belirlenen hata durumları, **Exception** sınıfından türemiş olan SystemException sınıfı ile temsil edilir. Sıklıkla rastlanabilecek hatalardan olan diğer hatalar ise **SystemException** sınıfından türemiş olan ArgumentException, StackOverFlowException, ArithmeticException gibi istisnai durum sınıfları ile temsil edilir. Yine SystemException sınıfından türemiş olan IOException sınıfı da daha önce gördüğümüz girdi/çıktı işlemleri sırasında meydana gelebilecek hataları temsil etmek için kullanılır. Örneğin, FileStream nesnesine parametre olarak verilen bir dosyanın bulunamaması durumunda IOException sınıfından türemiş olan FileNotFoundException sınıfı kullanılır. Diğer bir önemli istisnai durum sınıfı ise IndexOutOfRangeException sınıfıdır. Bu sınıf bir dizinin sınırları aşıldığında fırlatılır. İstisnai durum yakalama sırasında oluşan hatalardan sonra oluşturulan nesneler için "FileNotFoundException" hatası fırlatılır şeklinde bir deyim kullanacağız. Bir hatanın fırlatılması, çalışma zamanında ilgili hatayı temsil eden sınıf türünden bir nesnenin oluşturulması anlamına gelir.

Bahsedilen bütün bu hazır istisnai durum sınıflarının yanında programcı isterse kendine özel istisnai durum sınıfları da oluşturabilir. Bunun için ApplicationException sınıfı kullanılır. Programcının bildirdiği istisnai durum sınıfları, genellikle Exception sınıfından türetilmiş olan ApplicationException sınıfından türetilerek bildirilir. .NET sınıf kütüphanesinde yüzlerce istisnai durum sınıfı vardır. Bunların tamamını bu kitapta anlatmak mümkün değildir. Ancak konunun ilerleyen kısımlarında en çok kullanılan sınıflar ile temel sınıf olan Exception sınıfı ayrıntılarıyla anlatılacaktır.

İstisnai Durum Yakalama

İstisnai durum yakalama mekanizmasını sağlayan 4 tane anahtar sözcük vardır. Bunlar try, catch, throw ve finally anahtar sözcükleridir. try, catch ve finally anahtar sözcükleri bir kod bloğunu temsil ederken throw bir hatanın fırlatılmasını sağlar. Şimdi bu anahtar sözcüklerinin kısaca ne işe yaradıklarına bakalım.

try: Kaynak koddaki hataların bulunacağı kod bloklarını temsil eder. Yani try anahtar sözcüğü ile açılan blokta istisnai durum meydana gelirse ilgili hatayı temsil eden istisnai durum sınıfı fırlatılır.

catch: try bloğundan fırlatılan hatanın yakalanıp gelen hataya göre işlemler yapan kod bloğunu temsil eder.

finally: try ve catch bloklarında açılan kaynakların kapatılma işlemleri burada yapılır. Örneğin açılan bir dosyanın kapatılması gibi. Finally bloğundaki kodlar hata ister oluşsun ister oluşmasın mutlaka çalıştırılır. Finally bloğunu belirtmek zorunda değiliz. Ancak try ve catch blokları istisnai durum yakalama mekanizmasının çalışması için mutlaka olmalıdır.

throw: try bloğu içinde belirli bir hata olduğunda ilgili hata sınıfı nesnesini fırlatmak için kullanılır. Eğer try bloğunda çağrılan bir metot ya da oluşturulan bir nesne zaten çeşitli durumlarda hata fırlatıyorsa try bloğunda throw anahtar sözcüğünü kullanmaya gerek olmayabilir.

En genel hali ile C# dilinde istisnai durum yakalama mekanizmasının nasıl kurulduğu aşağıda verilmiştir.

```
try
{
      //Hatanın fırlatıldığı bölüm
}

catch(HataSınıfı a)
{
      //Hatanın elde edilmesi
}
finally
{
      //Kaynakların temizlenmesi
}
```

Önce yukarıdaki sistemin nasıl çalıştığına bakalım. Sonra basit bir örnekle devam edeceğiz. try bloğunda programımızın normal kodu bulunmaktadır. Yani özel bir kod parçası değildir. Eğer herhangi bir hata meydana gelmez ise kodlar normal olarak çalıştırılmaya devam eder. try bloğu bittikten sonrada herhangi bir hata fırlatılmadığı için akış finally bloğuna gelir ve buradaki kodlar çalıştırılır. Eğer try bloğunda herhangi bir hata ile karşılaşılırsa programın akışı catch bloğuna gelir. Fırlatılan hatanın türüne göre farklı catch blokları da olabilir. Catch bloğu çalıştırıldıktan sonra son olarak finally bloğu çalıştırılır ve programın akışı finally bloğunun sonundan devam eder. Bu noktada bilinmesi gereken en önemli noktalar, finally bloğunun isteğe bağlı olması, istediğimiz kadar catch bloklarını oluşturabilmemiz ve hatayı yakalamak için catch ve try bloklarını mutlaka bildirmemiz gerektiğidir. Try bloğundan sonra mutlaka catch ya da finally bloğunun gelmesi gerekir. Eğer try bloğundan sonra finally bloğu geliyorsa try bloğundaki hata yakalanmaz, dolayısıyla mantıklı bir kullanım değildir.

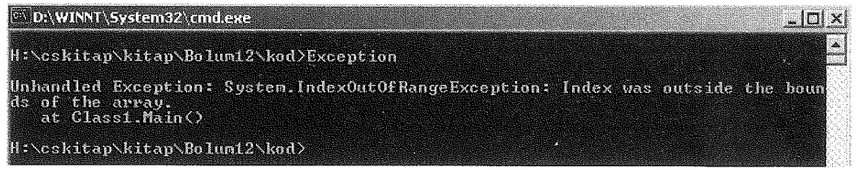
try ve catch blokları ard arda gelmelidir.

Şimdi istisnai durum yakalama ile ilgili bir örnek verelim. Program yazarken belki de en sık karşılaştığımız hatalardan biri, bir dizinin sınırlarını aşmaktır. Bir dizinin olmayan elemanına ulaşmağa çalıştığımızda .NET, çalışma zamanında IndexOutOfRangeException istisnai durum nesnesi fırlatır.

Örneğin aşağıdaki programı yazıp çalıştırın,

```
using System;
class Class1
{
      static void Main()
      {
          int[ ] a = new int[ 5] ;

          a[ 10] = 50;
      }
}
```

Programın normal bir şekilde derlenir ancak komut satırından programı çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.



Bu program, bir çalışma zamanı hatasından dolayı program sonlandırılmıştır. Fakat bu tür hataları kendimiz try ve catch blokları ile yakalayıp kullanıcıyı değişik şekillerde uyarabiliriz. Ya da catch bloğunda programı sonlandırmadan bazı metotları çağırıp değişik işlemler yapabiliriz.

Şimdi de bu programın ikinci versiyonunu yazalım.

```
using System;
class Class1
{
      static void Main()
      {
          int[ ] a = new int[ 5] ;

          try
          {
              a[ 10] = 50;
          }
          catch
          {
              Console.WriteLine("İstisnai durum oluştu");
          }

          finally
          {
```

```
            Console.Write("Finally bloğu");
        }
    }
}
```

Programı çalıştırdığımızda ekrana

```
İstisnai durum oluştu
Finally bloğu
```

yazacaktır.

Bu programdaki try bloğunda bulunan

```
a[ 10] = 50;
```

satırına gelindiğinde .NET tarafından IndexOutOfRangeException hatası fırlatılır. catch bloğu ise bu hatanın fırlatılması sonrasında ekrana bir uyarı mesajı vermektedir. Ancak catch bloğunda try bloğundan fırlatılan hata sınıfı ile ilgili bir bilgi yoktur. Yani istisnai durumun kaynağını belirtecek herhangi bir nesne yoktur.

Gerçek uygulamalarda try blokları yukarıda olduğu gibi tek satır değildir ve genellikle birden fazla istisnai durum oluşur. Bu durumda her istisnai durum için farklı catch blokları çalıştırılması istenir. Bu yüzden catch bloğunda istisnai durum sınıfı ile ilgili parametrik bir bilgi verilir. catch bloğunun bu şekildeki kullanımına aşağıda bir örnek verilmiştir.

Yukarıdaki programın üçüncü versiyonunu da aşağıdaki gibi değiştirin. Burada değişen tek şey catch bloğunda yakalanacak hatanın türünün de belirtilmesidir. Ayrıca finally bloğu şu an için çok önemli olmadığı için kaldırılmıştır.

```
using System;

class Class1
{
    static void Main()
    {
        int[ ] a = new int[ 5] ;

        try
        {
            a[ 10] = 50;
        }
        catch(IndexOutOfRangeException excp)
        {
            Console.WriteLine("Hata Mesajı : " +
                                        excp.Message);
        }
    }
}
```


Programı derleyip çalıştırdığınızda ekrana

```
Hata Mesajı : Index was outside the bounds of the array
```

yazacaktır.

Programdan da görüldüğü üzere catch bloğunda `IndexOutOfRangeException` hata sınıfının beklendiği belirtilmektedir. Dolayısıyla bu programdaki catch bloğu sadece `IndexOutOfRangeException` hatası fırlatıldığı zaman çalıştırılacaktır. Örneğin aşağıdaki programda bildirilen bir catch bloğu olmasına rağmen fırlatılan hata farklı bir sınıfı temsil ettiği için çalışma zamanında hata yakalanamamıştır, dolayısıyla catch bloğu çalıştırılmamıştır.

```
using System;

class Class1
{
    static void Main()
    {
        int a = 10;
        int b = 0;
        int c;

        try
        {
            c = a / b;
        }
        catch(IndexOutOfRangeException excp)
        {
            Console.WriteLine("Hata Mesajı : " +
                                        excp.Message);
        }
    }
}
```

Bu programdaki try bloğunda c = a / b; deyimi ile fırlatılan hata `IndexOutOfRangeException` sınıfını temsil etmediği için programdaki catch bloğu çalıştırılmamıştır. Bir sayıyı 0'a bölmeye çalıştığımızda .NET DivideByZeroException sınıfı ile temsil edilen hata fırlatır. Bu yüzden bu programdaki catch bloğunu

```
catch(DivideByZeroException excp)
{
    Console.WriteLine("Hata Mesajı : " + excp.Message);
}
```

şeklinde değiştirirsek çalışma zamanındaki bu istisnai durumu yakalamış oluruz. Bir önceki programdaki

```
excp.Message
```

ifadesi kafanızı karıştırmış olabilir. Ancak şimdi bu konuyu açıklığa kavuşturacağız. Konunun başında belirtildiği gibi bütün istisnai durumları temsil eden sınıflar, System isim alanında bulunan Exception sınıfından türetilmiştir. Dolayısıyla Exception sınıfında bulunan bütün üye elemanlar kendi oluşturacağımız sınıflar da dahil bütün sınıflara kalıtım yolu ile geçmiştir. Şimdi Exception sınıfının önemli üye elemanlarını inceleyelim.

Message: String türünden olan bu özellik fırlatılan hata ile ilgili açıklayıcı bir mesaj saklar. Message özelliği sanal bir özellik olduğu için çok biçimlilik desteklenir.

Source: Bu özellik catch bloğunda yakalanan istisnai durum nesnesinin gönderildiği uygulamanın ya da sınıfın adıdır. Source özelliği de string türündendir ve sanaldır.

HelpLink: Bu özellikle fırlatılan hata ile ilgili yardım dosyasının yol bilgisi saklanır. Dolayısıyla hata fırlatıldığında istersek catch bloğunda bu özellikten faydalanarak yardım dosyasını açtırabiliriz.

StackTrace: İstisnai durumun fırlatıldığı metot ve program ile ilgili bilgi içerir. Bu özellikte string türündendir ve aynı zamanda sanaldır. Bu özelliğin doğaldır ki sadece get bloğu tanımlanmıştır.

TargetSite: İstisnai durumu fırlatan metot ile ilgili bilgiler veren yansıma konusunda göreceğimiz MethodBase sınıfı türünden bir özelliktir.

InnerException: Bu özellik eğer catch bloğu içerisinden bir hata fırlatılırsa catch bloğuna gelinmesine yol açan istisnai durumun Exception nesnesidir.

Exception sınıfının ToString() metodu ise ilgili istisnai durum nesnesinin string şeklindeki gösterimini döndürür. Bu metodu ve yukarıda anlatılan özellikleri test etmek okuyucuya bırakılmıştır.

İstisnai durum yakalama ile ilgili diğer önemli bir noktada metot gövdelerinde oluşabilecek istisnai durumların yakalanması ile ilgilidir. Örneğin aşağıdaki statik Metot1() metodunda dizinin sınırları aşılmıştır. Ancak bu metot try bloğu içinde çağrıldığında catch bloğunun metodun gövdesinde olmaması hatanın yakalanmayacağı anlamına gelmez. Bu yüzden istisnai durumu oluşturacak metodun çağrıldığı yerde catch bloğu kullanılıp hata yakalanabilir.

```
using System;

class ExceptionTest
{
    public static void Main()
    {
        try
        {
            Metot1();
        }

        catch(IndexOutOfRangeException excp)
        {
            Console.Write(excp.Message);
        }
    }

    public static void Metot1()
    {
        int[] a = new int[5];
        a[10] = 5;
    }
}
```

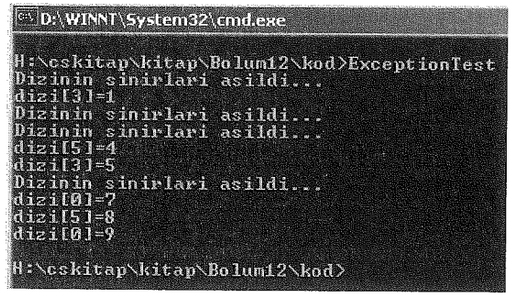

Metot1() çağrıldığı anda eğer Metot1() metodunun çalışması sırasında istisnai bir durum oluşursa önce metodun gövdesinde ilgili hatayı yakalayacak bir catch bloğunun olup olmadığı kontrol edilir. Eğer Metot1()'in gövdesinde catch bloğu yoksa yukarıdaki programda da olduğu gibi Metot1()'in çağrıldığı Main() metodu içindeki catch bloğu çalıştırılır.

Görüldüğü gibi istisnai durum yakalama mekanizması ile programları sonlandırmadan istenilen kod blokları çalıştırılabilir. Bu duruma en güzel örnek döngüler içinde try ve catch bloklarını kullanılırken görebiliriz. Aşağıdaki programda bir dizinin elemanları rastgele değiştiriliyor. Rastgele seçilen sayılar değiştirilecek elemanın hangisi olduğunu belirliyor. Dolayısıyla rastgele olan bu indeks numarası dizinin sınırlarını aşmış olabilir. Eğer dizinin sınırı aşıldı ise catch bloğunda hata yakalanıp ilgili mesaj verildikten sonra döngüye devam ediliyor.

```
using System;

class ExceptionTest
{
    public static void Main()
    {
        int[] dizi = new int[6];
        Random rnd = new Random();
        int sayi;

        for(int i=0; i < 10; ++i)
        {
            try
            {
                sayi = rnd.Next(10);
```

```
                dizi[ sayi] = i;
                Console.WriteLine("dizi[ { 0} ] ={ 1} ",sayi,i);
            }

            catch(IndexOutOfRangeException)
            {
                Console.WriteLine("Dizinin sınırları
                                        aşıldı...");
            }
        }
    }
}
```

Programı derleyip çalıştırdığımızda aşağıdaki ekran görüntüsüne benzer bir görüntü elde etmelisiniz.



try bloğundan sonra mutlaka ya catch ya da finally bloğunun gelmesi gerekir. Aksi halde derleme hatası alınır.

throw Anahtar Sözcüğü

Şu ana kadar gördüğümüz istisnai durum sınıflarının fırlatılması .NET çalışma zamanında otomatik olarak yapılmaktaydı. Ancak bazı durumlarda try bloğu içinde özel durumlarda throw anahtar sözcüğünü kullanarak kendimiz istisnai durum nesnelerini fırlatmak isteyebiliriz. throw anahtar sözcüğünün kullanımı genellikle kendi tanımladığımız istisnai durum sınıflarının fırlatılması sırasında görülür. Ama nadiren de olsa .NET sınıf kütüphanesinde bulunan istisnai durum sınıflarının fırlatılması sırasında da kullanılır. Aslına bakarsanız şu ana kadar throw anahtar sözcüğünü kullanmamamıza rağmen, sınıf kütüphanelerini geliştirenler istisna durumları göz önünde bulundurarak throw anahtar sözcüğünü kullanmışlardır.

throw anahtar sözcüğü kullanılarak bir istisnai durum sınıfı türünden nesne fırlatılır. Örneğin try bloğu içerisinde bir IndexOutOfRangeException hatasını fırlatmak için

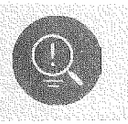
```
throw new IndexOutOfRangeException("Dizinin sınırları
                                        aşıldı...");
```


yazılır. Burada new anahtar sözcüğünü throw ile beraber kullanmamız şart değildir. İstisnai durum nesnesini ayrıca oluşturup throw ile aşağıdaki gibi de fırlatabiliriz.

```
IndexOutOfRangeException e = new IndexOutOfRangeException("Dizi
                                        sınırları aşıldı...");

e.HelpLink = "http://www.csharpnedir.com/exception.html";

throw e;
```

Bu iki kullanımda da IndexOutOfRangeException sınıfının yapıcı metodu ile hatanın mesajı varsayılan mesaj ile değiştirilmiş oluyor. Dolayısıyla catch bloğunda bu hata yakalandığı zaman istisnai durum sınıfı ile gönderilen nesnenin Message özelliği ile hata hakkında kullanıcıya bilgi verilir.

throw anahtar sözcüğünü try blokları içerisinde kullanmak zorunlu değildir. Ancak try blokları dışında throw ile fırlatılan istisnai durumlar herhangi bir catch bloğu olmadığı için yakalanamaz.



Şimdi throw anahtar sözcüğünü kullanarak .NET sınıf kütüphanesinde tanımlanan istisnai durum sınıfı nesnelerinin nasıl fırlatıldığını inceleyelim. Aşağıdaki programda Sayılar sınıfındaki mDeger değişkenine yapıcı metot ile 100'den büyük bir değer atandığında ya da Deger özelliğine 100'den büyük bir değer atandığında System isim alanında bulunan OverflowException hatası fırlatılıyor.

```
using System;

class Sayilar
{
    private int mDeger;

    public Sayilar(int mDeger)
    {
        if(mDeger > 100)
            throw new OverflowException("Deger 100' den
                                        büyük olamaz!");
        else
            this.mDeger = mDeger;
    }

    public int Deger
    {
        get
        {
            return mDeger;
```

```
        }
        set
        {
            if(value > 100)
                throw new OverflowException("Deger
                                100' den büyük olamaz!");
            else
                this.mDeger = value;
        }
    }
}

class ExceptionTest
{
    public static void Main()
    {
        try
        {
            Sayilar s = new Sayilar(500);
        }

        catch(OverflowException excp)
        {
            Console.WriteLine(excp.Message);
        }
    }
}
```

Bu programı derlediğimizde ekrana

```
Deger 100' den büyük olamaz!
```

yazacaktır. Çünkü Sayılar sınıfının yapıcı metodu 100'den büyük bir değer ile çağrılmıştır. Görüldüğü gibi Main() metodu içinde bulunan try bloğunda throw anahtar sözcüğü kullanılmamasına rağmen catch bloğunda OverflowException hatası yakalanmıştır.

OverflowException sınıfı aritmetik işlemlerde bir nesne tipinin tutabileceği maksimum değer aşıldığı zaman fırlatılır. Bu örnekte OverflowException yerine IndexOutOfRangeException sınıfı da kullanılabilirdi.

Çoğu durumda OverflowException gibi özel anlamı olan bu istisnai durum sınıflarını kendi sınıflarımız içinde kullanmak yerine kendi istisnai durum sınıflarımızı ApplicationException sınıfından türeterek oluşturacağız.

İstisnai durum sınıfları adından da anlaşılacağı üzere gerçekten sadece istisnai durumlarda fırlatılmalıdır. Eğer programın güvenliğini klasik yöntemlerle sağlayabiliyorsak istisnai bir durumun oluşmasını engelleyebiliriz. Örneğin kullanıcının sayı girmesi gerektiği bir durumda yazı (string) girdiğinde yanlış veri girildi diye uyarı verip sayının tekrar istenilmesi daha uygundur. Bütün profesyonel uygulamalarda yanlış veri girildiğinde programı sonlandırmak yerine yanlış veri girildiğini uyarı şeklinde verip verinin tekrar girilmesi istenir. Bu yüzden bu tür durumlarda istisnai durum sınıf nesnelerini fırlatmamak profesyonel programcılık açısından önemlidir.

Birden Fazla catch Bloğu Kullanma

Gerçek uygulamalarda try blokları içinde fırlatılan hatalar tek türden olmayabilir. Bir try bloğunda birden fazla istisnai durum oluşma ihtimali varsa her bir istisnai durumu yakalamak için ayrı birer catch bloğu tanımlanır. Her bir catch bloğunda yakalanmak istenen istisnai durum sınıfının tipi belirtilir. Eğer istisnai durum sınıfı herhangi bir catch bloğundaki türe uygun bir şekilde atanamıyorsa istisnai durum yakalanamaz dolayısıyla program sonlandırılır.

Aşağıdaki programda kullanıcıdan 1-100 arasında bir sayı girilmesi bekleniyor. Kullanıcıdan ReadLine() metodu ile aldığımız değer string türünden olacağı için bu değeri int türüne çevirmek için Int32 yapısının Parse() metodunu kullanıyoruz. Ancak Parse() metodu kendisine parametre olarak gönderilen yazıyı int türüne çeviremediği zaman `FormatException` hatası fırlatır. Bu yüzden bu hatayı yakalamak için FormatException nesnesi bekleyen bir catch bloğu yazılmıştır. Aynı zamanda kullanıcının gireceği değerin de 1 ile 100 arasında olmasını istediğimiz için eğer girilen değer bu aralıkta değilse OverflowException hatasını da biz fırlatıyoruz.

```
using System;

class ExceptionTest
{
    public static void Main()
    {
        int i;
        string str;

        try
        {
            Console.Write("1-100 arasında sayı girin : ");

            str = Console.ReadLine();

            if(i > 100 || i < 1)
                throw new OverflowException("Yanlış değer!");
            i = Int32.Parse(str);
        }

        catch(OverflowException excp)
        {
```

```
            Console.WriteLine(excp.Message);
        }

        catch(FormatException excp)
        {
            Console.WriteLine(excp.Message);
        }
    }
}
```

Bu programda sayının 1-100 arasında olup olmadığını if-else blokları ile de kontrol edebilirdik. Gerçekte bu daha doğru bir yöntem olabilir. Bu örnek sadece konunun daha iyi bir şekilde anlaşılması açısından verilmiştir. Ayrıca bu örnek ile istisnai durum yakalama mekanizmasının ne kadar esnek bir sistem olduğunu da görmüş oluyoruz.

Yukarıdaki programda bulunan catch bloklarından sonra finally bloğu yazılmış olsaydı hem OverflowException istisnai durumu olduğunda hem de FromatException istisnai durumu olduğunda finally bloğundaki kodlar çalıştırılırdı.

Şimdi, birden fazla catch bloğunun kullanılması ile ilgili önemli noktaları maddeler halinde inceleyelim.

1. Eğer try bloğundan fırlatılan istisnai durum nesneleri catch bloklarındaki herhangi bir tür ile uyumlu değilse fırlatılan hata yakalanamayacağı için program istisnai durum oluştuğunda sonlandırılır. Örneğin, catch bloklarını

```
catch(OverflowException excp)
{
    Console.WriteLine(excp.Message);
}

catch(IndexOutOfRangeException excp)
{
    Console.WriteLine(excp.Message);
}
```

şeklinde değiştirdiğimizde eğer kullanıcı "56s5A" şeklinde bir yazı girerse Parse() metodu tarafından gönderilen FormatException hatası yakalanamayacaktır. Dolayısıyla programın çalışması sonlandırılacaktır.

2. Eğer try bloğundan gelen istisnai durum sınıfının türü catch bloklarındaki türlerden birinden türemiş ise yine bu catch bloğu tarafından yakalanabilir. Ancak catch bloklarında ilk aranan şart fırlatılan istisnai durum sınıfının türü ile aynı olmasıdır. Aşağıdaki ikinci catch bloğunda throw ile fırlatılmış bütün istisnai durum nesneleri yakalanabilmektedir. Çünkü bütün istisnai durum sınıfları Exception sınıfından türe-

tilmiştir. Hatırlarsanız temel sınıf referansına türemiş sınıf referanslarını atayabiliyorduk. İstisnai durum sınıfları için bu kural aynen geçerlidir.

```
catch(OverflowException excp)
{
    Console.WriteLine(excp.Message);
}

catch(Exception excp)
{
    Console.WriteLine(excp.Message);
}
```

Eğer ilgili try bloğundan fırlatılan istisnai durum sınıf nesnesi OverflowException türünden ise birinci catch bloğu çalıştırılır. Tabi sonra da ikinci blok çalıştırılmaz. Her ne kadar OverflowException referansları Exception referansına atanabilse de OverflowException hatası fırlatıldığında birinci catch bloğu çalıştırılır. Çünkü tam bir tür uyumluluğu vardır.

3. Eğer istisnai durumun türünden bağımsız bir hata yakalama yapmak istiyorsak catch bloğunu herhangi bir tür belirtmeden de kullanabiliriz. Örneğin

```
catch(OverflowException excp)
{
    Console.WriteLine(excp.Message);
}

catch
{
    Console.WriteLine("İstisnai durum oluştu!");
}
```

Bu kullanımda OverflowException hatası dışındaki hataların fırlatılması durumunda ikinci catch bloğu çalıştırılır. Ancak bu sefer hatanın kaynağı bilinmediği için fırlatılan hata ne olursa olsun aynı uyarıyı vermek zorundayız. Bu durumda sadece genel bir uyarı vermekten başka yapacak bir şey yoktur.

catch bloğunu bu şekilde kullanmak hatanın kaynağını açıkça belirtmediği için profesyonel uygulamalarda bu tür bir kullanımdan kaçınmamız gerekir.

4. İki catch bloğu arasına herhangi bir kod parçası giremez. Bu derleme zamanında hataya yol açar; örneğin aşağıdaki gibi bir kullanım geçersizdir.

```
try
{
```

```
        // Programın parçası
}
catch(OverflowException excp)
{
        Console.WriteLine(excp.Message);
}

int a = 5; // hatalı kullanım

catch
{
        Console.WriteLine("İstisnai durum oluştu!");
}
```

Bu şekildeki bir kullanımda derleyici ikinci catch bloğuna ait olan try bloğunu aramaktadır. Ancak try bloğu bir tane olduğu için derleme zamanında hata alınır. Bütün istisnai durumların Exception sınıfından türediğini göz önünde bulundurursak

```
catch
{
}
```

ile

```
catch(Exception excp)
{
}
```

blokları arasındaki farkı mutlaka merak etmişsinizdir. Bu iki catch bloğunu tamamı C# ile yazılmış bir programda kullanırsak her zaman ikinci catch bloğu çalıştırılacaktır. Başka dillerde yazılmış program parçacıklarının metotlarını C# programının içinden kullanabiliriz. Örneğin bir COM nesnesine ilişkin metot ya da sistem fonksiyonu istisnai bir durum oluşturduğunda birinci catch bloğu çalıştırılır. COM parçacıklarının yazılması bu kitabın konuları arasında olmadığı için burada değinilmeyecektir.

5. try ve catch blokları, blok faaliyet alanı kapsamına girmektedir. Dolayısıyla bu bloklarda tanımlanan nesneler blok dışında kullanılamaz. Aşağıdaki iki catch bloğunda da excp değişkeninin kullanılabilmesi bu yüzdendir.

```
catch(OverflowException excp)
{
        Console.WriteLine(excp.Message);
}

catch(Exception excp)
{
        Console.WriteLine(excp.Message);
}
```

6. Aynı türden istisnai durum nesneleri alan catch blokları yazılamaz. Örneğin aşağı-

daki gibi bir kullanım derleme zamanında hataya yol açacaktır.

```
catch(Exception excp)
{
        Console.WriteLine(excp.Message);
}

catch(Exception excp)
{
        Console.WriteLine(excp.Message);
}
```

İç içe Geçmiş (nested) try Blokları

Her try bloğunda fırlatılan hata nesneleri ancak bir catch bloğu tarafından yakalanabilir. İç içe geçmiş try bloklarını kullanarak bir try bloğundan fırlatılacak hataya ilişkin catch bloklarının sayısını artırabiliriz. İç içe geçmiş try bloklarında da her fırlatılan hata için sadece bir catch bloğu çalıştırılır fakat bu çalıştırılacak catch bloğu birkaç ihtimal arasından seçilir.

Aşağıda bir try bloğu içinde farklı bir try bloğunun açıldığını görüyorsunuz. Her try bloğuna ilişkin catch blokları da mevcuttur. Bu yapının değişik noktalarından hata fırlatılması durumunda neler olabileceğini inceleyelim.

```
try
{
        //A
        try
        {
            //B
        }
        catch
        {
            //C
            //İçteki catch bloğu
        }
        finally
        {
            //İçteki finally bloğu
        }
            //D
}
catch
{
        //Dıştaki catch bloğu
}
finally
{
        //Dıştaki finally bloğu
}
```

a. En dıştaki try bloğunun içinde fakat içteki try bloğunun dışında olan A ve D böl-

gelerinde herhangi bir istisnai durum meydana gelirse dıştaki catch ve dıştaki finally bloğu çalıştırılır.

A ve D bölgelerinde istisnai durum meydana gelmese bile en dıştaki finally bloğunun çalıştırılacağını unutmayın.

b. Bizi asıl ilgilendiren nokta B bölgesinde oluşacak bir istisnai durumdur. Eğer bu bölgede bir istisnai durum oluşursa öncelikle içteki catch bloğundan fırlatılan hataya uygun olanı çalıştırılır. Sonra da içteki finally bloğu çalıştırılır. Eğer B bölgesinden fırlatılan hatayı yakalayacak uygun bir catch bloğu bulunamaz ise dıştaki try bloğuna ait catch blokları kontrol edilir. Eğer burada uygun bir catch bloğu bulunursa bu blok çalıştırılır, uygun bir blok bulunmaz ise her zamanki gibi yakalanmayan istisnai durumdan dolayı program sonlandırılır. Bu durumda her iki finally bloğu da çalıştırılacaktır. İç içe geçmiş iki try bloğundaki kuralları 3 tane iç içe geçmiş try bloğu için de genelleştirebiliriz. Yani en içteki try bloğundan fırlatılan hatanın yakalanacağı catch bloğu en içteki catch blokları arasında aranır. Bulunmadığı zaman sırayla bir üst seviyedeki catch blokları kontrol edilir.

c. Diğer bir ilginç nokta da C bölgesinden fırlatılacak hatalardır. Bir catch bloğundan yeni bir istisnai durum oluşturmak aslında çok tavsiye edilmez ama nadiren de olsa catch bloklarında da istisnai durumlar oluşabilir. C bölgesinde istisnai durum oluştuğunda derleyici bu istisnai durumun sanki en dıştaki try bölümünden fırlatılmış gibi davranır. Yani C bölgesinden fırlatılan hataları yakalayacak catch bloğu en dıştaki catch blokları arasından seçilir.

catch blokları içerisinden istisnai durum sınıf nesnelerini fırlatmak geçerli olmasına rağmen çok sık rastlanan bir durum değildir.

Bir catch bloğunda istisnai durum sınıfı nesnesi fırlatma daha çok bir hatanın dıştaki try bloklarında da yakalanabilmesini sağlamak için kullanılır. Bunun sebebi bir catch bloğundan fırlatılan hata nesnesinin bir üst seviyedeki try bloğuna ilişkin catch bloklarında yakalanmasıdır.

Standart İstisnai Durum Sınıfları

C#'ta istisnai durum sınıflarını iki grupta inceleyebiliriz. Birincisi şu ana kadar gördüğümüz .NET çalışma zamanı (CLR) tarafından fırlatılanlar diğeri ise programcının ApplicationException sınıfından türettiği sınıflardır. C#'taki istisnai durumları temsil eden sınıfların çoğu System isim alanında bulunur. Bu sınıflar SystemException sınıfından türetilmiştir. Aşağıda CLR tarafından fırlatılan çok sık kullanılan bazı standart istisnai durum sınıfları ve açıklamalarını görüyorsunuz. .NET kütüphanesinde bulunan diğer istisnai durum sınıflarını MSDN yardım dokümanlarından öğrenebilirsiniz.

`System.OutOfMemoryException` – Programın çalıştırılması için yeterli bellek miktarı kalmadı ise fırlatılan sınıf nesnesinin türüdür.

`System.StackOverflowException` – Stack bellek bölgesinin birden fazla metot için kullanılması durumunda fırlatılır. Genellikle öz yinelemeli algoritmalar da bu hata sınıfı fırlatılabilir.

`System.NullReferenceException` – İçinde null değeri olan(nesne referansı taşımayan) bir nesne referansı üzerinden sınıfın üye elemanlarına erişmeye çalışıldığında fırlatılır.

`System.OverflowException` – Aritmetik işlemlerde değişkenlerin değerlerinde taşma olduğu zaman fırlatılır.

`System.InvalidCastException` – Çalışma zamanında geçersiz tür dönüşümü yapıldığında fırlatılır.

`System.IndexOutOfRangeException` – Bir dizinin indeks sınırları aşıldığında fırlatılır. Daha önce bu sınıfla ilgili örnekler yapmıştık.

`System.ArrayTypeMismatchException` – Bir dizinin herhangi bir elemanına geçersiz türden bir değer atandığı zaman fırlatılır.

`System.DivideByZeroException` – Sıfıra bölme yapıldığı zaman fırlatılır.

`System.ArithmeticException` – DivideByZeroException ve OverflowException sınıflarının temel sınıfıdır. Bu sınıf ile bahsedilen her iki hata durumu da yakalanabilir. Genel olarak aritmetik işlemler sonucunda oluşan istisnai durumlarda fırlatılır.

System.FormatException – Metotlara yanlış formatta parametre gönderildiğinde fırlatılır. Özellikle string işlemlerinde rastlanılır.

İstisnai Durum Sınıfları Oluşturmak

Her ne kadar standart istisnai durum sınıfları çok geniş de olsa bazen kendi tasarladığımız özel sınıflar için değişik istisnai durum sınıfları oluşturmak isteyebiliriz. Özellikle büyük çapta sınıf kütüphaneleri geliştirdiğimizde buna ihtiyacımız olacaktır.

Programcının bildireceği istisnai durum sınıfları ApplicationException sınıfından türetilmelidir. Bu sınıf zaten Exception sınıfından türediği için konunun başından anlatılan Exception sınıfının üye elemanları kalıtım yolu ile kendi özel sınıfımıza aktarılacaktır. İstersek bu üye elemanları devre dışı bırakıp kendimize göre düzenleyebiliriz.

Elbette istersek oluşturduğumuz yeni sınıfa yeni özellikler de ekleyebiliriz. Zaten çoğu durumda yeni özellikler eklemek durumunda kalacağız.

Aşağıda verilen örnek gerçek uygulamalarımızda istisnai durum yakalama mekanizmasından ne şekilde faydalanabileceğimizi yakından gösterecektir. Şimdi isterseniz aşağıdaki örneği açıklayalım. Bu örnekte Metrik isimli bir sınıf bildirilmiştir. Metrik sınıfı uzunluk ölçü birimini Metre ve Santim olarak tutacak bir veri modelidir. Santim değerleri en fazla 99 olacak şekilde ayarlanacaktır. Çünkü 1 m 120 santim yerine 2 metre 20 santim kullanılması istenmektedir. Santim değerini her değiştiren metot ya da özellik değiştirilmek istenen değeri 99 ile karşılaştırıyor. Eğer bu değer 99'dan büyükse bizim özel tanımladığımız HatalıMetrikİstisnası sınıf nesnesi throw anahtar sözcüğü kullanılarak fırlatılıyor. throw anahtar sözcüğünün kullanımı için hatırlarsanız herhangi özel bir blok açmaya gerek yoktu. Yeni bildirdiğimiz HatalıMetrikİstisnası istisnai durum sınıfı Metrik sınıfına ilişkin olduğu için bu sınıf içerisinde aşağıdaki gibi bildirilmiştir.

```
public class HatalıMetrikİstisnası : ApplicationException
{

}
```

Bu yüzden bu istisnai durum sınıfına erişmek için

```
Metrik.HatalıMetrikİstisnası e
```

şeklinde bir kullanım gereklidir.

`Metrik.HatalıMetrikİstisnası` sınıfında ApplicationException sınıfından kalıtım yolu ile gelen özelliklerin yanı sıra iki eleman da biz ekliyoruz.

Bunlardan

```
mHataliSantim
```

özelliği ile istisnai duruma yol açan 99'dan büyük santim değerleri tutuluyor. Dolayısıyla fırlatılan istisnai durum nesnelerini catch bloğu yakaladığında hangi değerin hataya yol açtığı öğrenilebilecek.

Diğer bir özellik olan

```
mHataKaynagi
```

ise istisnai durumun hangi metot ya da özellik tarafından oluşturulduğunu string türünden bir değişken ile ifade etmektedir.

İstisnai durum nesnelerini fırlatırken kendi bildirdiğimiz yapıcı metotları da kullanabiliriz. Bunun için,


```
public HatalıMetrikİstisnası(int mHataliSantim,string
mHataKaynagi)
{
      this.mHataKaynagi = mHataKaynagi;
      this.mHataliSantim = mHataliSantim;
}
```

yapıcı metodu bildirilmiştir. mHataKaynagi parametresi ile istisnai durum nesnesinin fırlatıldığı metot ya da özellik belirtiliyor.

Ayrıca oluşturduğumuz bu istisnai durum sınıfında, Exception sınıfının üye metodu olan ToString(), override anahtar sözcüğü kullanılarak devre dışı bırakılmıştır. Bu metot ile oluşan istisnai durumu temsil eden nesnenin hataya yol açan değeri ve hatanın oluştuğu metodu içeren bir string değeri elde edilmektedir.

HatalıMetrikİstisnası sınıfının yapıcı metodu ile gönderilen santim değeri aslında basit bir dönüşüm ile metreye çevrilip kalan santim değeri mSantim değişkenine atanabilir. Fakat bizim burada amacımız istisnai durum yakalama mekanizmasını ne şekilde kullanabileceğimizi öğrenmek olduğu için yukarıda anlatılan yöntem seçilmiştir.

Metrik sınıfında istisnai duruma yol açacak DegerVer() ve bir de yapıcı metot olmak üzere 2 metot ve bir özellik bulunmaktadır. Bu üye elemanları program içinde görebilirsiniz. Şimdi bu açıklamalardan sonra programı inceleyin. Ardından sonuçları inceleyeceğiz.

```
using System;

class Metrik
{
      private int mMetre;
      private int mSantim;

      public Metrik(int metre,int santim)
      {
          if(santim > 99)
              throw new HatalıMetrikİstisnası(santim,"Yapıcı
                                                    Metot");

          mMetre = metre;
          mSantim = santim;
      }

      public void DegerVer(int metre,int santim)
      {
          if(santim > 99)
              throw new HatalıMetrikİstisnası(santim,
                                        "DegerVer Metodu");
```

```
        mMetre = metre;
        mSantim = santim;
    }

    public int Metre
    {
        get{ return mMetre;}
        set{ mMetre = value;}
    }

    public int Santim
    {
        get{ return mSantim;}
        set
        {
            if(value > 99)
                  throw new HatalıMetrikİstisnası
                                (value,"Santim Özelliği");

            mSantim = value;
        }
    }

    public class HatalıMetrikİstisnası : ApplicationException
    {
        private int mHataliSantim;
        private string mHataKaynagi;

        public HatalıMetrikİstisnası(int mHataliSantim,
                                     string mHataKaynagi)
        {
            this.mHataKaynagi = mHataKaynagi;
            this.mHataliSantim = mHataliSantim;
        }

        public int HataliSantim
        {
            get{ return mHataliSantim;}
        }
        public string HataKaynagi
        {
            get{ return mHataKaynagi;}
        }

        public override string ToString()
        {
            string str1 = "Hata Kaynağı :
                                    "+mHataKaynagi+",";
            string str2 = " Hata degeri : "+mHataliSantim;
            return str1 + str2;
        }
    }
}
```



```
class ExceptionTest
{
    public static void Main()
    {
        Metrik m = new Metrik(10,52);

        try
        {
            m.Santim = 650;
        }
        catch(Metrik.HatalıMetrikİstisnası e)
        {
            Console.WriteLine(e.ToString());
        }
    }
}
```

Artık bizim de .NET sınıf kütüphanesinde olduğu gibi istisnai durumlarda hata nesnesi fırlatan bir sınıfımız mevcut. Şimdi isterseniz programın sonuçlarını irdeleyelim. Main() metodundaki

```
Metrik m = new Metrik(10,52);
```

tanımlaması ile herhangi bir istisnai durum ortaya çıkmaz. Ancak

```
try
{
    m.Santim = 650;
}
```

bloğunda m nesnesinin Santim özelliğine 650 değeri atandığı için HatalıMetrikİstisnası istisnai durum sınıfı nesnesi fırlatılacaktır. Programı derleyip çalıştırdığımızda ekrana

```
Hata Kaynağı = Santim Özelliği, Hata Değeri : 650
```

yazacaktır.

Gelin bir de Main() metodunu aşağıdaki gibi değiştirelim.

```
public static void Main()
{
Metrik m = new Metrik(10,5122);
}
```

programı yukarıdaki Main() metodu ile derlediğimizde konunun başında gördüğümüz "yakalanmayan istisnai durum var" hatası alınır ve program sonlandırılır. Ancak bu sefer verilen uyarı aşağıdaki ekran görüntüsünde olduğu gibi olacaktır.

```
D:\WINNT\System32\cmd.exe
H:\cskitap\kitap\Bolum12\kod>Metrik
Unhandled Exception: Hata Kaynagi : Yapici Metot, Hata degeri : 5122
H:\cskitap\kitap\Bolum12\kod>
```

Ekran görüntüsünden de görüldüğü gibi bu seferki istisnai durum yapıcı metotta oluşmaktadır.

Konuyu bitirmeden önce birbirlerinden türeyen istisnai durum sınıfları ile ilgili önemli bir noktayı daha vurgulamakta fayda var. Birden fazla catch bloğu kullanıldığı durumlarda eğer fırlatılan istisnai durumu yakalayabilecek birden fazla catch bloğu varsa bu bloklardan ilk sıradaki sınıf hiyerarşisinde en alttaki olmalıdır. Örneğin birden fazla catch bloğu içeren aşağıdaki gibi bir yapıda hiçbir zaman ikinci ve üçüncü catch blokları çalıştırılmaz. Bu yüzden aşağıdaki gibi bir kullanım, derleme aşamasında hata verecektir.

```
catch(Exception excp)
{
        Console.WriteLine(excp.Message);
}

catch(OverflowException excp)
{
        Console.WriteLine(excp.Message);
}

catch(IndexOutOfRangeException excp)
{
        Console.WriteLine(excp.Message);
}
```

Yukarıdaki catch bloklarını içeren bir program derlendiğinde

"A previous catch clause already catches all exceptions of this or a super type ('System. Exception')" (Bir önceki catch bloğu bütün istisnai durumları yakalayabilecek düzeydedir.) hatası alırız.

Yukarıdaki hatanın oluşmaması için catch bloklarını alt alta sıralarken özelden genele doğru bir sıralama yapmamız gerekir.

C# dilinin en önemli konularından biri olan istisnai durum yakalama mekanizmasının sonuna geldik. İstisnai durumlar yerinde kullanıldığı zaman programcıya büyük faydalar sağlamaktadır. Ancak oluşabilecek en küçük hata karşısında da istinai durum sınıfları fırlatmak anlamlı olmayacaktır. Bu yüzden hatanın gerçekten istisnai bir durum olduğuna inanıyorsanız bu mekanizmayı kullanın. Ayrıca oluşturduğunuz istisnai durum sınıflarındaki hata mesajlarını da olabildiğince açık ve net belirtmeliyiz.

Exception Filter(İstisna Filtreleme) Kavramı

C# 6.0 'dan itibaren catch blokları aynı Exception tipinden nesne alabilmektedir. Peki bu durumda ilgili Exception oluştuğunda hangi catch bloğu devreye girecek? İşte bu durumda karşımıza yeni bir C# özelliği olan ***Exception Filter*** çıkıyor. Buna göre catch bloklarından hemen sonra tek satırlık bir if deyimi yazabilmekteyiz. Eğer bu if deyimi true bir sonuç üretirse ilgili catch bloğu çalışacaktır aksi durumda catch bloğuna girmeyecektir ve ilgili Exception başka bir uygun Catch bloğu yok ise programın hata vermesine neden olacaktır.

Aşağıdaki örnekte fırlatılan Exception'ın karakterestiğine (ki bu kod içerisinde programcı tarafından belirlenir) göre her iki catch bloğunda aynı ***BTAException*** sınıfı olmasına ragmen farklı catch blokları çalıştırılmaktadır. Çünkü catch bloklarının yanında if deyimi ile bir filtreleme (yakalanan istisnanın kritik olup olmamasına göre farklı catch blokları çalışmaktadır.) yapılmaktadır.

```
public class BTAException : Exception
{
    public bool IsCritic { get; set; } = true;
}

class Personel : Entity
{
    public void Update()
    {
        try
        {
            //Db ye yaz
            throw new BTAException { IsCritic = true };
        }
        catch (BTAException exc) if (exc.IsCritic == true)
        {
        }
        catch (BTAException exc) if (exc.IsCritic == false)
        {
        }
    }
}
```

Aynı şekilde yukarıdaki örnekte istisna sadece kritik bir istisna ise hata yakalanıp aksi durumda istisnanın yoluna devam etmesi sağlanabilirdi. Zaten filtrelemenin de en büyük amacı budur. Yani belli durumlarda istisnanın yoluna devam etmesini sağlamaktır. Örneğin hata kritik ise loglamak istiyoruz ama değilse hatayı yakalamak istemiyoruz ki ilgili fonksiyonu çağıran diğer kod bloğuna hatamız aynı şekilde taşınsın.

Exception Filter amacı dışında kullanılmamalıdır. Örneğin filter olarak alağıdaki gibi geriye false dönen bir fonksiyon yazıp bu fonksiyon içerisinde Loglama vs gibi şeyler yapılmamalıdır.

```
public bool Logla (){
   //bir takım kodlar
   return false;

}

...
catch (BTAException exc) if (Logla())
...
```

Önemli Not : Exception Filter kavramı C# 6.0'dan itibaren kullanılabilir.

Temsilciler (delegates) ve Olaylar (events)

Bu başlık altında diğer dillerde olmayan iki yeni veri yapısı incelenecektir. Bunlardan birincisi metotların temsil edildiği veri yapısı olan temsilciler (delegate) ikincisi ise temsilcilerle yakından ilişkili olan bir olayın meydana geldiğini bildiren olay (event) veri yapısıdır.

Temsilciler (delegates)

Yazdığımız programları çalıştırdığımızda JIT derleyicisi tarafından makine koduna çevrilip belleğe yüklenir. Makine kodunun belleğe ne şekilde yükleneceği bellek modeline göre değişebilir. Ancak değişmeyen tek şey belleğe yüklenen programımızdaki nesnelere ya da metotlara erişmek için bu elemanların bellekteki adreslerinin kullanılmasıdır. Örneğin aşağıdaki gibi bir program düşünelim.

```
using System;

class Temsilciler
{
      public static int Metot1()
      {
          return 5;

      }
      public static void Main()
      {
          Console.WriteLine("Temsilcilere")     ;
          int a = Metot1();
          Console.Write(" başlarken...");
      }
}
```

Programı çalıştırdığımızda CLR başlangıç noktası olan Main() metodunu arar. Main() metodunu bulması bellekteki belirli bir adresten itibaren makine kodunu çalıştırması

demektir. Main() metodu satır satır işlenirken Metot1() satırına gelindiğinde bu metodun bellekte bulunduğu adrese gidilir ve oradaki kodlar çalıştırılır. Değişkenler nasıl ki bellekte bir bölgede bulunuyorsa metotlar da belleğin bir bölgesindedir. Çeşitli programlama dillerinde metot adreslerine de erişmek mümkündür. C# ta ise metotların bir nevi adreslerini tutan yeni veri modeli geliştirilmiştir. İşte bu veri modelini temsil eden tip delegate'dir. delegate veri tipi C#'ın temel tiplerinden birisidir. Yani delegate isimli bir anahtar sözcük bulunmaktadır. Ancak diğer tiplerde de olduğu gibi delegate anahtar sözcüğü ile bildirilen temsilciler System.Delegate sınıfı türünden başka bir şey değildir.

Metotlar da nesneler gibi bellekte bulunur.

Temsilciler metotlara referansta bulunur. Daha alt seviyede düşünürsek temsilciler metotların bellekteki adreslerini tutar. Adres kavramı C# programcılarından tamamen gizlenmiştir. O halde temsilciler, metotları temsil eden bir veri tipidir diyebiliriz. Biraz daha açıklayalım. Yukarıdaki programda Metot1() metodunu temsil eden bir temsilci bildirebiliriz. O halde bildirdiğimiz bu temsilci ile Metot1() metodunu çağırmamızdan doğal bir şey olamaz. Zaten daha önce biz farkına varmasak da metot isimleri aslında metodun bellekte fiziksel olarak bulunduğu yerin başlangıç adresini temsil etmektedir. Bellek düzeyinde düşünürsek; bir metodun ismi aslında bellekteki bir adresi temsil ediyor. Bu adres ise program belleğe yüklendiğinde metodun bellekte bulunduğu yerin başlangıç adresidir.

Bu kısa girişten sonra temsilcileri yakından inceleyelim. Temsilciler **delegate** anahtar sözcüğü kullanılarak bildirilir. Bir temsilci, gövdesi olmayan metot bildirimi gibi yapılır. Temsilci bildiriminin en genel hali aşağıdaki gibidir.

```
delegate GeriDönüşDeğeri temsilci(parametreler);
```

Yukarıdaki ifadeye göre geri dönüş değeri int olan ve herhangi bir parametresi olmayan metodu temsil eden temsilci aşağıdaki gibi bildirilir.

```
delegate int temsilci();
```

Bildirilen bu temsilci geri dönüş değeri int olan ve parametresi olmayan bir metodu temsil edebilir.

Temsilci bildirimine bir örnek daha verelim:

```
public delegate void Komutlar(string str);
```

Komutlar isimli temsilci ile geri dönüş değeri void olan ve bir tane string parametresi olan metotları temsil edebiliriz.

"temsilci" bir Delegate türünün bildirimidir. Yani temsilci nesneleri tanımlamak için new anahtar sözcüğü kullanılır.

Temsilcilerin kullanılmasının amacı derleme zamanında belli olmayan metot çağrımlarını çalışma zamanında belirlemektir. Temsilcilerin nasıl kullanıldığını daha iyi anlamak için aşağıdaki örneği inceleyelim.

```
using System;

class Temsilciler
{
      public delegate void MetotTemsilcisi();

      public static void Metot1()
      {
          Console.WriteLine("Ben Metot1' im");
      }

      public static void Metot2()
      {
          Console.WriteLine("Ben Metot2' yim");
      }
      public static void Main()
      {
          MetotTemsilcisi temsilci = new MetotTemsilcisi(Metot1);
          temsilci();

          temsilci = new MetotTemsilcisi(Metot2);
          temsilci();
      }
}
```

Bu programı çalıştırdığımızda önce Metot1()'in sonra da Metot2()'nin çağrıldığını göreceksiniz. MetotTemsilcisi isimli temsilciyi bildirdikten sonra Main() metodu içerisinde bir temsilci nesnesi oluşturuyoruz. Bu işlem bir sınıf nesnesi oluşturmaktan farksızdır.

```
MetotTemsilcisi temsilci = new MetotTemsilcisi(Metot1);
```

ifadesinde temsilci nesnesi MetotTemsilcisi türünden bir nesnedir. Bu nesne oluşturulurken yapıcı metot parametre olarak geri dönüş değeri void ve parametresi olmayan bir metot ismi almaktadır. Dolayısıyla temsilci nesnesi artık parametre olarak gönderilen bu metodu temsil etmektedir. Zira

```
temsilci();
```

şeklindeki bir çağrım aslında Metot1() metodunu çağırmaktadır.

```
temsilci = new MetotTemsilcisi(Metot2);
```


satırında ise yeni bir MetotTemsilcisi nesnesi oluşturulup bu nesnenin referansı temsilci' ye aktarılıyor. Dolayısıyla temsilci nesnesi artık Metot2()' yi temsil etmektedir. Bunu görmek için

```
temsilci();
```

şeklinde yeni bir çağrım yapılmıştır.

temsilci nesnesini geri dönüş değeri int olan ve iki tane int parametre alan metot ile tanımlayamayız. Çünkü temsilcinin imzası ile metodun imzası aynı değildir.

Bir temsilci, temsilcinin imzasına uygun herhangi bir metodu temsil edebilir. Bu metotların statik ya da normal metot olması bir şeyi değiştirmez.

delegate bir veri tipi olduğu için public, private gibi erişim belirleyiciler ile bildirilebilirler.

Şimdi temsilcilere neden ihtiyaç duyduğumuzu daha iyi kavrayabilmek için gerçekçi bir örnek yapalım. Bu örneğe kısaca komut yorumlayıcısı da diyebiliriz. Komut yorumlayıcısı konsol ekranından girilen her komut için ayrı bir metot çalıştırılmasını sağlar. Bunu şu ana kadar gördüğümüz yollarla da yapabiliriz, ancak temsilcileri kullanarak bu işi nasıl sistematik bir şekilde yaptığımızı göreceğiz. Önce aşağıdaki programı inceleyin ve anlamaya çalışın ardından neler yaptığımıza bakacağız.

```
using System;

class Temsilciler
{
      public delegate void KomutMetodu();

      public struct KomutYapısı
      {
          public KomutMetodu KomutMetot;
          public string Komut;
      }

      public static void Dir()
      {
          Console.WriteLine("dır komutu çalıştı");
      }

      public static void Edit()
      {
          Console.WriteLine("edit komutu çalıştı");
      }
```

```
        public static void Copy()
        {
            Console.WriteLine("copy komutu çalıştı");
        }

        public static void Rename()
        {
            Console.WriteLine("rename komutu çalıştı");
        }
        public static void Main()
        {
            KomutYapısı[] komutlar = new KomutYapısı[4];

            komutlar[0].Komut = "dir";
            komutlar[0].KomutMetot = new KomutMetodu(Dir);

            komutlar[1].Komut = "edit";
            komutlar[1].KomutMetot = new KomutMetodu(Edit);

            komutlar[2].Komut = "copy";
            komutlar[2].KomutMetot = new KomutMetodu(Copy);

            komutlar[3].Komut = "rename";
            komutlar[3].KomutMetot = new KomutMetodu(Rename);

            string GirilenKomut;

            while(true)
            {
                Console.Write("DOS :> ");
                GirilenKomut = Console.ReadLine();

                for (int i=0 ; i<komutlar.Length; ++i)
                {
                        if(GirilenKomut == komutlar[i].Komut)
                                komutlar[i].KomutMetot();
                }
            }
        }
}
```

Programın çalışma anından bir görüntü aşağıdaki gibidir.

```
D:\WINNT\System32\cmd.exe - KomutYorumla

H:\cskitap\kitap\Bolum12\kod>KomutYorumla
DOS :> dir
dir komutu çalisti
DOS :> edit
edit komutu çalisti
DOS :> dirr
DOS :>
DOS :> rename
rename komutu çalisti
DOS :> _
```


Şimdi programda nelerin olup bittiğine bakalım.

```
public delegate void KomutMetodu();
```

temsilcisi ile komut metotlarını işleteceğimiz metotları temsil ediyoruz. KomutYapısı isimli veri yapısı her bir komuta ilişkin komut metnini ve komuta ilişkin metodu temsil etmektedir.

```
public struct KomutYapısı
{
        public KomutMetodu KomutMetot;
        public string Komut;
}
```

şeklinde bildirilen yapı ile bu işlem yapılmaktadır. Main() metodunda çalıştırmak istenilen komutlar bir KomutYapısı yapı dizisi içinde tutulmaktadır. Bu dizinin her bir elemanı bir komut ve komuta ilişkin metodu temsil eden bir yapı nesnesidir. Bu yüzden

```
komutlar[0].KomutMetot();
```

şeklindeki bir deyim ile komutlar dizisinin birinci elemanında bulunan temsilcinin temsil ettiği metot çağrılır. Sonsuz bir döngü içinde kullanıcının girdiği komut, komutlar dizisi içindeki her yapı elemanının string türünden olan Komut değişkeni ile kontrol ediliyor. Eğer komutlar dizisinde, kullanıcının girdiği komut yazısı bulunursa, bulunan elemanın KomutMetot temsilcisi ile komuta ilişkin metot çağrılmaktadır.

Bu örneği geliştirerek daha güzel komut yorumlayıcıları yapabilirsiniz. Örneğin ilk aşamada hatalı komut girildiğinde uyarı verilmesini sağlayabilirsiniz.

Örnekten de görüldüğü üzere delegate normal bir sınıftır. Dolayısıyla bir metodun parametresi temsilci türünden olabilir. Mesela temsilciler ile ilgili verdiğimiz örneği aşağıdaki gibi değiştirelim.

```
using System;

class Temsilciler
{
        public delegate void MetotTemsilcisi();
        public static void Metot1()
        {
            Console.WriteLine("Ben Metot1'im");
        }

        public static void Metot2()
        {
            Console.WriteLine("Ben Metot2'yim");
        }
```

```
        public static void Metot3(MetotTemsilcisi mt)
        {
            mt();
        }

        public static void Main()
        {
            MetotTemsilcisi temsilci = new
                                    MetotTemsilcisi(Metot1);
            Metot3(temsilci);

            temsilci = new MetotTemsilcisi(Metot2);
            Metot3(temsilci);
        }
}
```

Bu programı da çalıştırdığımızda önce Metot1()'in sonra da Metot2()'nin çağrıldığını göreceksiniz. İlk programdan tek farkı temsilcinin temsil ettiği metodun Metot3() isimli metot tarafından çalıştırılmasıdır.

Metot3() içinde, temsilcinin temsil ettiği metodun nasıl çağrıldığına dikkat edin.

Çoklu Temsilciler (Multicast Delegate)

delegate veri yapısının aynı imzaya sahip birden fazla metodu temsil edebildiğini öğrendiğinizde biraz şaşırabilirsiniz. Evet, bir temsilci ile birden fazla metot temsil edilebilir. Üstelik temsilci çağrımı yapıldığında temsilcinin temsil ettiği bütün metotlar sırayla çalıştırılır. Tabi istersek bu temsil edilen metotların sadece birini ya da birkaçını da çalıştırabiliriz.

Bir temsilciye birden fazla metot eklemek için += operatörü kullanılır. Aynı şekilde bir temsilciden bir metot çıkarmak için -= operatörü kullanılır. Aslında + ve – operatörlerini de kullanabiliriz. Ancak += ve -= operatörleri işimizi biraz daha kolaylaştırmaktadır.

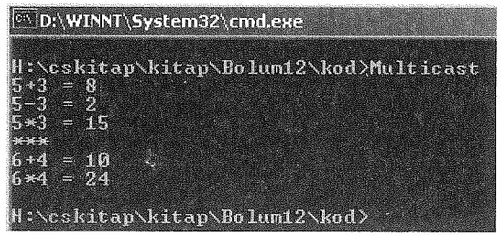
Aşağıda iki sayıyı toplayan, çarpan ve çıkaran 3 metot bildirilmiştir. Bu 3 metodun bir temsilci tarafından aynı anda nasıl çalıştırıldıklarını örnek üzerinden inceleyin.

```
using System;

class Temsilciler
{
        public delegate void MetotTemsilcisi(int a,int b);
```



```
        public static void Toplama(int a,int b)
        {
            Console.WriteLine("{0}+{1} = {2}",a , b ,a + b);
        }

        public static void Cikarma(int a,int b)
        {
            Console.WriteLine("{0}-{1} = {2}",a , b ,a - b);
        }

        public static void Carpma(int a,int b)
        {
            Console.WriteLine("{0}*{1} = {2}",a , b ,a * b);
        }

        public static void Main()
        {
            MetotTemsilcisi temsilci = null;

            temsilci += new MetotTemsilcisi(Toplama);
            temsilci += new MetotTemsilcisi(Cikarma);
            temsilci += new MetotTemsilcisi(Carpma);

            temsilci(5,3);

            Console.WriteLine("***");

            temsilci -= new MetotTemsilcisi(Cikarma);
            temsilci(6,4);
        }
}
```

Bu programı çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.



Ekran çıktısından da görüldüğü üzere += operatörü ile temsilciye eklenen metotlar

```
temsilci(5,3);
```

şeklindeki bir deyimle sırayla çağrılmıştır. -= operatörü ile temsilciden çıkarılan metot

```
temsilci(6,4);
```

deyimi ise artık çalıştırılmamıştır.

Çoklu temsilci çağırımında metotlar temsilciye eklenme sırasına göre çağrılır..

Bir temsilciye metot eklemek için aşağıdaki deyim de kullanılabilir.

```
temsilci = temsilci + new MetotTemsilcisi(Toplama);
```

Aynı kural temsilciden metot çıkarmak için de geçerlidir.

Delegate ve Multicast Delegate Sınıfları

Tek metodu temsil eden temsilciler System.Delegate sınıfı nesneleridir. Çoklu temsilciler ise System.MultiCastDelegate sınıfını temsil etmektedirler. MultiCastDelegate sınıfı Delegate sınıfından türetilmiştir. Bu yüzden birtakım ortak metotlar ve özellikler içerirler.

Delegate sınıfının önemli üye elemanları aşağıdaki gibidir.

`GetInvocationList()` : Bu metot ile çoklu temsilci yapısında bulunan bütün metotlar System.Delegate türünden bir diziye aktarılır.

`DynamicInvoke(object[] parametre_dizisi)` : Bir temsilcinin temsil ettiği metodu çağırmak için kullanılır. Özellikle GetInvocationList() metodu ile çoklu bir temsilcinin metotları Delegate dizisine aktarıldığı zaman her bir metodu ayrı ayrı çağırmak için DynamicInvoke() metodu kullanılabilir. parametre_dizisi ise metodun alacağı parametreleri içerir.

`Combine(Delegate[] temsilciler)`

`Combine(Delegate temsilci1, Delegate temsilci2)` : Statik metot olan Combine çoklu bir temsilciye temsilciler ile belirtilen Delegate dizisindeki metotları ekler. İkinci kullanımda ise temsilci1 ve temsilci2 isimli temsilcilerin bütün metotlarını arka arkaya ekler.

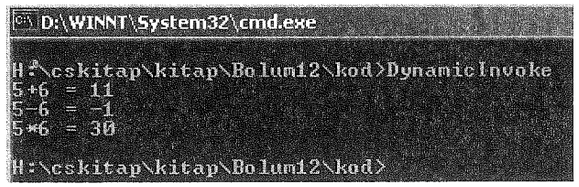
Yukarıda anlatılan `GetInvocationList()` ve `DynamicInvoke()` metotları aynı zamanda MultiCastDelegate sınıfının da üye elemanlarıdır.

`GetInvocationList()` ve `DynamicInvoke()` metotlarının kullanımına aşağıda bir örnek verilmiştir.

```
using System;

class Temsilciler
{
      public delegate void MetotTemsilcisi(int a,int b);
```



```
      public static void Toplama(int a,int b)
      {
          Console.WriteLine("{0}+{1} = {2}",a , b ,a + b);
      }

      public static void Cikarma(int a,int b)
      {
          Console.WriteLine("{0}-{1} = {2}",a , b ,a - b);
      }

      public static void Carpma(int a,int b)
      {
          Console.WriteLine("{0}*{1} = {2}",a , b ,a * b);
      }

      public static void Main()
      {
          MetotTemsilcisi temsilci = null;

          temsilci = temsilci + new MetotTemsilcisi(Toplama);
          temsilci += new MetotTemsilcisi(Cikarma);
          temsilci += new MetotTemsilcisi(Carpma);

          Delegate[] t = temsilci.GetInvocationList();
          object [] parametreler = {5,6};

          t[0].DynamicInvoke(parametreler);
          t[1].DynamicInvoke(parametreler);
          t[2].DynamicInvoke(parametreler);
      }
}
```



Görüldüğü gibi `GetInvocationList()` ve `DynamicInvoke()` metotları kullanılarak çoklu temsilci içindeki metotlar tek tek çağırılabilir. Programı derleyip çalıştırdığımızda yukarıdaki ekran görüntüsünü elde ederiz.

İsimsiz Metotlar (Anonymous Methods) ve Yan Etkileri

C# 2.0'dan itibaren temsilci nesnelerine atanacak kod bloklarını ya da olaylara ilişkilendirilecek metotları yeni bir metot tanımlamadan belirleyebiliyoruz. Örneğin bir temsilci nesnesine aşağıdaki örnekte olduğu gibi direkt bir kod bloğu bağlayabiliyoruz. Aslında arka planda tanımlanan şey metoddan başka bir şey değildir. Neden böyle bir şeye ihtiyaç duyulduğundan aşağıda bahsedeceğim.

```
using System;

namespace isimsizmetotlar
{
    public delegate double Temsilci(double a, double b);
    class uygulama
    {
        static void Main(string[] args)
        {
            //temsilci nesnesine bir kod bloğu bağlanıyor.
            Temsilci t = delegate(double pi, double r)
            {
                return pi * r * r;
            };

            //Temsilci ile kod bloğunun çağrılması
            double alan = t(3.14, 5);
            Console.WriteLine(alan);
        }
    }
}
```

Yukarıdaki örnekte de görüldüğü gibi bu tanımlama biçiminde new anahtar sözcüğünün kullanılmasına gerek kalmadı. Ayrıca tanımladığımız kod bloğuna da herhangi bir isim vermedik. **İsimsiz metot** denilmesinin sebebi de budur zaten. İsimsiz metotların bir özelliği de kod bloğu oluştururken temsilcinin tipini bilmemize gerek olmamasıdır. delegate anahtar sözcüğü otamatik olarak temsilcinin adının yerini tutmaktadır.

Önemli Not: İsimsiz metotların (kod blokları) temsilcilere atandıktan sonraki noktalı virgüle dikkat ediniz.

İsimsiz metotlar aynı zamanda olaylar ile birlikte de kullanılmaktadır. Örneğin, daha önceki örneklerde yaptığımız üzere **Buton** isimli nesnemizin **ButonKliklendi** olayına ilişkin bildirimler aşağıdaki 3 şekilde de yapılabilir.

```
using System;
```



```
namespace isimsizmetotlar
{
        //Bu bölümda Buton nesnemizin ve temsilcilerimizin
olduğunu varsayıyoruz. Daha önceki örneklerimizden kopyalayıp
yapıştırabilirsiniz.
    class Program
    {
        static void Main(string[] args)
        {

            Buton b = new Buton();

            b.ButonKliklendi += delegate
            {
                Console.WriteLine("buton tıklandı 1");
            };

            b.ButonKliklendi += delegate(object sender, EventArgs
e)
            {
                Console.WriteLine("buton tıklandı 2");
            };

            b.ButonKliklendi += buton_klik;
        }

        static void buton_klik()
        {
            Console.WriteLine("buton tıklandı 3");
        }
    }
}
```

Örnekteki üç tanımlama şeklinden en önemlisi üçüncü tanımlamadır. Dikkat ederseniz ButonKliklendi olayına direkt olarak metot ismi verilmiştir. Temsilci nesnesi arka planda otomatik olarak oluşturulmakta ve olaya eklenmektir. Programcının işini kolaylaştırması için otomatik dönüşüm yapılması önemlidir. Bu kitabın konuları içinde olmayan bir konudan örnek vermek istiyorum. Windows tabanlı programlarda sıkça karşılaşılan **çok kanallı** (*multi thread*) uygulamalar geliştirebilmek için .NET'te **Thread** denilen bir yapı vardır. Thread sınıfının **ThreadStart** isimli bir temsilci parametresi alan metodu vardır. Bu metot yardımıyla bir metot üzerinden yeni **thread** (*kanal*) başlatılır. Eskiden yeni thread aşağıdaki gibi açılırdı.

```
//ThreadStart bir temsilcidir.
ThreadStart ts =  new ThreadStart(metot);

Thread t = new Thread(ts);
t.Start();
```

Temsilcilerin metot isminden otomatik oluşturulması sayesinde yukarıdaki kod C# 2.0 aşağıdaki gibi olabilmektedir.

```
Thread t = new Thread(metot);
t.Start();
```

Yukarıdaki örnekte bahsedilen Thread kavramını detaylı bir şekilde öğrenmek için Windows Programlama kitaplarına başvurmanız gerekir.

İsimsiz metotlar gerçek projelerde sıkça kullanılabilecek (kullanılsa bile bu haliyle çok gerekli bir özellik değildir) bir durum değildir. (metot isimlerin otomatik olarak temsilcilere dönüşmesi hariç) İsimsiz metotlar C# 3.0 ile birlikte gelen lambda ifadeleri ve devamındaki özellikler için bir temel oluşturması amacıyla eklenmiştir.

covariance ve contravariance Durumları

C# 2.0'dan önce temsilcilerin temsil edeceği metotların prototipi temsilcinin prototipi ile aynı olmak zorunda idi. Diğer bir deyişle temsil edilen metotların geri dönüşlerinde ya da parametrelerinde kalıtımsal bir ilişki göz önünde bulundurulmuyordu. Örneğin **Dortgen** sınıfının **Sekil** isimli sınıftan türediğini düşünelim. Bu durumda aşağıdaki her iki temsilci nesnesi oluşturma biçimi de C# 2.0'dan önce hata verecektir.

```
using System;
using System.Collections.Generic;
using System.Text;

public delegate Sekil Temsilci1();
public delegate Dortgen Temsilci2();

public class Sekil
{
    public Sekil()
    {
    }
}

public class Dortgen : Sekil
{
    public Dortgen()
    {
    }
}

class Program
{
    static void Main(string[] args)
    {
```



```
        //aşağıdaki iki tanımla şeklinde C# 1.0 ve C# 1.1 de
        //geçersiz ilen C# 2.0 geçerli hale gelmiştir.

        //covariance durumu
        Temsilci1 t1 = new Temsilci1(MetotDortgen);

        //contravariance durumu
        Temsilci1 t2 = new Temsilci1(MetotSekil);
    }

    public static Sekil MetotSekil()
    {
        return null;
    }

    public static Dortgen MetotDortgen()
    {
        return null;
    }
}
```

Yukarıdaki **Main** metodunda bulunan tanımlamalar C# 1.1 de geçersiz iken aynı tanımlamalar C# 2.0'dan itibaren geçerli hale gelmiştir. Yani temsilciler de artık çok biçimliliği (polymorphism) desteklemektedir.

Olaylar (Events)

Olaylar, temsilcilerin özel bir halidir. Aslına bakarsanız olaylar olmadan da C#'taki olay mantığını temsilcilerle kurabilirdik. Ancak işlerin daha sade olması için C# dilini tasarlayanlar olay bildirimi için ayrıca **event** anahtar sözcüğünü kullanabilmemizi sağlamışlardır. Peki, olay ne demektir? Grafik arayüzü içeren programlarda olaylar sıklıkla kullanılmıştır. Örneğin bir Windows uygulamasında bir butona tıkladığımızda kendi yazdığımız bir bölüm kodun çalışması olay mantığı ile gerçekleşmektedir. Buradaki olay butonun tıklanmasıdır. Butona tıklanınca butonun tıklanması ile ilişkin yazdığımız metot çalıştırılır. Bu metodu çağıran ise olay yöneticisi dediğimiz temsilcilerdir. Her ne kadar grafik arayüzü olan programlarda olaylar sıklıkla kullanılsa da bu zorunlu değildir. Kendi oluşturduğumuz basit sınıflar ile de olayların çalışma mantığını inceleyebiliriz.

Olaylar temsilcilerin özel bir halidir.

Olaylar sınıfların bir üye elemanıdır. Bir olay en genel şekliyle aşağıdaki gibi bildirilir.

```
[erişim belirleyici] event [temsilci türü] [olay ismi];
```

Şimdi temsilcilerin ve olayların bu sıkı ilişkisini görmek için aşağıdaki programı ince-

leyelim. Bu programda bir olayın meydana getirilmesi ve olay sonucunun değerlendirilmesi ile ilgili bütün yapılar mevcuttur.

```
using System;
//Olay yöneticisi bildirimi
delegate void OlayYoneticisi();

//Olay sınıfı bildirimi
class Buton
{
      //Olayın bildirimi
      public event OlayYoneticisi ButonKlik;

      //Olayı meydana getirecek metot
      public void Kliklendi()
      {
          if (ButonKlik != null)
              ButonKlik();
      }
}
public class OlayTest
{
      static void Main()
      {
          Buton buton = new Buton();

          //Olay sonrası işletilecek metotların
          //eklenmesi.
          buton.ButonKlik += new OlayYoneticisi(ButonKlik);

          //Olayın meydana getirilmesi
          buton.Kliklendi();
      }

      //Olay sonrası işletilecek metot(Event Handler)
      public static void ButonKlik()
      {
          Console.WriteLine("Buton kliklendi!");
      }
}
```

Programı derleyip çalıştırdığımızda ekrana

```
Buton kliklendi!
```

yazdığını göreceksiniz. Şimdi bu programı yakından inceleyelim. Programdaki en kritik yapı Buton sınıfıdır. Buton sınıfında


```
public event OlayYoneticisi ButonKlik;
```

şeklinde bir olay bildirimi yapılıyor. Bu bildirimde OlayYoneticisi temsilcisi kullanılmıştır.

```
delegate void OlayYoneticisi();
```

şeklindeki temsilci bildiriminden anlamamız gereken ButonKlik isimli olay meydana geldiğinde geri dönüş değeri void ve parametresi olmayan metotların çalıştırılacağıdır.

Buton sınıfında Kliklendi() metodu butonun tıklanmasını sağlar. Yani olayı meydana getirmek için buton nesneleri üzerinden bu metodu çağırmamız gerekir. Diğer bir deyişle

```
Buton buton = new Buton();
buton.Kliklendi();
```

deyimleri ile olay başlatılmış olur. Olayı başlatan Kliklendi() metodunun gövdesine dikkat edin.

```
if (ButonKlik != null)
      ButonKlik();
```

deyimleri ile yapılmak istenen tam olarak şudur: ButonKlik olayı nesnesinde olay olduğunda eğer olayı değerlendirecek bir metot bulunmuyorsa ButonKlik olayı ile temsil edilen metotlar çağrılmasın. Eğer bu kontrolu yapmamış olsaydık ButonKlik olayı tarafından herhangi bir metot çağrılamayacağı için NullReferenceException istisnai durumu meydana gelirdi.

Şimdi Main() metodunun içine bakalım. Main() olayın meydana getirildiği metottur.

```
buton.ButonKlik += new OlayYoneticisi(ButonKlik);
```

ifadesi dikkatinizi çekmiş olabilir. Olay nesnelerinde de temsilcilerde olduğu gibi += ve -= operatörlerini kullanabiliriz. Bu ifade ile Buton sınıfındaki olay elemanına yeni bir metot eklenmektedir. Dolayısıyla ButonKlik olayı meydana geldiğinde bu eklediğimiz metotlar çalıştırılacaktır. Bu deyimle eklenen ButonKlik() statik metot olduğu için herhangi bir nesne üzerinden çağırmamıza gerek yoktur. Son olarak buton nesnesi üzerinden

```
buton.Kliklendi();
```

deyimi ile butonun kliklenme olayı gerçekleştiriliyor ve statik ButonKlik() metodu olayı değerlendirmek için çalıştırılıyor. Gördüğünüz gibi yapılan bir buton tıklandığından bir metodu otomatik olarak işletmekten başka bir şey değildir.

Bu örnekte olayın başlatılması ve olay oluştuktan sonra çalıştırılacak metot (event handler) bizim tarafımızdan yapılmıştır. Özellikle Windows programlamada olayın oluşturulması ile biz çok fazla ilgilenmeyiz. Olaylar, işletim sistemi tarafından bizim programa gönderilir. Bizim yapmamız gereken belirlenen bu olaylar gerçekleştiğinde çalışacak kodları belirlemektir. Kitabın son bölümünde Windows programlama ile

ilgili temel bilgiler anlatılırken .NET'in pencere elemanlarına (Windows forms) yönelik olayları ne şekilde işlediğini inceleyeceğiz.

Bu programda buton kliklenmesi olayında sadece bir metot çağrılmıştır. Ancak event nesnesine += operatörü ile yeni metotlar ekleyerek olaydan sonra birden fazla metodun çalıştırılmasını sağlayabiliriz. Bunu görmek için OlayTest sınıfını aşağıdaki gibi değiştirip tekrar derleyin.

```
public class OlayTest
{
      static void Main()
      {
          Buton buton = new Buton();

          buton.ButonKlik += new OlayYoneticisi(ButonKlik1);
          buton.ButonKlik += new OlayYoneticisi(ButonKlik2);
          buton.Kliklendi();
      }

      public static void ButonKlik1()
      {
          Console.WriteLine("Buton kliklendi!(1)");
      }

      public static void ButonKlik2()
      {
          Console.WriteLine("Buton kliklendi!(2)");
      }
}
```

Bu sefer buton nesnesi üzerinden Kliklendi() metodu çağrıldığında OlayTest sınıfı içindeki hem ButonKlik1() hem de ButonKlik2() metodunun çağrıldığını görürüz. Yani ekrana

```
Buton kliklendi!(1)
Buton kliklendi!(2)
```

yazılacaktır.

Bu metotların olay temsilcisine eklenme sırasına göre çağrıldığına dikkat edin.

Bu programdaki ButonKlik1() ve ButonKlik2() metotlarına **event handler** denilmektedir.

Bu programda olay yöneticisi olan OlayYoneticisi isimli temsilcinin geri dönüş değeri void ve herhangi bir parametre almayan metotlar ile çalışabileceğine dikkat etmişsinizdir.

Teoride her türden temsilci olayları yönetmek için kullanılabilir ancak olay sonrasında çalıştırılacak metotların geri dönüş değerlerine genellikle gerek duyulmaz. Bu yüzden temsilciler geri dönüş değeri void olan metotları temsil edecek şekilde bildirilir. Ancak bildirilen bu temsilciler her türlü parametreyi alabilir. Bu örnekte temsilci üzerinden çağrılacak metotlar herhangi bir parametre almamıştır. Ancak bu şart değildir. Olayları işletecek metotların istenildiği kadar parametresi olabilir. Bu, çoğu durumda gerekli bile olabilir. Bu konuya ilerleyen kısımlarda daha ayrıntılı bir şekilde değineceğiz.

Olayları işletecek metotların statik metot olma zorunluluğu yoktur. Olay nesnesine herhangi bir nesne üzerinden diğer sınıfların normal metotları da eklenebilir. Aşağıdaki örnekte Pencere isimli sınıftaki ButonKlik metodunun olay nesnesine nasıl eklendiğine dikkat edin.

```
using System;

delegate void OlayYoneticisi();

class Buton
{
      public event OlayYoneticisi ButonKlik;

      public void Kliklendi()
      {
          if (ButonKlik != null)
              ButonKlik();
      }
}

class Pencere
{
      int PencereNo;

      public Pencere(int no)
      {
          PencereNo = no;
      }
      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);

        buton.ButonKlik += new OlayYoneticisi(ButonKlik);
        buton.ButonKlik += new OlayYoneticisi(p1.ButonKlik);
        buton.ButonKlik += new OlayYoneticisi(p2.ButonKlik);

        buton.Kliklendi();
    }

    public static void ButonKlik()
    {
        Console.WriteLine("Statik metot olayı algıladı.");
    }
}
```

Program derlenip çalıştırıldığında aşağıdaki ekran görüntüsü elde edilir.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum12\kod>Olaylar
Statik metot olayi algiladi.
1 nolu pencere olayi algiladi.
2 nolu pencere olayi algiladi.

H:\cskitap\kitap\Bolum12\kod>
```

Bu ekran çıktısı butonun klikleme olayı gerçekleştiğinde p1 ve p2 nesneleri üzerinden ButonKlik metotlarının yanısıra statik ButonKlik metodunun çalıştırıldığını gösterir.

buton.ButonKlik -= new OlayYoneticisi(p2.ButonKlik);

Olaylar bir tür olduğu için arayüzlerde de bildirilebilir. Ayrıca arayüzler sanal (virtual) ve özet (abstract) olarak da bildirilebilir.

add ve remove Erişimcileri

Olay yöneticisine yeni metotlar eklediğimizde ya da metot çıkardığımızda belirli kod bloğunu çalıştırmak için olay bildirimine add ve remove bloklarını eklememiz gerekir. add ve remove bloklarının kullanımı sınıfların özelliklerinde kullandığımız set ve get bloklarına benzer. Bundan önceki örneklerde bir olay yöneticisine += operatörü ile metot eklerken ya da -= operatörü ile metot çıkarırken herhangi bir kısıtlamamız yoktu. Ancak bazen olay yöneticisine eklenecek metotlar ile ilgili kısıtlamalarımız

olabilir. Örneğin bir olay için sadece belirli sayıda metodun çalıştırılmasını isteyebiliriz. Bu durumda olay yöneticisine eklenen metotların kontrolünü yapmamız gerekir. Bunun için olayların bildirilmesi sırasında add ve remove blokları kullanılır. Olay yöneticisine += operatörü ile metot eklendiğinde add bloğu -= operatörü ile metot çıkarıldığı zaman remove bloğu çalıştırılır.

Aşağıdaki örnekte Buton sınıfı geliştirilerek ButonKlik olayı gerçekleştiğinde olay yöneticisi tarafından en fazla iki adet metot çalıştırılacak şekilde düzenlenmiştir. Bunun için `OlayYoneticisi` temsilcisi türünden iki elemanlı bir dizi tanımlanmıştır. add ve remove bloklarında olay dizisi içindeki temsilciler kontrol edilmektedir. add bloğunda null değerde olmayan bir temsilci yoksa olay temsilcisi dolu anlamına geldiği için bir uyarı veriliyor. Aynı şekilde remove bloğunda eğer dizinin her iki elemanı da null değerde ise -= operatörü ile olay yöneticisinden çıkarılacak metot bulunamadı anlamına gelecektir.

add bloğunda kullanılan value anahtar sözcüğü += operatörü ile eklenmek istenen metodu temsil etmektedir.

Programın kaynak kodu aşağıdaki gibidir.

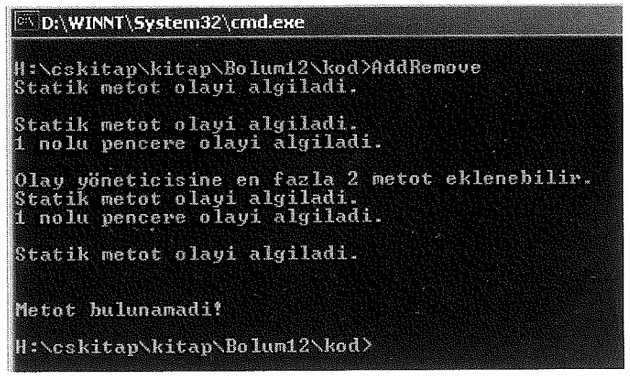
```
using System;

delegate void OlayYoneticisi();

class Buton
{
    OlayYoneticisi[] olay = new OlayYoneticisi[2];

    public event OlayYoneticisi ButonKlik
    {
        add
        {
            int i;

            for(i=0; i<2; ++i)
                if(olay[i] == null)
                {
                    olay[i] = value;
                    break;
                }

            if(i==2)
                Console.WriteLine("Olay yöneticisine en
                                   fazla 2 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;
    }

    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 yöneticisi 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();

        //Geçersiz ekleme(Metot bulunamadı)
        buton.ButonKlik -= new OlayYoneticisi(ButonKlik);
        buton.Kliklendi();
    }
    public static void ButonKlik()
    {
        Console.WriteLine("Statik metot olayı algıladı.");
    }
}
```

Programı çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.



Programın ekran çıktısından da görüldüğü gibi ButonKlik olayına ilişkin en fazla iki metot çağrılmaktadır. Olayların bu şekilde kullanılmasıyla oluşturduğumuz sınıf kütüphanelerindeki olayları rahatlıkla kontrol altına alabiliriz.

add ve remove bloklarını içeren olaylar özet(abstract) olarak bildirilemez.

add ve remove blokları ayrı ayrı bulunamaz. Her iki blokta mutlaka aynı anda kullanılmalıdır.

Olaylar ile Nesneler Arası Mesajlaşma

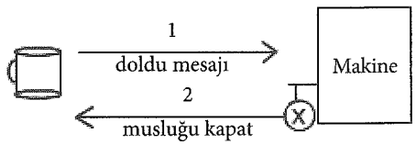
Event (*olay*) mekanizmasının kullanım amaçlarından belki de en önemlisi nesneler arasında minimum bağlantı ile aralarında mesajlaşma sağlamaktır. Bu bölümde mesajlaşma için basit bir örnek vereceğiz. Örneğimizde **Çay Makinesi** ve **Fincan** nesneleri bulunmaktadır. Çay Makinası belli bir debi ile kendisine verilen bir fincanı çay ile doldurmaktadır. Fincanın elbette ki bir kapasitesi vardır. Fincan dolduğunda Çay Makinasının bundan haberi olmalıdır ki musluğunu kapatsın. İşte fincan ve makina arasındaki bu haberleşme **BardakDoldu** isimli bir olayla gerçekleştirilmektedir. Bir önceki **event** örneğinde bütün elemanlar detaylı bir şekilde açıklandığından burada örnek kod tekrar açıklanmayacaktır.

```
using System;
namespace delegateeventornek
{
    public delegate void BardakDolunca();

    class Program
    {
        static void Main(string[] args)
        {
            Bardak b = new Bardak();
            Makina m = new Makina(b);
            m.Doldur(3, b);
        }
    }

    class Bardak
    {
        int Kapasite = 20;
        int miktar = 0;

        public event BardakDolunca BardakDoldu;

        public void BardagiDoldur(int debi)
        {
            for (int i = 0; i < Kapasite; i++)
            {
```



```
                if (miktar + debi < Kapasite && debi > 0)
                {
                    miktar += debi;
                    Console.WriteLine(miktar.ToString());
                }
                else
                {
                    Console.WriteLine("bardak doldu");
                    BardakDoldu();
                    return;
                }
            }
        }
    }

    class Makina
    {
        int debi;
        public void Doldur(int Debi, Bardak b)
        {
          debi = Debi;
          b.BardagiDoldur(debi);
        }

        void MusluguKapa()
        {
          debi = 0;
          Console.WriteLine("musluk kapandı");
        }

        public Makina(Bardak b)
        {
          b.BardakDoldu += new BardakDolunca(MusluguKapa);
        }
    }
}
```

Yandaki şekilde nesneler arasındaki iletişim basit bir şekilde gösterilmektedir.



Yukarıdaki örnek elbette ki birçok şekilde simule edilebilirdi. Ancak event mekanizmasını kullanmamızın sebebi nesneler arasındaki bağlılığı minimuma indirgemektir ki bu nesne yönelimli tasarımdaki en önemli kavramlardan biridir.

.NET ve Olaylar

.NET sınıf kütüphanesinde özellikle Windows programlamaya ilişkin isim alanlarında birçok olay tanımlanmıştır. Bu olayları işleyen metotların (event handler) prototipi genellikle aşağıdaki gibidir.

```
void EventHandler(object kaynak, EventArgs e)
```

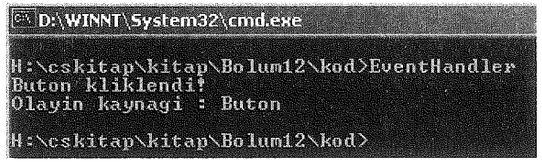
kaynak parametresi olayı elde eden nesneyi temsil etmektedir. e parametresi ise System isim alanında bulunan EventArgs sınıfı türündendir; olayı meydana getirildiği yerden, olayı işleyen metoda ekstra bilgi göndermek için kullanılır. EventArgs sınıfında işimize yarayabilecek çok fazla eleman yoktur. Bu yüzden EventArgs sınıfından yeni bir sınıf türetip kendi olay sınıflarımız için yeni özellikler eklemeliyiz. Böylece diğer .NET olayları ile uyumlu bir olay tasarlamış oluruz. Ayrıca .NET uyumlu olaylar için önceden tanımlanmış EventHandler temsilcisi bulunmaktadır. Bu temsilci ile yukarıda bahsedilen türdeki metotlar ile çalışabiliriz. Dolayısıyla .NET uyumlu olaylar geliştiriyorsak ekstra bir temsilci bildirmemize gerek yoktur.

Aşağıda .NET uyumlu olay metotlarına ve EventHandler temsilcisinin kullanımına bir örnek verilmiştir.

```
using System;
class Buton
{
      public event EventHandler ButonKlik;

      public void Kliklendi()
      {
          if (ButonKlik != null)
              ButonKlik(this, new EventArgs());
      }
}
public class OlayTest
{
      static void Main()
      {
          Buton buton = new Buton();

          buton.ButonKlik += new EventHandler(ButonKlik);

          buton.Kliklendi();
      }

      public static void ButonKlik(object kaynak, EventArgs e)
      {
          Console.WriteLine("Buton kliklendi!");
          Console.WriteLine("Olayın kaynağı : {0}",kaynak);
      }
}
```


Gördüğünüz gibi EventHandler temsilcisi sayesinde olayları yönetmek için kendimiz yeni herhangi bir temsilci bildirmedik.

Programı derleyip çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.



Eğer olay mantığını tam olarak anlamadığınızı düşünüyorsanız bu konuyu tekrar okumanızı tavsiye ederim, zira bu konunun anlaşılması diğer konulara nazaran biraz daha zordur.

Önişlemci Komutları

Bu kısımda kaynak kodumuzun derleme aşamasında ne şekilde ele alınacağını bildiren ön işlemci komutlarını ve bu komutların nerelerde kullanılabileceklerini inceleyeceğiz.

Önişlemci

Önişlemci, çalıştırılabilir kod oluşturulmadan önce kaynak kod üzerindeki önişlemci komutlarını yorumlayan bir birimdir. C ve C++ programcılarının sıklıkla kullandığı önişlemci komutları C# dilinde pek kullanılmaz. Zaten .NET'te önişlemci olarak ayrı bir birim bulunmaz. Önişlemci denilen birim derleyicinin bir parçası olarak görülür. Buna rağmen birazdan inceleyeceğimiz önişlemci komutları C ve C++ dillerindeki komutlara çok benzediği için geleneği bozmamak için bunlara önişlemci komutları denilmektedir.

C ve C++ dillerinde de olduğu gibi ön işlemci komutları çalıştırılabilir kodda görülmez.

Ön işlemci komutları C ve C++ dillerinde olduğu gibi # karakteri ile kullanılırlar. C# ile kullanabileceğimiz önişlemci komutlarına toplu olarak bir göz atalım.

Bu komutlar aşağıda toplu olarak verilmiştir.

```
#define #undef
#if #endif
#else #elif
```

```
#warning #error
#region #endregion
#line
```

Şimdi bütün bu önişlemci komutlarını detaylı bir şekilde inceleyelim.

#define ve #undef

Kaynak kodda bir sembol tanımlamak için #define komutu kullanılır. Tanımlanan sembolün ne işe yaradığını merak ediyor olabilirsiniz. Aslına bakarsanız #define ile tanımlanan bir sembolün tek başına hiçbir anlamı yoktur. Ancak diğer önişlemci komutlarını özellikle de #if önişlemci komutunu inceledikten sonra #define komutunun ne şekilde kullanılabileceğini daha iyi göreceğiz. Şimdilik #define önişlemci komutunun genellikle koşullu derleme yapmak için kullanıldığını bilmenizde fayda var. Koşullu derleme kaynak kodun sadece bir bölümünün derleme modülüne iletmek anlamına gelmektedir.

Önişlemci komutları kaynak kodda herhangi bir nesne tanımlayabileceğimiz türlerin bildirimi yapılmadan önce kullanılmalıdır. Dolayısıyla #define önişlemci komutunu genellikle kaynak kodun en başına yazarız. Aşağıda #define önişlemci komutunun kullanımına bir örnek verilmiştir.

```
#define ENGLISH
#define TURKCE

using System;

public class Onislemci
{
      static void Main()
      {

      }
}
```

Yukarıda #define ile tanımlanan ENGLISH ve TURKCE sembolleri kaynak kodun sonuna kadar tanınacaktır.

Önişlemci komutlarının kullanıldığı satırlar ';' ile sonlanmaz. Önişlemci için komut ayırıcı karakter satır sonu karakteridir.

Yukarıdaki kaynak koddan da görüldüğü üzere TURKCE sembolü tanımlanırken #define komutundan sonra boşluk karakterleri bırakılmıştır. Buradaki boşluklar tanımlanan sembole dahil edilmez.

#undef önişlemci komutu #define önişlemci komutu ile tamamen zıt bir işlem yapar. #undef ile tanımlanan sembolü kaldırır. Kullanımı #define önişlemci komutunun kullanımı ile aynıdır. Örneğin ENGLISH sembolünün tanımını kaldırmak için

```
#undef ENGLISH
```

yazılır. #undef ile belirtilen sembol eğer tanımlı değilse herhangi bir hata oluşmaz. Aynı şekilde tanımlı olan bir sembolü #define ile tekrar tanımlamak da herhangi bir hataya yol açmaz.

#define ile tanımlanan sembollerin büyük harflerle yazılma zorunluluğu yoktur. Ama geleneksel olarak bu semboller büyük harflerle yazılır.

#if ve #endif

#if ve #endif önişlemci komutları birlikte kullanılır. Daha doğrusu bu iki komut bir bütündür. Bu komutlar genellikle koşullu derleme yapmak için kullanılır. #if önişlemci komutu ile bir sembolün tanımlanıp tanımlanmadığı kontrol edilir. #if önişlemci komutu ile birlikte kullanılan sembol eğer #define ile tanımlanmışsa true değerine sahiptir, tanımlanmamışsa false değerine sahiptir.

Bu iki önişlemci komutunun genel kullanım biçimi aşağıdaki gibidir.

```
#if TURKCE
 //Kodlar
#endif
```

Önişlemci, #if TURKCE satırını gördüğünde TURKCE sembolünün tanımlanıp tanımlanmadığını kontrol eder. Eğer tanımlanmışsa #if ile #endif önişlemci komutları arasındaki kodlar derleme modülüne iletilir. Eğer TURKCE sembolü tanımlanmamışsa bu kodlar derleme modülüne iletilmez. Yani oluşturulan çalıştırılabilir dosyaya bu kodlar eklenmez.

Koşullu derleme özellikle bir programın birden fazla versiyonu aynı kaynak dosya içinde bulunuyorsa duruma göre farklı versiyonlarını derleme modülüne iletmek için kullanılır. C gibi yapısal programlama dillerinde bulunan başlık dosyalarını kaynak koda eklerken isim çakışmasını önlemek için de bu önişlemci komutları kullanılır. Ancak bu komutlar C#'taki isim organizasyonu isim alanlarıyla çok iyi bir şekilde organize edildiği için bu amaçla kullanılmaz.

#if ve #endif önişlemci komutlarının sonuçlarını daha yakından görmek için aşağıdaki programı inceleyelim.

```
using System;

public class Onislemci
```

```
{
    static void Main()
    {
        #if ENGLISH
            Console.WriteLine("English Version");
        #endif

        Console.WriteLine("Ortak Kodlar");
    }
}
```

Bu program derlendiğinde ENGLISH sembolü tanımlı olmadığı için ekrana sadece

```
Ortak Kodlar
```

yazacaktır. Kaynak kodun en başına

```
#define ENGLISH
```

komutunu eklediğimizde bu sefer ekrana

```
English Version
Ortak Kodlar
```

yazacaktır.

#if önişlemci komutu ile mantıksal operatörleri de kullanabiliriz. Örneğin hem ENGLISH hem de TURKISH sembolünün tanımlanıp tanımlanmadığı değişik şekillerde test edilebilir. Aşağıda bazı test örnekleri verilmiştir.

1. #if ENGLISH && TURKCE

Bu kullanımda hem ENGLISH hem de TURKCE sembolleri tanımlanmışsa #if ile #endif arasındaki kodlar derleme modulüne iletilir.

2. #if ENGLISH || TURKCE

Bu kullanımda ENGLISH ya da TURKCE sembollerinden herhangi biri tanımlanmışsa #if ile #endif arasındaki kodlar derleme modulüne iletilir.

3. #if ENGLISH && (!TURKCE)

Bu kullanımda ENGLISH sembolü tanımlanmış ve TURKCE sembolü tanımlanmamışsa #if ile #endif arasındaki kodlar derleme modulüne iletilir.

4. #if ENGLISH || (TURKCE==false)

Bu kullanımda ENGLISH sembolü tanımlanmış ya da TURKCE sembolü tanımlanmamışsa #if ile #endif arasındaki kodlar derleme modulüne iletilir.

Bu örnekleri değişik şekillerde çoğaltmamız mümkündür.

#else ve #elif

C# dilindeki if-else yapısındaki gibi karar verme mekanizmasını genişletmek için #else ya da #elif önişlemci komutları kullanılır. #else önişlemci komutunun genel kullanım biçimi aşağıdaki gibidir.

```
#if SEMBOLLER
      //1.Bölge
#else
      //2.Bölge
#endif
```

Eğer SEMBOL tanımlanmışsa 1. bölgedeki kodlar derleme modülüne iletilecektir. Eğer SEMBOL tanımlanmamışsa 2. bölgedeki kodlar derleme modülüne iletilecektir.

#else önişlemci komutunun derleme sonucunu daha iyi görmek için aynı isimli fakat farklı dil versiyonlarında olan iki sınıf bildiriminin nasıl ayrı ayrı derleme modülüne iletildiğini aşağıda verilen örnekte inceleyin.

```
#define TURKCE

using System;

#if TURKCE
      class Deneme
      {
          public Deneme()
          {
              Console.Write("Bir nesne oluşturuldu.");
          }
      }
#else
      class Deneme
      {
          public Deneme()
          {
              Console.Write("One object is constructed");
          }
      }
#endif

public class Onislemci
{
```

```
        static void Main()
        {
            Deneme d = new Deneme();
        }
}
```

Bu programı derleyip çalıştırdığınızda ekrana

```
Bir nesne oluşturuldu.
```

yazıldığını görürsünüz. Normal şartlarda aynı isimli sınıfların bildirimini bir isim alanında yapmayız. Halbuki bu programda Deneme isimli sınıfın iki farklı versiyonunun bildirimi yapılmıştır. Bunun sebebi bu iki versiyondan sadece bir tanesinin derleme modülüne iletilecek olmasıdır.

Diğer bir önişlemci komutu olan #elif ise "else if" deyiminin kısaltılmış biçimidir. İkiden fazla derleme koşulu mevcut ise #else önişlemci komutu yerine #elif komutunu kullanabiliriz.

#if, #endif ve #elif önişlemci komutları birlikte kullanılarak aşağıdaki gibi merdiven şeklinde bir derleme koşulu oluşturulabilir.

```
#if SEMBOLLER
      //1.Bölge
#elif SEMBOLLER
      //2.Bölge
#elif SEMBOLLER
      //3.Bölge
.
.
.

#else
      //n.Bölge
#endif
```

İç içe geçmiş #if ve #elif bloklarını da aşağıdaki şekilde oluşturabiliriz.

```
#if SEMBOL1
      //1.Bölge
      #if
          //2.Bölge
      #endif
#elif SEMBOL3
      //3.Bölge
#else
      //4.Bölge
#end if
```


#error ve #warning

#error önişlemci komutu derleyiciyi hata vermeye zorlamak için kullanılır. #error önişlemci komutu aşağıdaki gibi kullanılır.

```
#error hata mesajı
```

Aşağıdaki programda hem ENGLISH hem de TURKCE aynı anda tanımlanmışsa derleyici #error önişlemci komutu ile belirtilen hatayı verecektir. Dolayısıyla kaynak kod derlenmeyecektir.

```
#define TURKCE
#define ENGLISH

using System;

#if TURKCE && ENGLISH
      #error TURKCE ve ENGLISH aynı anda tanımlanamaz!
#endif

class Onislemci
{
      static void Main()
      {

      }
}
```

Programı derlediğimizde aşağıdaki gibi bir ekran görüntüsü ile karşılaşırız.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum12\kod>csc Error.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserve
d.

Error.cs(7,9): error CS1029: #error: 'TURKCE ve ENGLISH aynı
        anda tanımlanamaz!'

H:\cskitap\kitap\Bolum12\kod>
```

#warning önişlemci komut #error önişlemci komut ile hemen hemen aynıdır. Tek fark #warning ile hata yerine uyarı verilmesidir. Uyarı verilince doğal olarak program derlenecektir. Fakat #error komutunda program derlenmez. Bir önceki programdaki

```
#error TURKCE ve ENGLISH aynı anda tanımlanamaz!
```

satırını

```
#warning TURKCE ve ENGLISH sembollerinin aynı anda tanımlı olması
tehlikelidir
```

şeklinde değiştirirsek aşağıdaki ekran görüntüsü elde edilir.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum12\kod>csc Warning.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserve
d.

Warning.cs(7,11): warning CS1030: #warning: 'TURKCE ve
        ENGLISH sembollerinin ayni anda tanimli olmasi
        tehlikelidir'

H:\cskitap\kitap\Bolum12\kod>
```

#line

#line önişlemci komut hata ayıklama işlemlerinde sıklıkla kullanılır. #line önişlemci komutu kullanıldığı satırdan bir sonraki satırı #line komutu ile belirlenen sayıya atar. Aynı zamanda dosyanın ismini de #line ile belirtilen isim ile değiştirir. #line komutu aşağıdaki biçimde kullanılır.

```
#line SatırNo "YeniDosya.cs"
```

#line önişlemci komutunun kullanılmasına bir örnek verelim. Aşağıdaki programda Main() metodu içinde

```
int a = false;
```

şeklinde bilinçli bir şekilde hata yapılmıştır. Bu hata satırından önce

```
#line 1 "YeniDosya.cs"
```

komutu ile satır numarası ve dosya ismi değiştiriliyor.

```
using System;

class Onislemci
{
      static void Main()
      {
          #line 1 "YeniDosya.cs"
          int a = false;
      }
}
```

Programı derlediğimizde derleyici, aşağıdaki ekran görüntüsünde olduğu gibi hatanın "YeniDosya.cs" dosyasında ve 1 numaralı satırda olduğunu belirtmektedir.


```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum12\kod>csc Line.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserve
d.

YeniDosya.cs(1,11): error CS0029: Cannot implicitly convert
        type 'bool' to 'int'

H:\cskitap\kitap\Bolum12\kod>
```

#region ve #endregion

Bu komutların faydaları Visual Studio.NET gibi görsel editörlerde görülmektedir. #region ve #endregion komutlarının etkisi Visual Studio.NET aracında aşağıdaki gibidir.

```
Kaynak kodumuz

using System;

class Onislemci
{
      static void Main()
      {
          #region YeniAlan

          Console.WriteLine("Sayı gir:");
          int a = Convert.ToInt32(Console.ReadLine());
          Console.WriteLine(a);

          #endregion

      }
}
```

şeklinde ise Visual Studio.NET'teki editör ile #region ve #endregion komutları arasındaki kodlar aşağıdaki şekilde olduğu gibi kümelenebilir. (Farenin imleci #region ile belirlenen isim üzerine geldiğinde ise kümelenen kod parçaları gösterilir. Farenin imleci uzaklaştığında kodlar da görünmez hale gelir.)

```
using System;

class Onislemci
{
    static void Main()
    {
        YeniAlan
                #region YeniAlan

                Console.WriteLine("Sayı gir:");
                int a = Convert.ToInt32(Console.ReadLine());
                Console.WriteLine(a);

                #endregion
    }
}
```

Kitabın son bölümünde Visual Studio.NET ile ilgili detaylı bilgi verilecektir.

C#'ta Gösterici (Pointer) Kullanımı ve Emniyetsiz (unsafe) Kod

Bu kısımda C ve C++ dillerinde sıklıkla kullanılan üstelik bu dilleri güçlü kılan bir özellik olan göstericileri C# dilinde nasıl kullanabileceğimizi inceleyeceğiz. Göstericilerin diğer ismiyle pointer'ların programlamadaki önemi büyüktür. Bunun sebebi bellek yönetiminin direkt programcıya bırakılmasıdır. Her ne kadar .NET platformunda bellek yönetimi gereksiz bilgi toplayıcısı (**garbage collector**) dediğimiz mekanizma ile otomatik olarak sağlanıyorsa da bazen belleğe direkt erişmemiz gerekebilir. Bu durumda gösterici kullanmamız gerekir. Peki, gösterici ne demektir? Göstericiler nesnelerin bellekte bulunan adreslerini tutan veri yapılarıdır. Biz değişkenlere eriştiğimizde onların isimleri ile değerlerini kullanıyoruz. Fakat mikroişlemci düzeyinde değişken diye bir şey yoktur. Mikroişlemci düzeyinde nesneler bellekteki gözeneklerde bulunan 8 bitlik (1 byte) değerlerden ibarettir. Bellekteki her gözeneğin bir adresi vardır. Mikroişlemci bellekteki herhangi bir nesneye erişmek için ise o nesnenin bellekteki adresini kullanır. Mikroişlemcinin bu işlemleri tamamen elektronik düzeyde yaptığı için bu konunun ayrıntılarına bu kitapta değinilmeyecektir. Bellekteki bu adreslerin programlamadaki karşılığı ise göstericilerdir. Göstericiler ile bir nesne oluşturulduğunda nesnenin adresi elde edilebilir. Ve bu adresteki bellek gözeneğine istenildiği şekilde yeni değerler yerleştirilebilir. Dolayısıyla göstericiler ile bellek istenildiği gibi yönetilebilir.

.NET'in altyapısında gösterici kavramı sıklıkla kullanılırken göstericilerin açıkça kullanımı programcılar için gizlenmiştir. Bunun nedeni gösterici kullanımının programlardaki görülemeyen hatalara sıkça yol açabilmesidir. Özellikle dili yeni öğrenenler için gösterici hataları içinden çıkılmaz bir hale gelebilmektedir. C#'ta göstericiler yerine referans değişkenleri mevcuttur. Referanslar heap bellek bölgesindeki nesnelerin başlangıç adresini tutar. Ancak bu adresin değerine kesinlikle ulaşamayız. Oysa C ve C++ dillerinde stack bölgesindeki değişkenlerin de adreslerine erişebilmemiz mümkündür. Üstelik değişkenlerin adreslerini örneğin 0x965474 şeklinde elde etmemiz bile mümkündür. C ve C++ programcılarına pek yabancı gelmeyecektir bunlar, ancak programlama dünyasına C# ile giren biri için göstericilere neden ihtiyaç duyabileceğimiz pek anlamlı gelmeyebilir. Şunu da söyleyelim ki çok istisnai durumlar dışında göstericilere ihtiyacımız olmayacak, peki bu istisna durumlar nelerdir? Bu durumları aşağıdaki gibi sıralayabiliriz:

a. Geriye Uyumluluk (Backward Compatibility): COM ve WinAPI'deki fonksiyonlar gibi sık sık gösterici kullanan fonksiyonları C# ile programlarımızdan çağırabilmek için parametre olarak gösterici alan fonksiyonlara gösterici göndermemiz gerekebilir. Eğer C#'ta gösterici kullanımına izin verilmemiş olsaydı tür uyumsuzluğu yüzünden gösterici kullanan eski COM ve DLL'lere erişebilmemiz mümkün olamazdı.

b. Performans: C ve C++ dillerinin günümüze kadar çok popüler olmasının altında yatan nedenlerden biri de belleğe direkt erişimi sağladığı için performansın inanılmaz derecede yükselmesidir. Bizim için performansın çok önemli olduğu yerlerde gösterici kullanmamızdan daha doğal birşey olamaz.

c. Alt Seviye İşlemler: Donanım arayüzleri ile direkt bir ilişki içinde olacak programlarda göstericilerin kullanımı mecburi gibidir. Bazen de belleğe kullanıcıların direkt erişebilmesi gereken programlar olabilir. Bu durumlarda da göstericilerden faydalanabiliriz.

d. Hata Ayıklayıcı Programlar: Debugger dediğimiz hata ayıklayıcı programları yazabilmek için mutlaka belleğe direkt erişimin olması gerekir. Bu tür programlar göstericiler kullanılarak etkili biçimde yazılabilir.

Bütün bunların yanında gösterici kullanmanın birçok sıkıntıyı da beraberinde getireceği kesindir. Öncelikle kullanımının zor olması ve kestirilmesi zor olan hatalara yol açabilmesi bu sıkıntıların en başında gelenidir. Zaten C#'ta gösterici kullanacağımız zaman kod bloklarını **unsafe** (emniyetsiz) anahtar sözcüğü ile işaretlememiz gerekir. Aksi halde program derlenemeyecektir. Normal bir metot içinde gösterici kullanımı yasaktır.

Göstericiler ile ilgili temel işlemlere geçmeden önce göstericileri kullanabileceğimiz yani unsafe anahtar sözcüğünü kullanabileceğimiz yerleri inceleyelim.

1. unsafe olarak işaretlenen sınıfların bütün metotlarında gösterici kullanabiliriz.

```
unsafe class Sınıf
{

}
```

2. Normal bir metot içinde herhangi bir bloğu unsafe olarak aşağıdaki gibi işaretleyip dilediğimiz gibi gösterici kullanabiliriz. unsafe bloklarının dışında ise gösterici kullanamayız.

```
int NormalMetot(int a, string str)
{
 unsafe
 {

 }
}
```

3. Normal bir metodu unsafe olarak işaretleyip sadece o metodun içinde de gösterici kullanabiliriz.

```
unsafe int NormalMetot(int a, string str)
{

}
```

4. Bir sınıfın üye değişkenlerinden biri unsafe olarak işaretlenip gösterici olarak bildirilebilir. Ancak bir metot içinde yerel bir gösterici tanımlanamaz. Yerel bir gösterici tanımlamak için unsafe olarak işaretlenmiş metot ya da blok kullanılır.

```
class Sınıf
{
 unsafe char *ptr;
}
```

Gösterici Bildirimi ve Gösterici Operatörleri

C#'ta göstericiler ayrı bir veri türüdür. Bu veri türü ile nesnelerin bellekteki adresleri tutulur. Örneğin, bellekteki 4 byte'lık bir alanda bulunan int türünden bir nesnenin ilk byte'ının adresi int türünden bir göstericide tutulur. Aynı şekilde 2 byte'lık char türünden bir nesnenin bellekteki ilk byte'ının adresi char türünden bir göstericide tutulur. Bütün değer tipleri ile göstericiler bildirilebilir. Bu değer tipleri bildiğiniz gibi int, byte, char gibi tiplerdir. Aynı zamanda yapılarda değer tipleri olduğu için yapı türünden göstericiler de bildirebiliriz. Sınıf türünden nesneler için gösterici bildirmek mümkün değildir. Çünkü sınıflar referans tipleridir. Zaten bellekteki bir sınıf nesnesine ulaşmak için derleyici bizden gizli olarak o nesnesinin bellekteki adresini kullanır.

Aşağıda int türünden bir gösterici ile char türünden bir göstericinin nasıl bildirildiğini görüyorsunuz. Gösterici bildirimi C ve C++ dillerinde olduğu gibi bildirilen gösterici türünün yanına * karakteri eklenerek yapılır.

```
char* ptr1;
int* ptr2;
```

Şimdi bu deyimlerin ne anlama geldiğine bakalım. Yukarıda bildirilen ptr1 ve ptr2 göstericileri 4 byte'lık verilerdir. Aslına bakarsanız C#'taki bütün göstericiler 4 byte büyüklüğündedir. O zaman char türünden olan ptr1 göstericisi ile int türünden olan ptr2 göstericisi arasındaki fark nedir? Madem her ikisi de 4 byte büyüklüğünde neden türleri farklı? Bu soruların cevabını gösterici aritmetiğini anlatırken daha net bir şekilde anlayacaksınız ama şimdi kısaca açıklamakta fayda var:

Bütün göstericiler iki bileşen içerir. Bu bileşenlerden birincisi adres bileşenidir. Adres bileşeni nesnenin bellekte bulunduğu yerin başlangıç adresidir. İkinci bileşen ise tür bileşenidir. Bu bileşen ise ilgili adresteki nesneye ulaşmak istediğimizde bellekten ne kadarlık bilgi okunacağını sağlar. Örneğin, int türünden bir adresteki bilgiyi okumak istediğimizde 4 byte'lık bir bilgi okunacaktır, aynı şekilde char türünden bir gösterici ile 2 byte'lık bir bilgi okunacaktır. Göstericilerde eğer tür bilgisi bulunmamış olsaydı belleğin herhangi bir adresindeki değere ne şekilde ulaşılacağı bilinemezdi.

Birden fazla aynı türden gösterici bildirmek için normal veri türlerinde olduğu gibi virgül operatörü kullanılabilir. Örneğin int türünden iki gösterici aşağıdaki gibi bildirilir.


```
int* ptr1, ptr2;
```

C ve C++ dillerinde bu şekildeki bir bildirimde int türden ptr1 ve int türünden ptr2 nesnesi oluşturulur. Ancak C# dilinde göstericiler tamamen ayrı bir veri türü olarak ele alındığı için yukarıdaki deyim ile iki tane int türünden gösterici bildirmiş oluruz. Kısacası yukarıdaki deyim ile aşağıdaki deyimler eş değerdedir.

```
int* ptr1;
int* ptr2;
```

Göstericilerle ilgili temel işlemlere geçmeden önce gösterici operatörlerlerini inceleyeceğiz. Gösterici operatörleri ile bir nesnenin bellekteki adresi elde edilebilir, aynı şekilde gösterici operatörleri ile bilinen bir adrese bir değer yerleştirilebilir.

Göstericileri sabit ifadeler ile de bildirebiliriz. Örneğin bellekteki 123874 adresini işaret eden double türden bir gösterici aşağıdaki gibi bildirilir.

```
double* ptr = (double*)123874;
```

Göstericilerin bu şekilde bildirilmesi çok tehlikelidir. Çünkü sabit olarak belirttiğimiz adreste ne olduğunu bilmiyoruz. Bilmediğimiz bir adres ile oynamak benzine ateşle yaklaşmak gibidir, zira elimizdeki adres sabiti üzerinde çalıştığımız işletim sisteminin kullandığı bellek bölgesi bile olabilir.

& Operatörü

Adres operatörü olarak bilinen bu operatör değişkenlerin veya nesnelerin bellekte bulundukları adresleri elde etmek için kullanılır. Bu operatör hangi tür değişkenle kullanılırsa o türden bir gösterici üretilir. Örneğin aşağıda int türden bir göstericiye daha önceden tanımlanan int türden bir nesnenin bellekteki adresi atanmaktadır.

```
int a = 5;
int* ptr1;
ptr1 = &a;
```

Aynı şekilde char türünden bir göstericiye bellekte bulunan char türünden bir nesnenin adresi aşağıdaki gibi atanır.

```
char a = 'A';
char* ptr1;
ptr1 = &a;
```

& operatörünün operandı değer tipinde bir nesne olmalıdır.

* Operatörü

İçerik operatörü olan *, bir adresteki bilgileri elde etmek için ya da o adresteki bilgiyi değiştirmek için kullanılır. * operatörü hangi tür ile kullanıldıysa bellekten okunacak bilgi miktarı ya da yazılacak bilgi miktarı o türün büyüklüğü ile aynıdır. Örneğin içerik operatörü ile char türden bir adresteki değeri değiştirmek için bellekte 2 byte'lık bir bölge değiştirilir. Çünkü char türü iki byte büyüklüğündedir. Aşağıda double türden bir adres olan a nesnesinin adresine içerik operatörü ile yeni değer atanıyor.

```
double a = 5;
double * ptr1;
ptr1 = &a;
*ptr1 = 10;
```

Yukarıda a nesnesinin adresi double türünden bir göstericiye atandıktan sonra içerik operatörü kullanılarak bu adresteki değer 10 olarak değiştiriliyor. Şimdi aklımıza şöyle bir soru gelebilir: double türünden ptr1 göstericisinin sayısal bileşeni belleğin bir byte'lık adres dilimini göstermesine rağmen 8 byte büyüklüğünde olan double türünden bir değer bu adrese nasıl yerleştirildi? Bu sorunun cevabı göstericilerin ikinci bileşeni olan tür bilgisinde saklıdır. Biz içerik operatörü ile hangi türden adrese erişiyorsak bellekte erişilen byte miktarı bu türün byte miktarı kadardır. Örneğin yukarıdaki a nesnesinin bellekteki adresi 0x568740 ise içerik operatörünü bu adres ile kullandığımızda 0x568740 ile 0x568747 adresleri arasındaki byte değerlerini değiştirmiş oluruz.

Adresler genellikle onaltılık sayı sisteminde gösterilir. Bir adres sabitinin onaltılık sayı sisteminde yazılmış olduğunu belirtmek için 0x öneki kullanılır.

* operatörünün operandı bir adres olmak zorundadır.

Şimdi adres ve içerik operatörlerinin çalışabilir bir uygulamada test edelim. Aşağıdaki örneği yazın.

```
using System;

class Gosterici
{
    static void Main()
    {
        unsafe
        {
            double a = 5;
            double* ptr1;
```



```
            ptr1 = &a;
            *ptr1 = 10;
            Console.WriteLine(a);

            char c = 'A';
            char* ptr2;

            ptr2 = &c;
            *ptr2 = 'S';
            Console.WriteLine(c);
        }
    }
}
```

Bu programı şu ana kadar derlediğimiz şekilde derleyemeyiz. Bunun sebebi programın içinde unsafe kod bloğunun bulunmasıdır. Bu programı derleyebilmek için komut satırı derleyicisine programın içinde unsafe bloğunun olduğunu belirtmemiz gerekir. Komut satırına

csc /unsafe KaynakKod.cs

ya da

csc -unsafe KaynakKod.cs

yazarak programı derleyebilirsiniz. Eğer Visual Studio.NET kullanıyorsanız projeye sağ tıklayıp proje özelliklerine gelip Build kısmından "Allow unsafe code blocks" kısmını true olacak şekilde değiştirin.

Komut satırı derleyicisi ile programın nasıl derlendiği ve programın ekran çıktısı aşağıdaki ekran görüntüsünde verilmiştir.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum12\kod>csc /unsafe Gosterici.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

H:\cskitap\kitap\Bolum12\kod>Gosterici
10
S

H:\cskitap\kitap\Bolum12\kod>
```

Programın sonucundan da görüldüğü gibi a ve c nesnelerinin değerleri değişmiştir. Çünkü bu nesnelerin bulunduğu adresler & operatörü ile elde edilip, * içerik operatörü ile bu adreslerdeki değerler değiştirilmiştir. Göstericilerin kullanım mantığı

referansların kullanım mantığı ile hemen hemen aynıdır. Tek farkı referansların heap bellek bölgesindeki adresleri göstericilerin ise stack bellek bölgesindeki adresleri temsil etmesidir.

C# tür güvenliğine büyük önem vermektedir. Özellikle belleğe direkt erişim söz konusu ise bunun önemi daha da artmaktadır. Örneğin aşağıdaki deyimleri içeren bir program düşünün.

```
int* ptr1;
*ptr1 = 50;
```

Bu deyimleri içeren bir program derlenemeyecektir. Çünkü ptr1 göstericisinde hangi adresin tutulduğu belli değildir. Bu yüzden adresi belli olmayan bellek bölgesine bir değer yerleştirmek imkansızdır. C ve C++ dillerinde bu kullanım tamamen geçerlidir. Ama gelin bir de bunun sakıncasına bir göz atalım. ptr1 göstericisi bildirildiğinde ptr1'de rastgele bir adres değeri bulunmaktadır. Bu adres de rastgele bir adres olduğu için, o an bellekte çalışan kendi programımızdaki bir değişkenin adresi bile olabilir. Ya da sistemde çalışan başka bir prosesteki elemanların adresleri de olabilir. Kaynağını bilmediğimiz bir adresteki değeri * operatörü ile değiştirdiğimizde ise hiç tahmin edemeyeceğimiz çalışma zamanı hataları alabiliriz. İşin kötüsü adresler rastgele olduğu için programımızın test aşamasında bu hatalar oluşmayabilir. Adresler nasıl rastgele ise hataların oluşması da rastgeledir. Programımız piyasada iken böyle bir hatanın farkına varılması iş maliyetlerini ne kadar artıracağını siz tahmin edin artık. Bütün bu dezavantajlar göstericilerin asla gereksiz olduğu anlamına gelmemelidir. Sadece göstericileri kullanırken daha dikkatli davranmamız gerektiğinin göstergesidir. Ama yine de C# bu tür kullanımları tamamen ortadan kaldırmaktadır. İlk değer verilmeyen bir göstericiye içerik operatörü ile erişilemez.

Örneğin:

```
using System;

class Gosterici
{
      static void Main()
      {
          unsafe
          {
              int* ptr1;
              *ptr1 = 50;
          }
      }
}
```

programı derleme zamanında "Use of unassigned local variable 'ptr1'" yani "Değeri atanmamış ptr1 nesnesi kullanımı" hatası verilecektir.

Göstericiler arasında diğer türlerde olduğu gibi dönüşümler de yapılabilir. Örneğin int türünden bir gösterici, char türünden bir göstericiye aşağıdaki gibi dönüştürülebilir.

```
char c = 'A';
int i = 10;

char* ptr1 = &c;
int* ptr2 = &i;
ptr1 = (char*)ptr2;
```

Göstericiler arasındaki tür dönüşümü açıkça belirtilmelidir. Mesela aşağıdaki gibi int türünden bir göstericiye char türünden bir göstericiyi bilinçsiz olarak atamak derleme zamanında hataya yol açacaktır.

```
char c = 'A';
int i = 10;

char* ptr1 = &c;
int* ptr2 = &i;
ptr2 = ptr1;
```

Göstericiler arasında yapılan tür dönüşümünde göstericinin sayısal değeri değişmez. Değişen şey göstericinin tür bileşenidir. Bütün göstericilerin 4 byte büyüklüğünde olduğunu düşünürsek dönüşüm sonrasında göstericideki sayısal değerin değişmesi olanaksızdır.

Bunu daha iyi görmek için göstericilerin sayısal değerlerini ekrana yazdırmayı deneyelim. Bir göstericinin sayısal değerini yani adres bileşenini elde etmek için göstericiyi tamsayı türlerinden birine dönüştürmemiz gerekir. Göstericiler tamsayı türlerine aşağıdaki gibi dönüştürülebilir.

```
using System;

class Gosterici
{
      static void Main()
      {
          unsafe
          {
              char c = 'A';
              int a = 10;

              char* ptr1 = &c;
              int* ptr2 = &a;

              uint cAdres = (uint)ptr1;
              long aAdres = (long)ptr2;
```

```
            Console.WriteLine("c' nin adresi :
                                        {0:X}",cAdres);
            Console.WriteLine("a' nın adresi :
                                        {0:X}",aAdres);
        }
    }
}
```

Bu programda char türünden ve int türünden göstericilerin sayısal bileşenlerini elde etmek için

```
uint cAdres = (uint)ptr1;
int aAdres = (int)ptr2;
```

deyimleri ile açıkça belirterek tür dönüşümü yapılmaktadır. Göstericiler ile diğer bütün tamsayı türleri arasında da dönüşüm yapmak mümkündür. 4 byte büyüklüğündeki uint tipi bir göstericinin adres bileşeninde tutabileceği bütün adresleri alabileceği için bu dönüşüm için genellikle uint ya da daha büyük bir tamsayı veri tipi kullanılır. int türü bir göstericinin adres bileşenini elde etmek için kullanılabilir ancak çok tehlikelidir. Zira int türü 4 byte olmasına rağmen bir adresi ifade edebilecek değerler ancak 2 byte'lık bir bilgi kadar olabilir. Çünkü int türü işaretli türdür yani negatif değerler de içerebilir. Negatif sayılar bir adres sabiti olamayacağı için adres bileşenlerini int türünde elde etmek adres bileşeninin yanlış elde edilmesine sebep olabilir.

Yukarıdaki program derlendiğinde ekrana a ve c nesnelerinin bellekteki adresleri onaltılık sayı sisteminde yazılacaktır.

Bir sayıyı onaltılık sayı sisteminde yazmak için "X" biçim belirleyicisini kullandığımızı hatırlayın.

Göstericilerin adres bileşenlerini bu dönüşümü yapmadan elde etmek mümkün değildir. Zaten çoğu durumda bir göstericiyi adres bileşeni gibi kullanmak derleme hatasına yol açar. Örneğin aşağıdaki örnekte yapıldığı gibi WriteLine() metodu ile bir göstericinin sayısal bileşeni ekrana yazdırılamaz.

```
using System;

class Gosterici
{
    static void Main()
    {
        unsafe
        {
            int a = 10;
            int* ptr2 = &a;
```



```
            Console.WriteLine(ptr2);
        }
    }
}
```

Göstericilerin adres bileşenini elde etmek için yapılan bu dönüşümün açıkça belirtilmesi gerekir. Aksi halde derleme hatası alınır.

Göstericilerle ilgili diğer bir önemli yapı ise **void** göstericilerdir. void olan göstericilerde tür bilgisi saklanmamaktadır. void göstericilere herhangi bir türden gösterici atanabilir. void göstericiler daha çok eskiden yazılmış API fonksiyonlarında void parametre alan fonksiyonları programlarımız içinden çağırmak için kullanılır. void göstericilerin kullanımına aşağıda bir örnek verilmiştir.

```
int* ptr1;
void * pVoid;
pVoid = ptr1;
```

void göstericilerinin tür bilgisi olmadığı için bütün gösterici türleri void göstericilere atanabilir.

sizeof Operatörü

sizeof operatörü temel türlerin ve yapıların bellekte ne kadar alan kapladıklarını verir. Örneğin sizeof(int) = 4, sizeof(char) = 2 ' dir. sizeof operatörünü sınıflar için kullanamayız ancak bildirdiğimiz yapılar için kullanabiliriz. Sizeof operatörü daha çok yapı türleri için kullanılır. Çünkü temel veri türlerinin büyüklüğü sabit olduğu için zaten herkes tarafından bilinmektedir.

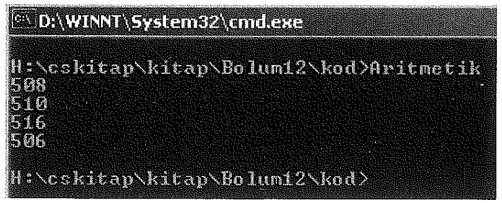
Gösterici Aritmetiği ve Dizi İşlemleri

Göstericilerin adres bileşenlerine sabit tamsayı değerleri ekleyebiliriz, aynı şekilde göstericilerin adres bileşeninden sabit bir tamsayı değerini çıkarabiliriz. Ancak göstericilere uygulanan bu toplama ve çıkarma işlemleri biraz farklıdır. Göstericilere sabit değerlerin eklenmesi ya da bir değerin göstericiden çıkarılması, göstericideki tür bileşeni ile yakından ilgilidir. Bir göstericinin değerini bir artırmak göstericinin adres bileşenini, göstericinin türünün içerdiği byte sayısı kadar artırmak demektir. Aynı kural çıkarma işlemi içinde geçerlidir. Örneğin int türünden bir gösterici ile 1 sayısını toplamak göstericinin adres bileşenini 4 artırmak anlamına gelir. Çünkü int türü 4 byte büyüklüğündedir. Aynı şekilde int türünden bir göstericiden 1 sayısını çıkarmak göstericinin adres bileşenini 4 eksiltmek anlamına gelir. Göstericilerle yapılan bu tür aritmetik işlemlerin tamamına gösterici aritmetiği denilmektedir.

Gösterici aritmetiğini daha yakından görmek için aşağıdaki programı yazın ve sonucunu inceleyin.

```
using System;

class Gosterici
{
      unsafe static void Main()
      {
          int* ptr1 = (int*)500;
          char* ptr2 = (char*)500;
          double* ptr3 = (double*)500;
          byte* ptr4 = (byte*)500;

          ptr1 += 2;
          ptr2 += 5;
          ptr3 += 2;
          ptr4 += 6;

          Console.WriteLine((uint)ptr1);
          Console.WriteLine((uint)ptr2);
          Console.WriteLine((uint)ptr3);
          Console.WriteLine((uint)ptr4);
      }
}
```

Programı /unsafe argümanı ile derleyip çalıştırdığımızda aşağıdaki ekran görüntüsünüz elde ederiz. Programda Main() metodunun unsafe olarak işaretlendiğine dikkat edin.



Programın çıktısından da görüldüğü üzere int türünden bir göstericiye 2 sayısını eklemek göstericinin adres bileşenini 2*4=8 kadar artırmıştır. Aynı şekilde char türünden bir göstericiye 5 değerini eklemek göstericinin adres bileşenini 5*2 =10 kadar artırmıştır. Toplama yerine çıkarma işlemi yapılmış olsaydı bu sefer aynı oranda adres bileşeni eksiltilmiş olacaktı.

Göstericiler üzerinde sadece tamsayılarla aritmetik işlemler yapılabilir. Göstericiler ile aşağıdaki aritmetik operatörleri kullanabiliriz.

```
+ , - , -- , ++ , -=, +=
```

void göstericilerde herhangi bir tür bilgisi olmadığı için bu tür göstericiler üzerinde aritmetik işlemler yapılamaz. Çünkü void türünden bir göstericiye örneğin 1 eklemek istediğimizde göstericinin adres bileşeninin kaç byte öteleneceği belli değildir.

Göstericiler üzerinde yapılabilecek diğer bir önemli işlem de iki göstericinin birbirinden çıkarılmasıdır. Gösterici türleri aynı olmak şartıyla iki göstericiyi birbirinden çıkarabiliriz. Ancak iki göstericinin çıkarılması sonucunda üretilen değer bir gösterici türü değildir. İki gösterici arasındaki fark, adres bileşenlerinin sayısal farkının gösterici türlerinin büyüklüğünden kaç byte edeceğidir. Diğer bir deyişle adres bileşenlerinin sayısal farkı alınıp gösterici türünün byte miktarına göre bir değer belirlenir. İki göstericinin farkı long türünden bir değer üretir. İki göstericinin farkına örnek verecek olursak, int türünden 5008 adres ile int türünden 5000 adresinin farkı (5008-5000) % sizeof(int) tir. Yani sonuç long türünden 2'dir. Aşağıdaki programı yazarak sonucu görebilirsiniz.

```
using System;

class Gosterici
{
      unsafe static void Main()
      {
          int* ptr1 = (int*)500;
          int* ptr2 = (int*)508;

          long fark=ptr2 - ptr1;
          Console.WriteLine(fark);
      }
}
```

Diğer bir ilginç nokta iki göstericinin adres bileşenlerinin farkı gösterici türlerinin büyüklüğünün tam katı olmadığında görülür. Örneğin ptr2 göstericisini bildirimini

```
int* ptr2 = (int*)507;
```

şeklinde değiştirdiğinizde bu sefer ekrana 1 yazdığını görürsünüz. Buradan çıkarmamız gereken sonuç iki göstericinin farkı adres bileşenlerinin sayısal farkının olmamasıdır.

İki göstericinin farkı long türünden bir değer üretir. Bu yüzden iki göstericinin farkı açıkça bir tür dönüşümü yapılmadıkça long türünden küçük türden olan değişkenlere atanamaz.

Göstericiler ile kullanılabilecek diğer operatörler ise ==, < ve > gibi karşılaştırma operatörleridir. Bu operatörler iki göstericinin adres bileşenini karşılaştırıp true ya da false değeri üretirler. Karşılaştırma operatörleri göstericiler için çok istisnai durumlar dışında anlamlı değildir. Bu istisna durumlardan biri göstericileri kullanarak dizi işlemleri yaptığımızda görülür.

fixed Anahtar Sözcüğü

Bildiğiniz gibi C#'ta tanımladığımız referans değişkenleri, heap bellek bölgesindeki adresleri temsil ederler. Ancak biz adresler yerine nesnenin ismini kullanırız. Gereksiz bilgi toplayıcısı (garbage collector) bellek optimizasyonu açısından heap bellek bölgesindeki nesnelerin yerlerini her an değiştirebilir. Bu yer değişiminden bizim haberimiz olmaz, çünkü nesnenin yeri değiştiği anda bu nesneye referans olan stack bellek bölgesindeki değişkenin adres bileşeni de değiştirilir. Dolayısıyla biz aynı referans ile farklı bellek bölgesini isteğimiz dışında kullanmış oluruz. Ancak bazı durumlarda gereksiz nesne toplayıcısına bir nesnenin adresini değiştirmemesi için ikna etmek durumunda kalırız. Bu, özellikle sınıf nesnelerinin üye elemanlarından birinin adresi ile işlem yapmamız gerektiği durumlarda karşımıza çıkar. Bir değişkenin adresinin belirlenen bir faaliyet alanı boyunca değişmeden kalması için bunu gereksiz bilgi toplayıcısına bildirmemiz gerekir. Bunun için **fixed** anahtar sözcüğü kullanılır.

Zaten fixed anahtar sözcüğünü kullanmadan referans türünden nesnelerin üye elemanlarının adreslerini elde etmemiz mümkün değildir. Üye elemanlarının adreslerini elde edemediğimiz bu tür nesnelere **managed type** (**yönetilen tip**) denilmektedir. Buna göre sınıflar managed type kapsamına girmektedir.

Aşağıdaki programda ManagedType isimli sınıfın int türünden olan x elemanının adresi bir göstericiye atanmak isteniyor.

```
using System;

class ManagedType
{
      public int x;
      public ManagedType(int x)
      {
          this.x = x;
      }
}

class Gosterici
{
      unsafe static void Main()
      {
          ManagedType mt = new ManagedType(5);

          int* ptr1 = &(mt.x);
      }
}
```

ManagedType sınıfının x elemanı değer tipi olmasına rağmen mt nesnesi üzerinden x değişkeninin adresi elde edilememektedir. Çünkü x değişkeninin adresi gereksiz nesne toplayıcısı tarafından her an değiştirilebilir. Eğer yukarıdaki kod geçerli olmuş olsaydı x değişkeninin adresi değiştiği anda ptr1 göstericisi nereye ait olduğu bilinmeyen bir adres bilgisi taşıyor olacaktı. x değişkeninin bir blok içerisinde sabit adreste olmasını istiyorsak aşağıdaki gibi **fixed** anahtar sözcüğünü kullanmalıyız.


```
using System;

class ManagedType
{
      public int x;
      public ManagedType(int x)
      {
          this.x = x;
      }
}

class Gosterici
{
      unsafe static void Main()
      {

          ManagedType mt = new ManagedType(5);

          fixed(int* ptr1 = &(mt.x))
          {
              //x' in adresi bu blokta asla değişmez.
          }
      }
}
```

Yukarıdaki **fixed** ile işaretlenmiş blokta x'in adresinin değişmeyeceği garanti altına alınmıştır. Birden fazla değişkeni fixed olarak işaretlemek için aşağıdaki gibi bir kullanım geçerli kılınmıştır.

```
ManagedType mt1 = new ManagedType(5);
ManagedType mt2 = new ManagedType(5);

fixed(int* ptr1 = &(mt1.x))
fixed(int* ptr1 = &(mt2.x))
{
      //x' in adresi bu blokta asla değişmez.
}
```

Öte yandan bir fixed bildirimi içinde adreslerinin değişmesini istemediğimiz elemanları virgül ile ayırarak aşağıdaki gibi de bildirebiliriz.

```
ManagedType mt1 = new ManagedType(5);
ManagedType mt2 = new ManagedType(5);

fixed(int* ptr1 = &(mt2.x), ptr2 = &(mt2.x))
{
      //x' in adresi bu blokta asla değişmez.
}
```

Göstericiler ile Dizi İşlemleri

Diziler konusunda anlatıldığı gibi C#'ta tanımladığımız diziler System.Array sınıfı türündendir. Yani bütün diziler managed type kapsamına girerler. Bu yüzden tanımladığımız bir dizinin herhangi bir elemanının adresini fixed bloğu kullanmadan bir göstericiye atayamayız. Bu yüzden göstericiler ile dizi işlemleri yaparken ya fixed blokları kullanıp gereksiz nesne toplayıcısını uyarmalıyız ya da **stackalloc** anahtar sözcüğünü kullanarak kendimiz **unmanaged type (yönetilemeyen tip)** dizileri oluşturmalıyız. Her iki durumu da birazdan inceleyeceğiz.

Diziler konusunu anlatırken dizi elemanlarının bellekte ardışıl bulunduğunu söylemiştik. O halde bir dizinin elemanlarını elde etmek için dizinin ilk elemanının adresini ve dizinin boyutunu bilmemiz yeterlidir. System.Array sınıfı dizilerini yani şu ana kadar gördüğümüz dizilerin elemanlarına göstericiler yardımıyla rahatlıkla ulaşabiliriz. Bunun için ilk olarak fixed ile işaretlenmiş bir blok içerisinde dizinin ilk elemanının adresini bir göstericiye atamalıyız. Ardından göstericinin değerini bir döngü içinde birer birer artırdığımızda her döngüde dizinin bir sonraki elemanına ulaşmış oluruz. Dizilere bu şekilde erişebilmemizi sağlayan ise gösterici aritmetiğidir. Tabi dizilerin elemanlarının bellekte ardışıl bulunması da bu işlemi bu şekilde yapmamızı sağlayan ilk etkendir.

Şimdi yönetilen türden (managed type) bir dizinin elemanlarına nasıl eriştiğimizi bir örnek üzerinde inceleyelim.

```
using System;

class Gosterici
{
      unsafe static void Main()
      {

          int[ ] a = { 1,2,3,4}
;

          fixed(int* ptr = &a[ 0] )
          {
              for(int i=0; i<a.Length; ++i)
                    Console.WriteLine(*(ptr+i));
          }

      }
}
```

Programı derlediğinizde dizinin elemanlarının

```
1
2
3
4
```


şeklinde ekrana yazdırıldığını görürsünüz. (programı derlerken /unsafe argümanını kullanmayı unutmayın). Programın kritik noktası

```
fixed(int* ptr = &a[ 0] )
```

satırı ile dizinin ilk elemanın adresinin elde edilmesidir. Zira dizinin diğer elemanları da sırayla ilk elemandan itibaren her eleman için adres değeri 4 byte artacak şekildedir. Diğer önemli nokta ise for döngüsü içindeki

```
*(ptr+i)
```

ifadesidir. Bu ifade ile her döngüde ptr göstericisinin adres bileşeni, döngü değişkeni kadar artırılıyor, sözgelimi döngü değişkeni 1 ise ptr'nin adres bileşeni 4 artırılıyor. Bu da dizinin ikinci elemanının bellekte bulunduğu adrestir. İçerik operatörü ile bu adrese erişildiğinde ise dizinin elemanı elde edilmiş olur.

Gösterici dizileri ile ilgili diğer önemli nokta göstericilerin indeksleyici gibi kullanılabilmesidir. Örneğin yukarıdaki örnekte bulunan

```
*(ptr+i)
```

ifadesini

```
ptr[ i]
```

şeklinde değiştirebiliriz. Buradan aşağıdaki eşitlikleri çıkarabiliriz.

```
*(ptr+0) == ptr[ 0]
*(ptr+1) == ptr[ 1]
*(ptr+2) == ptr[ 2]
*(ptr+3) == ptr[ 3]
```

ptr[i] bir nesne belirtirken, (ptr+i) bir adres belirtir.

Dizilerin isimleri aslında dizilerin ilk elemanının adresini temsil etmektedir. Örneğin aşağıdaki programda bir dizinin ilk elemanının adresi ile dizinin ismi int türünden bir göstericiye atanıyor. Bu iki göstericinin adres bileşenleri yazdırıldığında sonucun aynı olduğu görülmektedir.

```
using System;

class Gosterici
{
      unsafe static void Main()
      {
          int[ ] a = { 1,2,3,4} ;
```

```
        fixed(int* ptr1 = a, ptr2 = &a[ 0] )
        {
            Console.WriteLine((uint)ptr1);
            Console.WriteLine((uint)ptr2);
        }
    }
}
```

Programı derleyip çalıştırdığınızda ekrana aynı sayının alt alta iki kez yazıldığını görürsünüz.

Yönetilen tiplerle çalışmak her ne kadar kolay olsa da bazı performans eksiklikleri vardır. Örneğin, bir System.Array dizisinin bir elemanına erişmek ile stack bölgesinde oluşturacağımız bir dizinin elemanına ulaşmamız arasında zaman açısından büyük bir fark vardır. Bu yüzden yüksek performanslı dizilerle çalışmak için System.Array sınıfının dışında stack tabanlı diziler oluşturmamız gerekir. Stack tabanlı diziler yönetilemeyen dizilerdir. Bu yüzden bu tür dizileri kullanırken dikkatli olmalıyız. Çünkü her an bize tahsis edilmeyen bir bellek alanı üzerinde işlem yapıyor olabiliriz. Ancak yönetilen dizilerde dizinin sınırlarını aşmak mümkün değildir. Hatırlarsanız bir dizinin sınırları aşılınca çalışma zamanında IndexOutOfRangeException istisnai durumu meydana geliyordu. Oysa stack tabanlı dizilerde dizinin sınırları belirli değildir ve tabii ki dizinin sınırlarını aşmak kısıtlanmamıştır. Eğer dizinin sınırları aşılmışsa muhtemelen bu işlem bir hata sonucu yapılmıştır. Hiçbir programcı kendisine ait olmayan bir bellek alanında işlem yapmamalıdır. Aksi halde sonuçlarına katlanması gerekir.

Stack tabanlı diziler **stackalloc** anahtar sözcüğü ile yapılır. stackalloc bize istediğimiz miktarda stack bellek bölgesinde alan tahsis eder ve tahsis edilen bu alanın başlangıç adresini geri döndürür. Dolayısıyla elimizde olan bu başlangıç adresi ile stackalloc ile bize ayrılmış olan bütün bellek bölgelerine erişebiliriz. stackalloc anahtar sözcüğünün kullanımı aşağıdaki gibidir.

```
int * dizi = stackalloc int[ 10] ;
```

Bu deyim ile stack bellek bölgesinde 10*sizeof(int) = 40 bytelık bir alan programcının kullanması için tahsis edilir. Bu alan, dizinin faaliyet alanı bitinceye kadar bizim emrimizdedir. Tahsis edilen bu 40 byte büyüklüğündeki bellek alanının ilk byte'ının adresi ise int türünden gösterici olan dizi elemanına aktarılır. Dolayısıyla dizi göstericisi ile içerik operatörünü kullandığımızda bize ayrılan 10 int'lik alanın ilk elemanına erişmiş oluruz.

stackalloc ile tahsis edilen bellek alanlarının ardışık olması garanti altına alınmıştır.

stackalloc ile alan tahsisatı yapılır. Ancak alan tahsisatı yapılan bellek bölgesi ile ilgili hiçbir işlem yapılmaz. Yani yukarıdaki deyim ile, içinde tamamen rastgele değerlerin

bulunduğu 40 byte'lık bir alanımız olur. Bu alandaki değerlerin rastgele değerler olduğunu görmek için aşağıdaki programı yazın.

```
using System;

class Gosterici
{
    unsafe static void Main()
    {
        int * dizi = stackalloc int[ 10] ;

        for(int i=0; i<10;++i)
            Console.WriteLine("* (dizi+{ 0} ) = { 1} ",i,dizi[ i] );
    }
}
```

Yukarıdaki programda bir gösterici ile bellekteki ardışıl bölgelere indeksleyici operatörü ile nasıl eriştiğimize dikkat edin.

Programı /unsafe argümanı ile beraber derleyip çalıştırdıktan sonra aşağıdaki ekran görüntüsünü elde etmeniz gerekir. Tabi bu değerler rastgele olduğu için sizdeki görüntü tamamen farklı olacaktır. Rastgele değerden kasıt çalışma zamanında Random gibi bir sınıfın kullanılıp rastgele bir sayı üretilmesi değildir. Buradaki sayılar daha önce çalışmış olan programlardan kalan çöp değerlerdir.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum12\kod>StackAlloc
*(dizi+0) = 0
*(dizi+1) = 1244236
*(dizi+2) = 1243328
*(dizi+3) = 1350496
*(dizi+4) = 1243344
*(dizi+5) = 1287174
*(dizi+6) = 1243328
*(dizi+7) = 0
*(dizi+8) = 0
*(dizi+9) = 1243404

H:\cskitap\kitap\Bolum12\kod>
```

```
Console.WriteLine("* (dizi+{ 0} ) = { 1} ",i,dizi[ i] );
```

satırındaki

```
dizi[ i]
```

yerine

```
* (dizi + i)
```

yazmamız hiçbir şeyi değiştirmezdi. Daha önce bu iki kullanımın eşdeğer olduğunu belirtmiştik.

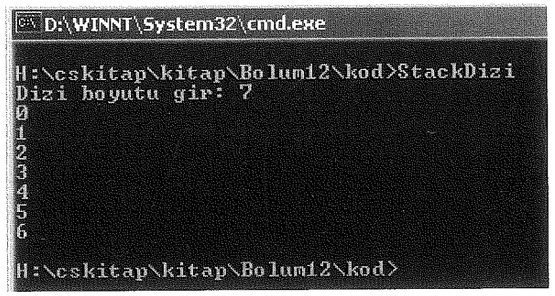
Stack tabanlı dizilerle ilgili bilinmesi gereken en önemli nokta dizinin sınırlarının aşılması ile ilgilidir. Daha önceden de denildiği gibi stack tabanlı dizilerin sınırları aşıldığında herhangi bir uyarı verilmez. Elbette ki program başarıyla derlenir ancak çalışma zamanında bize ait olmayan bir adresin içeriğini değiştirmiş oluruz ki bu da bir programcının başına gelebilecek en tehlikeli durumdur. Örneğin aşağıdaki programda stackalloc ile 10 int türünden nesnelik alan tahsis edilmektedir. Buna rağmen istediğimiz kadar alanı kullanabiliyoruz.

```
using System;

class Gosterici
{
      unsafe static void Main()
      {
          int * dizi = stackalloc int[ 10] ;

          for(int i=0; i<50;++i)
              *(dizi+i) = i;
      }
}
```

stackalloc ile oluşturacağımız dizilerin boyutunun derleme zamanında bilinmesi zorunlu değildir. Örneğin çalışma zamanında kullanıcının belirlediği sayıda elemana sahip olan bir ardışıl bellek bölgesi aşağıdaki programda olduğu gibi tahsis edilebilir.

```
using System;

class StackAlloc
{
      unsafe static void Main()
      {
          Console.Write("Dizi boyutu gir: ");
          uint boyut=0;

          try
          {
              boyut = Convert.ToUInt32(Console.ReadLine());
          }
          catch(FormatException e)
          {
              Console.WriteLine(e.Message);
          }

          int * dizi = stackalloc int[ (int)10] ;

          for(int i=0; i<(int)boyut; ++i)
          {
              *(dizi+i) = i;
              Console.WriteLine(dizi[ i] );
          }
      }
}
```


Bu program ile çalışma zamanında 7 elemanlı stack tabanlı bir dizinin oluşturulduğunu aşağıdaki ekran görüntüsünden görebilirsiniz.



Yapı (struct) Türünden Göstericiler

int, char, double gibi veri türleri aslında birer yapıdır. Bu konunun başında bütün değer tipleri ile gösterici tanımlayabileceğimizi söylemiştik. C# temel veri türlerinin yanısıra kendi bildirdiğimiz yapılar da değer türündendir. O halde bir yapı göstericisi bildirebilmemiz doğal bir durumdur. Yapı göstericilerinin bildirilmesi temel veri türlerinden gösterici bildirimi ile aynıdır. Oradaki kuralların tamamı yapılar için de geçerlidir. Fakat yapı göstericisi bildirmenin bir şartı vardır, o da yapının üye elemanlarının tamamının değer tipi olma zorunluluğudur. Örneğin aşağıdaki yapı göstericisi bildirimi geçersizdir.

```
using System;

struct Yapı
{
      int x;
      char c;
      string s;
      public Yapı(int x,char c, string str)
      {
          this.x = x;
          this.c = c;
          this.s = str;
      }
}

class StackAlloc
{
      unsafe static void Main()
      {
          Yapı yapi = new Yapı(2,'a',"Denem");

          Yapı* pYapı = &yapi;
      }
}
```

Yapı türünden göstericinin bildirilememesinin sebebi yapının yönetilen türden (managed type) bir üye elemanının bulunmasıdır. Bu üye elemanı da doğal olarak string türüdür. Yapı bildiriminden string türünü çıkarıp aşağıdaki gibi gerekli değişiklikleri yaptığımızda Yapı türünden göstericileri bildirebiliriz.

```
using System;

struct Yapı
{
      public int x;
      public char c;

      public Yapı(int x,char c)
      {
          this.x = x;
          this.c = c;
      }
}

class StackAlloc
{
      unsafe static void Main()
      {
          Yapı yapi = new Yapı(2,'a');

          Yapı* pYapı = &yapi;
      }
}
```

Yapı göstericileri üzerinden yapı göstericisinin adresine ilişkin nesnelerin elemanlarına özel bir operatör olan -> operatörü ile erişebiliriz. Örneğin yukarıdaki programın Main() metodunu aşağıdaki gibi değiştirdiğinizde ekrana yapi nesnesinin x ve c elemanları yazdırılacaktır.

```
unsafe static void Main()
{
      Yapı yapi = new Yapı(2,'a');

      Yapı* pYapı = &yapi;

      Console.WriteLine("yapi.x= " + pYapı->x);
      Console.WriteLine("yapi.c= " + pYapı->c);
}
```

'->' operatörüne ok operatörü de denilmektedir. Ayrıca, yukarıdaki programda ok operatörü ile yapının elemanlarına ulaşabilmek için yapının elemanlarını public olacak şekilde değiştirmemiz gerektiğini unutmayın.

Yapının elemanlarına ok operatörü yerine yapı nesnesinin içeriğini * operatörü ile elde edip nokta operatörü ile de ulaşabiliriz. Buna göre yukarıdaki Main() metodu ile aşağıdaki Main() metodu eşdeğerdir.

```
unsafe static void Main()
{
      Yapı yapi = new Yapı(2,'a');

      Yapı* pYapı = &yapi;
      Console.WriteLine("yapi.x= " + (*pYapı).x);
      Console.WriteLine("yapi.c= " + (*pYapı).c);
}
```

Her iki Main() metodunun ürettiği çıktı aynıdır.

Göstericilerin en çok kullanıldığı diğer bir uygulama alanı da karakter işlemleridir. Bir yazıyı karakter dizisi olarak temsil edip yazılar ile ilgili işlemler yapılabilir. C#'taki string türünün altında gerçekleşen olaylar da zaten bundan ibarettir. Karakter dizileri ile ilgili en önemli nokta bir yazıyı char türünden bir göstericiye atayabilmemizdir. Karakter dizileri olan stringlerdeki her bir karakter bellekte ardışıl bulunmaktadır. Dolayısıyla yazıdaki ilk karakterin adresini bildiğimizde yazıdaki bütün karakterlere erişebiliriz. C#'taki string türü managed type olduğu için char türünden bir göstericiye bir yazının ilk karakterinin adresini atamak için **fixed** anahtar sözcüğünü kullanmalıyız. Aşağıdaki programda bir yazının char türünden göstericiye nasıl atandığını ve bu gösterici ile yazıdaki her karaktere ne şekilde erişildiğini görüyorsunuz.

```
using System;

class KarakterDizisi
{
      unsafe static void Main()
      {
          fixed(char* ptr = "Sefer Algan")
          {
              for(int i=0; ptr[i] != '\0'; ++i)
                    Console.Write(ptr[i]);
          }
      }
}
```

Buradaki en önemli nokta ptr[i]'nin '\0' karakteri ile karşılaştırıldığı yerdir. Göstericiler ile bellekte char türünün büyüklüğü kadar ilerlerken yazının nerede sonlandığını bilemeyiz. Bunun için

```
char* ptr = "Sefer Algan"
```

deyimi ile belleğe yerleştirilen 'n' karakterinden sonra null değerini ifade eden '\0' karakter yerleştirilir. Bu karaktere rastlanıldığı zaman yazının sonuna gelmiş bulunuyoruz.

Yansıma ve Nitelikler

Bu kısımda C#'taki farklı bir konuya giriş niteliğinde değinilecektir. Bir tür hakkında çalışma zamanında çeşitli bilgiler elde etmemizi sağlayan Type sınıfını ve bununla bağlantılı olan diğer sınıfları inceleyeceğiz Ayrıca .NET ortamında yazılmış en küçük birim olan Assembly hakkında da bilgi verilecektir. Son olarak metotlar ve diğer elemanlar ile ilgili ekstra bilgi vermemizi sağlayan nitelikleri (attributes) inceleyeceğiz.

Assembly Kavramı ve Assembly Sınıfı

C# ile yazdığımız programlar derlenirken, programdaki elemanlar (metot, sınıf vs) hakkında çeşitli bilgileri içeren metadata dediğimiz bilgiler ile beraber derlenir. CLR birimi, çalışma zamanında ilgili elemanların kullanımı için bu metadata dediğimiz bilgilerden faydalanır. İşte kaynak kodumuzun ve bu metadata bilgilerinin bulunduğu .NET birimine assembly denilmektedir. C#'taki bu assembly'leri temsil eden Assembly sınıfı yardımı ile çalışma zamanında metadata dediğimiz verilere erişmemiz mümkündür. Bu metadata veriler programdaki üye elemanların çeşitli özellikleridir, örneğin bir metodun parametreleri, bir sınıfın üye değişkenleri gibi.

Derlenmiş her birime assembly denilmektedir. Derlenmiş dosyalara exe ve dll uzantılı dosyaları örnek verebiliriz.

Çalışan programımız içinde bir assembly olabileceği gibi birden fazla assembly de olabilir. Programımıza ilişkin bütün assembly'leri elde etmek için AppDomain sınıfını kullanıyoruz. **AppDomain** sınıfının statik CurrentDomain özelliği ile o an üzerinde çalışılan AppDomain nesnesi elde edilir. AppDomain bir uygulama için konteyner gibi çalışır. Assembly'ler bu AppDomain'ler içine yüklenir. Bu yüzden bir AppDomain içinde birden fazla Assembly olabilir. Her bir assembly'yi temsil etmek için System.Reflection isim alanında Assembly sınıfı mevcuttur. Assembly'leri elde etmek için en çok kullanılan yöntemler aşağıdaki gibidir.

1. AppDomain sınıfının GetAssemblies() statik metodu ile programa ilişkin bütün assembly'ler bir Assembly türden diziye aktarılır. Örneğin aşağıdaki programda programa ilişkin Assembly'lerin sayısı ve her assembly dosyasının yeri yazdırılıyor. Assembly sınıfının Location özelliği assembly dosyasının bulunduğu dizini verir.

```
using System;
using System.Reflection;

class YansimaVeAssembly
{
      static void Main()
      {
          AppDomain ad = AppDomain.CurrentDomain;
```



```
          Assembly[] assembly = ad.GetAssemblies();

          Console.WriteLine("Toplam Assembly sayisi :
                                    " + assembly.Length);

          for(int i=0; i<assembly.Length; ++i)
              Console.WriteLine("Assembly yeri:
                                    " + assembly[i].Location );
      }
}
```

Programı derleyip çalıştırdığımızda aşağıdaki ekran görüntüsünden de görüleceği üzere çalışan programımıza iki tane assembly dosyası yüklenmiştir.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum12\kod>AssemblyOrnek
Toplam Assembly sayisi : 2
Assembly yeri: d:\winnt\microsoft.net\framework\v1.0.3705\mscorlib.dll
Assembly yeri: H:\cskitap\kitap\Bolum12\kod\AssemblyOrnek.exe

H:\cskitap\kitap\Bolum12\kod>
```

Bu assembly'lerden biri doğal olarak kendi çalışan programımızın çalıştırılabilir exe dosyasıdır. Diğeri ise .NET sınıf kitaplığını kullandığımız için kullanılan mscorlib.dll isimli assembly'sidir. Kendi oluşturduğumuz bir sınıfı dll şeklinde paketleyip programda kullanmış olsaydık bu sefer çalışan programımıza 3 tane assembly yüklenmiş olurdu.

2. Assembly sınıfının GetExecutingAssembly() statik metodu ile o an çalışan programımıza ilişkin Assembly geri döndürülür. 1. maddede anlatılan örneğe göre bu metot AssemblyOrnek.exe assembly'sini geri döndürecektir. 1. maddedeki Main() metodunu aşağıdaki gibi değiştirerek bunu görebilirsiniz.

```
static void Main()
{
      Assembly asmb = Assembly.GetExecutingAssembly();
      Console.WriteLine("Assembly'nin başlangıç noktası :" +
                                        asmb.EntryPoint);
}
```

Bu Main() metodunu içeren programı derleyip çalıştırdığınızda ekrana

```
Assembly'nin başlangıç noktası :Void Main()
```

yazdığını görürsünüz. Assembly sınıfının EntryPoint özelliği assembly dosyasına ilişkin başlama noktasını verir. Çalıştırılabilir dosyalarda bu genellikle Main() metodudur. Dll şeklindeki Assembly'lerde ise giriş noktası yoktur. Yani bu özellik null değere sahiptir. EntryPoint daha sonra göreceğimiz MethodInfo sınıfı türündendir.

3. Assembly sınıfının LoadFrom(string assembly_yolu) metodu ile bir Assembly dos-

yasının yolu belirtilerek de Assembly sınıfı nesneleri elde edilebilir. Aşağıdaki programda mscorlib.dll dosyasına ilişkin assembly nesnesi elde ediliyor.

```
using System;
using System.Reflection;

class YansimaVeAssembly
{
      static void Main()
      {
          string Yol = @"d:\winnt\microsoft.net\framework\
                                    v1.0.3705\mscorlib.dll";

          Assembly asmb = Assembly.LoadFrom(Yol);
          Console.WriteLine("Assembly' nin başlangıç noktası
                                        :" + asmb.EntryPoint);
      }
}
```

mscorlib.dll dosyası sizin sisteminizde farklı yerde olabilir. .NET Framework'ü kurduğunuz dizini kontrol etmelisiniz.

Programı derleyip çalıştırdığınızda EntryPoint özelliğinin null değerde olduğunu görürsünüz. Çünkü mscorlib.dll assmbly'si çalıştırılabilir değildir.

Bir assembly dosyası genel olarak iki kısımdan oluşur. Birincisi assembly'lerde bulunan bütün türlerle ilgili bilgiler içeren manifest dediğimiz bölümdür. İkinci ise kaynak kodlarımızın aradildeki (**Intermediate Language**) karşılığıdır. Birazdan göreceğimiz **Type** sınıfının manifest dediğimiz bu bölümü kullanan çeşitli metotların assembly'lerdeki tür bilgilerini nasıl elde ettiğini göreceğiz.

Assembly sınıfının en önemli metodu GetTypes() isimli metodudur. Bu metot ile bir assembly'de bulunan bütün türler Type türünden bir diziye aktarılır. Type türünü az sonra detaylı bir şekilde inceleyeceğiz ama şimdilik GetTypes() metodunun kullanımına bir örnek verelim. Aşağıdaki programda programımıza ilişkin assembly'lerde bulunan bütün türlerin toplamı ayrı ayrı ekrana yazdırılmaktadır.

```
using System;
using System.Reflection;
class YansimaVeAssembly
{
      static void Main()
      {
          AppDomain ad = AppDomain.CurrentDomain;

          Assembly[] assembly = ad.GetAssemblies();
```



```
          for(int i=0; i<assembly.Length; ++i)
          {
              Console.WriteLine("Assembly: " +
                                    assembly[ i] .Location );
              Console.WriteLine("Tür sayısı: " +
                                assembly[ i] .GetTypes() .Length);
              Console.WriteLine();
          }
      }
}
```

Programı derleyip çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum12\kod>GetTypes
Assembly: d:\winnt\microsoft.net\framework\v1.0.3705\mscorli
b.dll
Tür sayisi: 1430

Assembly: H:\cskitap\kitap\Bolum12\kod\GetTypes.exe
Tür sayisi: 1

H:\cskitap\kitap\Bolum12\kod>
```

Assembly sınıfının diğer bir metodu GetType(string Tür) ise belirli bir tür ile ilgili Type nesnesini döndürür.

Assembly sınıfının daha birçok metodu bulunmaktadır. Bu bölümün sonuna geldiğinizde bütün bu metotları MSDN kütüphanesinden inceleyip anlayabilecek düzeye geleceksiniz.

Type Sınıfı ve Yansıma Kavramı

Hatırlarsanız operatörler bölümünde **typeof** operatörünün bir türe ilişkin Type türünde bir nesne döndürdüğünü söylemiştik ancak ayrıntılarını incelememiştik. **Type** türünün ayrıntıları bu konuda incelenecektir. Type sınıfı yansıma (reflection) kavramının en temel yapıtaşıdır. Type sınıfı bir assembly'deki tür bilgisine ilişkin bilgileri içerir. Type sınıfında çok sayıda metot bulunmaktadır. Burada bu metotlardan sadece en önemlileri incelenecektir.

Type sınıfı özet(abstract) sınıf olduğu için

```
Type t = new Type();
```

şeklinde bir Type nesnesi tanımlamak geçersizdir. Type türünden nesneler elde et-

mek için genellikle typeof operatörü ve Object sınıfının GetType() metodu kullanılır. Diğer bir yöntem ise Type sınıfının statik metodu olan GetType() metodunu kullanmaktır. Bu elde ettiğimiz Type türünden nesneleri kullanarak bir tür ile ilgili çeşitli bilgiler elde ederiz. Type türü yansıma kavramı için her ne kadar temel bir tür olsa da System.Reflection isim alanında her bir eleman tipine ilişkin bilgileri temsil etmek için birçok sınıf mevcuttur. Bu sınıflar aşağıda açıklamalarıyla beraber listelenmiştir. Bütün bu sınıfların System.Reflection isim alanında bulunduğunu unutmayın.(Type sınıfı System isim alanında bulunur).

1. MemberInfo sınıfı: Bir türdeki bütün elemanları temsil etmek için kullanılır. özet bir sınıf olduğu için new anahtar sözcüğü ile MemberInfo türünden nesneler tanımlanamaz. MemberInfo sınıfının önemli özellikleri aşağıda verilmiştir:

`Type DeclaringType`: Type türünden olan bu özellik ilgili üye elemanının bildirildiği türün Type nesnesidir.

`MemberTypes MemberType`: MemberTypes isimli numaralandırma türündendir. İlgili elemanın ne tür bir eleman olduğunu belirtir. MemberTypes numaralandırması Constructor, Method, Event, Field ve Property sembollerini içerir.

`string Name: İlgili türün ismini belirtir.`

Type sınıfının GetMembers() metodu kullanılarak bir türde bulunan bütün üye elemanlar MemberInfo türünden bir diziye aktarılabilir. Ayrıca GetMember(string str) metodu str ile belirtilen eleman ile ilgili MemberInfo türünden bir nesne döndürür. Aşağıdaki program, System isim alanında bulunan String türünün bütün metotlarını ekrana yazdırmaktadır.

```
using System;
using System.Reflection;

class Yansima
{
      static void Main()
      {
          Type t = typeof(String);

          MemberInfo[] elemanlar = t.GetMembers();

          foreach(MemberInfo mi in elemanlar)
          {
              //Üye eleman metot mu?
              if(mi.MemberType == MemberTypes.Method)
                     Console.WriteLine(mi.Name);
          }
      }
}
```

Programı derleyip çalıştırdığımızda String sınıfının bütün metotlarının isimleri alt

alta yazılacaktır.

2. MethodInfo sınıfı: Bir türdeki metotları temsil etmek için kullanılır. Bu sınıf MemberInfo sınıfından türemiş olan MethodBase sınıfından türetilmiştir. Dolayısıyla birinci maddede anlatılan MemberInfo sınıfının üye elemanları bu sınıf içinde de geçerlidir. MemberInfo sınıfına ek olarak metotların geri dönüş değerinin türünü Type sınıfı nesnesi olarak geri döndüren ReturnType özelliği mevcuttur. Ayrıca 3. maddede göreceğimiz ParameterInfo sınıfı türünden bir diziye geri dönen GetParameters() metodu da MethodInfo sınıfının üyesidir. Bir türe ilişkin metotları temsil eden bir dizi elde etmek için Type sınıfının GetMethods() metodu kullanılır. Örneğin

```
Type t = typeof(String);
MethodInfo[] mi = t.GetMethods();
```

deyimleri ile mi dizisi String türünün metotlarını temsil eder. String türüne ilişkin Type nesnesini typeof operatörünü kullanmadan aşağıdaki gibi de elde edebilirdik.

```
string str = "deneme";
Type t = str.GetType();
```

Type sınıfının GetMethods() metodunun ikinci bir kullanımı daha vardır. Bu metodun prototipi ise aşağıdaki gibidir.

```
MethodInfo[] GetMethods(BindingFlags kriterler)
```

BindingFlags System.Reflection isim alanında bulunan bir numaralandırmadır. Bu numaralandırma parametresi ile GetMethods() metodu sadece kritere uyan metotları MethodInfo dizi şeklinde döndürür. BindingFlags numaralandırmasının en önemli sembolleri aşağıda verilmiştir.

`DeclaredOnly`: Sadece ilgili türde bildirilen metotları belirtir. Yani kalıtım yolu ile gelmiş metotlar alınmaz.

`NonPublic`: public olarak işaretlenmemiş metotları temsil eder.

`Public`: public olarak işaretlenmiş metotları temsil eder.

`Static`: static olarak işaretlenmiş metotları temsil eder.

Type sınıfının bir de GetMethod(string str) metodu vardır. Bu metot, str ile belirlenen bir metoda ilişkin MethodInfo nesnesi döndürür.

Aşağıdaki programda Array sınıfının bütün metotları listeleniyor. Ayrıca IsStatic() metodu ile metodun statik olup olmadığı da test ediliyor.

```
using System;
using System.Reflection;
```

```
class Yansima
{
      static void Main()
      {
          Type t = typeof(Array);

          MethodInfo[] metotlar = t.GetMethods();

          int ToplamMetot = 0;

          foreach(MethodInfo mi in metotlar)
          {
              Console.Write("Metot ismi : ");
              Console.WriteLine(mi.Name);
              Console.Write("Geri dönüş tipi: ");
              Console.WriteLine(mi.ReturnType);
              Console.Write("Metot tipi: ");
              if(mi.IsStatic)
                    Console.WriteLine("Statik(Static)");
              else
                    Console.WriteLine("Örnek(Instance)");

              Console.WriteLine();

              ToplamMetot++;
          }

          Console.WriteLine("Toplam metot sayısı: "+
                                                ToplamMetot);
      }
}
```

Programın ekran çıktısının bir bölümü aşağıdaki gibidir. Bu programı diğer türler için uygulayarak hangi türün hangi metotlarının olduğunu rahatlıkla öğrenebilirsiniz.

```
D:\WINNT\System32\cmd.exe
Metot tipi: Statik(Static)

Metot ismi : Sort
Geri dönüs tipi: System.Void
Metot tipi: Statik(Static)

Metot ismi : Sort
Geri dönüs tipi: System.Void
Metot tipi: Statik(Static)

Metot ismi : Sort
Geri dönüs tipi: System.Void
Metot tipi: Statik(Static)

Metot ismi : Initialize
Geri dönüs tipi: System.Void
Metot tipi: Örnek(Instance)

Metot ismi : GetType
Geri dönüs tipi: System.Type
Metot tipi: Örnek(Instance)

Toplam metot sayisi: 53

H:\cskitap\kitap\Bolum12\kod>
```

Konunun daha iyi anlaşılması açısından bir örnek daha yapalım. Aşağıdaki program

bir assembly'deki bütün türlere ilişkin toplam metot sayısını bulmaktadır. Örnekte mscorlib.dll assembly'sindeki her bir tipteki toplam metot sayısı ekrana yazdırılıyor.

```
using System;
using System.Reflection;

class Yansima
{
      static void Main()
      {

          AppDomain ad = AppDomain.CurrentDomain;
          Assembly[] assembly = ad.GetAssemblies();

          Type[] tipler = assembly[0].GetTypes();

          foreach(Type tip in tipler)
          {
              Console.WriteLine("Tip: " + tip.Name);
              Console.WriteLine("Metot Sayısı: " +
                                    tip.GetMethods().Length);

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

MethodInfo sınıfının ayrıca IsStatic, IsAbstract, IsConstructor, IsPublic, IsPrivate, IsVirtual gibi metot ile ilgili bir kriteri kontrol eden özellikleri de bulunmaktadır. Eğer özellik isminde belirtilen kriterler sağlanıyorsa bu özellikler true değeri döndürür.

3. ParameterInfo sınıfı: Bir metottaki parametreler ParameterInfo sınıfı ile temsil edilir. Bir metodun parametrelerini elde etmek için MethodInfo sınıfının GetParameters() metodu kullanılabilir. Aşağıdaki programda kendi tanımladığımız Deneme isimli sınıftaki bütün metotların ptototiplerini yansıma sayesinde çalışma zamanında nasıl elde ettiğimizi görebilirsiniz.

```
using System;
using System.Reflection;

class Deneme
{
      public static double Toplam(int a, int b)
      {
          return a+b;
      }

      public static string Ekle(int a, int b)
      {
```

```
            return a.ToString() + b.ToString();
        }

        public static double Kare(double a)
        {
            return a*a;
        }
}

class Yansima
{
        static void Main()
        {
            Type t = typeof(Deneme);

            MethodInfo[] metotlar = t.GetMethods();

            Console.WriteLine(t.Name + " türünün metotları\n");

            foreach(MethodInfo metot in metotlar)
            {
                Console.Write(metot.ReturnType.Name + " " +
                                                metot.Name + "(");

                ParameterInfo[] p = metot.GetParameters();

                for(int i=0 ; i < p.Length; ++i)
                        Console.Write(p[i].ParameterType.Name
                                                + " " + p[i].Name +
                                    (i+1 < p.Length ? ", " : ""));

                Console.WriteLine(")\n");
            }
        }
}
```

Programı derleyip çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum12\kod>ParameterInfo
Deneme  türünün metotları

Int32 GetHashCode()

Boolean Equals(Object obj)

String ToString()

Double Toplam(Int32 a, Int32 b)

String Ekle(Int32 a, Int32 b)

Double Kare(Double a)

Void Yaz(Int32 a, Int32 b, String s)

Type GetType()

H:\cskitap\kitap\Bolum12\kod>
```

Gördüğünüz gibi Deneme sınıfında bildirmiş olduğumuz metotların yanısıra Object

sınıfından kalıtım yolu ile gelen metotlar da listelenmiştir. Bu programı kullanarak şu ana kadar gördüğümüz bütün türlere ilişkin metotların prototiplerini inceleyebilirsiniz. Örneğin ParameterInfo sınıfının burada değinmediğimiz diğer metotlarını herhalde merak ediyorsunuzdur!

4. ConstructorInfo sınıfı: Bir türün yapıcı metotları hakkında bilgi verir. Type sınıfının GetConstructors() metodu kullanılarak bir türe ilişkin yapıcı metotlar elde edilebilir. Aşağıdaki programda String türünün yapıcı metotları yazdırılmıştır. Bu programla istediğiniz bir nesneyi nasıl oluşturabileceğinizi yapıcı metotları listeleyerek görebilirsiniz.

```
using System;
using System.Reflection;

class YansimaVeAssemblies
{
        static void Main()
        {
            Type t = typeof(String);

            ConstructorInfo[] yapicilar = t.GetConstructors();

            Console.WriteLine(t.Name + " türünün metotları\n");

            foreach(ConstructorInfo yapici in yapicilar)
            {
                Console.Write(t.Name + "(");

                ParameterInfo[] p = yapici.GetParameters();

                for(int i=0 ; i < p.Length; ++i)
                  Console.Write(p[i].ParameterType.Name + " " +
                        p[i].Name + (i+1 < p.Length ? ", " : ""));

                Console.WriteLine(")\n");
            }
        }
}
```

Programın String türü için çıktısı aşağıdaki gibidir.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum12\kod>ConstructorInfo
String  türünün metotlari

String(Char* value)

String(Char* value, Int32 startIndex, Int32 length)

String(SByte* value)

String(SByte* value, Int32 startIndex, Int32 length)

String(SByte* value, Int32 startIndex, Int32 length, Encoding enc)

String(Char[] value, Int32 startIndex, Int32 length)

String(Char[] value)

String(Char c, Int32 count)

H:\cskitap\kitap\Bolum12\kod>
```

5. PropertyInfo, FieldInfo ve EventInfo sınıfları: Bu sınıflar sırası ile bir türdeki özellikleri, değişkenleri ve olayları temsil etmek için kullanılır. Type sınıfının GetProperties(), GetFields() ve GetEvents() metotları ile bu elemanlar bir dizi şeklinde geri döndürülür. Aşağıdaki programda Windows programlarında sıkça kullanılan butonları temsil eden bir sınıfın olay elemanları listeleniyor. Ayrıca olaya ilişkin olay yöneticisinin türü de ekrana yazdırılıyor.

```
using System;
using System.Reflection;

class YansimaVeAssemblies
{
      static void Main()
      {
          Type t = typeof(System.Windows.Forms.Button);

          EventInfo[] olaylar = t.GetEvents();

          Console.WriteLine(t.Name + " türünün olayları\n");

          foreach(EventInfo olay in olaylar)
          {
              Console.Write("Olay Adı: " + olay.Name);

              Console.WriteLine("("+(olay.IsMulticast ?
                                  "MultiCast" : "") + ")");
              Console.WriteLine("Olay Yöneticisi türü: " +
                                    olay.EventHandlerType);
              Console.WriteLine();
          }
      }
}
```


Programın ekran çıktısının bir bölümü aşağıda verilmiştir.

```
Olay Adi: KeyDown(MultiCast)
Olay Yöneticisi türü: System.Windows.Forms.KeyEventHandler

Olay Adi: KeyPress(MultiCast)
Olay Yöneticisi türü: System.Windows.Forms.KeyPressEventHandler

Olay Adi: KeyUp(MultiCast)
Olay Yöneticisi türü: System.Windows.Forms.KeyEventHandler

Olay Adi: Layout(MultiCast)
Olay Yöneticisi türü: System.Windows.Forms.LayoutEventHandler

Olay Adi: Leave(MultiCast)
Olay Yöneticisi türü: System.EventHandler
```

Diğer sınıflar da şu ana kadar gördüğümüz sınıflar ile hemen hemen aynıdır. Bu sınıfların diğer bütün metotlarını ve üye elemanlarını görmek için kendiniz bir program yazabilir ya da şu ana kadar yazdığımız programları kullanabilirsiniz.

Çalışma Zamanında Metot Çağırma

Bir türe ait metodu şu ana kadar hep derleme zamanında çağırdık. Ancak yansıma ile yani System.Reflection isim alanında bulunan sınıflar ile çalışma zamanında da metot çağırabiliriz. Bunun için ya MethodInfo sınıfının ya da ConstructorInfo sınıfının aşırı yüklenmiş metodu olan

```
object Invoke(object nesne, object[] parametreler)
```

kullanılır. Eğer ConstructorInfo sınıfı üzerinden metodu çağırırsak çalışma zamanında nesne oluşturmuş oluruz. Şimdi Invoke() metodunun parametrelerinin ne anlama geldiğine bakalım.

Birinci parametre ile çalışma zamanında çağrılacak metodun hangi nesne üzerinden çağrılacağı belirtilir. İkinci parametre ise çağrılacak metoda geçirilecek parametreleri içeren bir dizidir.

Çalışma zamanında Invoke() metodu ile çağrılacak metot eğer statik ise birinci parametre null olmalıdır. Aynı şekilde metot parametre almıyorsa Invoke() metodunun ikinci parametresi null olmalıdır.

Aşağıdaki programda bildirmiş olduğumuz statik MethodInvoke() isimli metot kendisine gönderilen 3 parametreye göre ilgili metodu çağırmaktadır. MethodInvoke() metodunun prototipi aşağıdaki gibidir.

```
object MethodInvoke(string metot, string tip, object[] args);
```

Programın kaynak kodu aşağıdaki gibidir.

```
using System;
using System.Reflection;

class Deneme
{
      public static int Topla(int a, int b)
      {
          return a+b;
      }
}

class Yansima
{
      static void Main()
      {
          object[] parametreler = {9,5};

          int a = (int) MetotInvoke("Topla","Deneme",parametrel
er);

          Console.WriteLine(a);
      }

      static object MetotInvoke(string metot,string tip,
 object[] args)
      {
          Type t = Type.GetType(tip);
          if(t==null)
              throw new Exception("Tür bulunamadı.");

          MethodInfo m = t.GetMethod(metot);
          if(m == null)
              throw new Exception("Metot bulunamadı.");

          if(m.IsStatic)
              return m.Invoke(null,args);

          object o = Activator.CreateInstance(t);
          return m.Invoke(o,args);
      }
}
```

Programı derleyip çalıştırdığımızda statik Topla() metodunun başarı ile çağrıldığını görürüz.

```
object o = Activator.CreateInstance(t);
```

ile t türünden çalışma zamanında bir nesne oluşturuluyor ve Invoke() metoduna bu nesnenin referansı parametre olarak geçiriliyor. Bu nesneyi oluşturmamızın sebebi statik olmayan metotların çağrılması için nesne referansına ihtiyaç duyulmasıdır.

Nitelikler (Attributes)

Nitelikler bir metot ya da diğer üye elemanlarla ilgili ekstra bilgiler vermek amacıyla bu elemanların bildiriminden önce yazılan bilgilerdir. System.Reflection sınıfındaki sınıflar kullanılarak bu ek bilgiler çalışma zamanında elde edilebilir. C# dilinde önceden tanımlanmış birçok nitelik vardır. Bu niteliklerden bazılarına bu konuda değineceğiz.

Bir nitelik, eleman bildiriminden önce açılan ve kapanan köşeli parantezler arasına yazılır. Örneğin aşağıdaki metot bildiriminden önce metoda bir nitelik eklenmektedir

```
[Nitelik(Argümanlar)]
public void Metot(int a)
{
}
```

Kendi niteliklerimizi tanımlamaya başlamadan önce C#'ta bulunan bazı önemli nitelikleri ve ne amaçla kullanıldıklarına bakalım.

`Conditional` niteliği: Bu nitelik ile metotların koşullu olarak çağrılıp çağrılmaması sağlanır. Metodun çağrılmasındaki kriter nitelik parantezinde belirtilen sembolün #define önişlemci komutu ile tanımlanıp tanımlanmamasıdır. Bu nitelik System.Diagnostic isim alanında bulunur. Conditional niteliği özelliklere ve üye değişkenlere uygulanamaz. Yani sadece metotlara uygulanabilir. Bu niteliğin kullanımına bir örnek verelim. Aşağıdaki programda #define önişlemci komutu ile tanımlanmış ENGLISH ve TURKCE sembollerinin durumuna göre metotların çağrılıp çağrılmayacağı belirleniyor.

```
#define TURKCE
using System;
using System.Diagnostics;

class Deneme
{
      [Conditional("TURKCE")]
      public void Turkce()
      {
          Console.WriteLine("Merhaba!");
      }

      [Conditional("ENGLISH")]
      public void English()
      {
          Console.WriteLine("Hello!");
      }
}
```

```
class Nitelikler
{
      static void Main()
      {
          Deneme den = new Deneme();

          den.English();
          den.Turkce();
      }
}
```

"den" nesnesi üzerinden English() metodu çağrılmasına rağmen programı çalıştırdığımızda ekrana sadece "Merhaba!" yazdığını görüyoruz. Bunun sebebi English() metodunun ancak ENGLISH sembolü tanımlanmışsa çağrılacağıdır. Diğer bir ilginç nokta ise metot çağrımının hata vermemesidir. Koşulu sağlamayan metot çağrılmak istenildiğinde metot çağrılmaz ve hiçbir hata da alınmaz. Conditional niteliğinin kullanımı #if önişlemci komutunun koşullu derleme yapmasına çok benzemektedir.

Yukarıdaki programda eğer

```
#define ENGLISH
```

komutunu da eklersek bu sefer hem Turkce() hem de English() metodu çağrılacaktır.

DllImport niteliği: Bu nitelik ile bir metodun .NET ortamında geliştirilmediği belirtilmektedir. Özellikle Windows API fonksiyonlarını kullanmak için DllImport niteliği kullanılır. Bu nitelik System.Runtime.InteropServices isim alanında bulunur. Bu niteliğin kullanımına bir örnek verelim.

Bir Windows API fonksiyonu olan MessageBox() fonksiyonunu C# ile yazılmış programımızda kullanabilmek için fonksiyonu DllImport niteliği ile bildirmeliyiz. Ayrıca fonksiyonun harici bir kaynakta olduğunu bildirmek için ise **extern** anahtar sözcüğü kullanılmalıdır. Windows API'daki MessageBox fonksiyonunun prototipi aşağıdaki gibidir.

```
int MessageBox(int aile, string mesaj, string baslik, int tip)
```

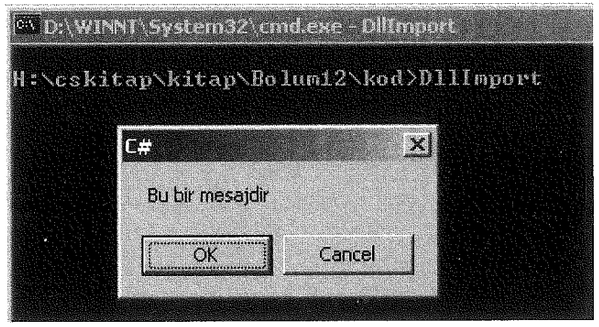
MessageBox fonksiyonu ile Windows işletim sisteminde sıkça gördüğümüz uyarı penceresi gösterilir.

```
using System;
using System.Runtime.InteropServices;

class Nitelikler
{
      [DllImport("User32.dll")]
      public static extern int MessageBox(int aile,string
```



```
                                  mesaj,string baslik,int tip);

      static void Main()
      {
          MessageBox(0,"Bu bir mesajdır","C#",1);
      }
}
```

Programı çalıştırdığımızda aşağıdaki gibi bir mesaj kutusu gösterilecektir.



MessageBox() metodunun tip parametresini değiştirerek değişik mesaj kutularını görebilirsiniz.

Obsolete niteliği: Bu nitelik ile bir metodun çağrılması durumunda hata ya da uyarı verilmesi sağlanır. Tıpkı önişlemci komutları olan #error ve #warning gibi. Obsolete niteliğinin iki kullanım biçimi vardır. Bunlar aşağıdaki gibidir.

```
[Obsolete("Uyarı mesajı")]
```

ve

```
[Obsolete("Uyarı mesajı", bool hata)]
```

Birinci kullanımda eğer Obsolete ile işaretlenmiş metot çağrılırsa derleyici uyarı mesajı verecektir. Hatırlarsanız derleyicinin uyarı vermesi programın derlenmesine engel olmuyordu. İkinci kullanımda ise eğer bool türünden olan hata true olarak girilirse ilgili metot çağrıldığında derleyici hata mesajı verecektir. Dolayısıyla program derlenemeyecektir. Aşağıdaki programda Obsolete niteliğinin kullanımına örnek verilmiştir.

```
using System;

class Nitelikler
{
      [Obsolete("Uyarı: Bu metot eski versiyondur.")]
```

```
        public static void Metot1()
        {
            Console.WriteLine("Uyarı veren metot");
        }

        [Obsolete("Hata: Bu metot artık kullanılamaz",true)]
        public static void Metot2()
        {
            Console.WriteLine("Uyarı veren metot");
        }

        static void Main()
        {
            //Derleyici Uyarı verecektir.
            Metot1();

            //Derleyici hata verecektir.
            Metot2();
        }
}
```

Obsolete niteliği System isim alanında bulunur.

`AttributeUsage` niteliği: Bu nitelik ile birazdan göreceğimiz kullanıcı tanımlı nitelik sınıflarının hangi elemanlar için kullanılabileceğini belirtir. Parametre olarak AttributeTargets isimli bir numaralandırma almaktadır. Bu numaralandırma ile programcının tanımladığı niteliklerin hangi elemanlara uygulanabileceği belirtilir. Aşağıda AttributeTargets numaralandırmasında bulunan semboller listelenmiştir.

All, Assembly, Class, Constructor, Delegate, Enum, Event, Field, Interface, Method, Module, Parameter, Property, ReturnValue, Struct

Bu niteliğin kullanımına bir örnek verecek olursak;

```
[AttributeUsage(AttributeTargets.Method)]
```

ifadesi, ilgili nitelik sadece metotlara uygulanabilir anlamına gelir.

Özel Nitelik Bildirimi ve Niteliklerin Elde Edilmesi

C#'taki niteliklerin tamamı aslında System.Attribute sınıfını temsil etmektedir. O halde Attribute sınıfından kalıtım yolu ile kendi niteliklerimizi oluşturabiliriz. Niteliklerin oluşturulması normal bir sınıf bildiriminden farksızdır. Örneğin aşağıda OzelNitelik adlı bir nitelik sınıfı bildirilip ardından bir metot bu nitelik ile işaretleniyor.


```
using System;
[AttributeUsage(AttributeTargets.Method)]
class OzelNitelik : Attribute
{
        private string mMesaj;

        public OzelNitelik(string mesaj)
        {
            mMesaj = mesaj;
        }

        public string Mesaj
        {
            get
            {
                return mMesaj;
            }
        }
}

class Nitelikler
{
        [OzelNitelik("Ozel nitelikli metot")]
        public static void Metot2()
        {
            Console.WriteLine("Metot2");
        }

        static void Main()
        {

        }
}
```

Bu programda bildirilen OzelNitelik isimli nitelik sınıfının bildirimini görüyorsunuz.

```
[AttributeUsage(AttributeTargets.Method)]
```

nitelik bildirimi ile OzelNitelik niteliğinin yalnızca metotlara uygulanabileceği belirtiliyor. Örneğin OzelNitelik niteliğini aşağıdaki gibi bir sınıf bildirimi için uygulayamayız.

```
[OzelNitelik("Nitelikli sınıf")]
class SınıfBildirimi
{
}
```

Niteliklerle ilgili diğer bir nokta niteliklerde isimli parametrelerin de tanımlanabilmesidir. Örneğin aşağıdaki gibi kullanılan bir niteliğe ilişkin sınıf bildirimi yapılabilir.

```
[OzelNitelik("Nitelikli sınıf",konu= "C#", no=5.6)]
```

Buradaki konu ve no elemanlarına isimli parametre denilmektedir. Bu isimli parametreler bir nitelik sınıfındaki public üye değişkenleri temsil etmektedir. Aşağıdaki nitelik sınıfı bildiriminde bunun nasıl olduğunu görebilirsiniz.

```
using System;

[AttributeUsage(AttributeTargets.Method)]
class OzelNitelik : Attribute
{
      private string mMesaj;
      public int No;
      public string Konu;

      public OzelNitelik(string mesaj)
      {
          mMesaj = mesaj;
      }

      public string Mesaj
      {
          get
          {
              return mMesaj;
          }
      }
}

class Nitelikler
{
      [OzelNitelik("Ozel nitelikli metot",No=5,Konu="C#")]
      public static void Metot2()
      {
          Console.WriteLine("Uyarı veren metot");
      }

      static void Main()
      {

      }
}
```

Bir metot OzelNitelik ile aşağıdaki değişik şekillerde bildirilebilir.

```
[OzelNitelik("Ozel nitelikli metot",No=5,Konu="C#")]
[OzelNitelik("Ozel nitelikli metot",No=5)]
[OzelNitelik("Ozel nitelikli metot", Konu="C#")]
[OzelNitelik("Ozel nitelikli metot")]
```

OzelNitelik sınıfındaki int türünden olan No ve string türünden olan Konu değişkenleri nitelik sınıfının isimli parametrelerini oluşturur.

Bir elemana ilişkin niteliklerin yansıma yolu ile elde etmek için Type sınıfının GetAttributes() metodu ya da Attribute sınıfının statik GetCustomAttributes() metodu kullanılır. Bu metotlarla ilgili ayrıntılı bilgi verilmeyecektir. Attribute sınıfını ve daha önce anlattığımız Type sınıfının elemanlarını MSDN kütüphanesinden inceleyerek bu konudaki bilgilerinizi pekiştirebilirsiniz.

Bu bölümün sonuna gelmiş bulunuyoruz. Bu bölümde C#'taki birçok önemli konuyu inceledik. Her bir konu ile ilgili değişik sayıda örnekler verdik. Bu örneklerin yanısıra sizin kendiniz yapmanız için birtakım program önerilerinde bulunuldu. Bu bölümde anlatılan konular ile ilk defa karşılaşıyorsanız verilen örnekleri bir defa daha incelemeniz konuları daha iyi anlamanız açısından önemlidir.

Plugin (Eklenti) Tabanlı Programlama ve Yansıma

Yansıma, profesyonel uygulamalarda oldukça fazla kullanılan bir kütüphanedir. En çok kullanıldığı alanlardan bir tanesi **eklenti** (*plugin*) tabanlı programlardır. Eklentiler, varolan bir uygulamanın ana yapısına hiç müdahale etmeden programa ekstra özellikler katan yapılara denilmektedir. Örneğin Adobe firmasının Photoshop uygulamasını düşünelim. Photoshop programı kurulduktan sonra üçüncü parti firmalar tarafından sağlanan birçok plugin (örneğin resim üzerinde çeşitli efektler veren eklentiler) ana uygulamayı değiştirmeden kurulabilmektedir. Belirli şartlar sağlandığı anda Photoshop uygulaması, ilgili plugini barındıran kütüphaneyi yükleyerek kendine entegre eder. Böylece birçok firmanın varolan bir programa yeni özellikler katabilmesi sağlanır. .NET dünyasında eklenti tabanlı programlar, **Reflection** kütüphaneleri kullanılmaktadır.

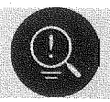
Biz basit bir örnek ile eklenti tabanlı bir mimari oluşturmaya çalışacağız. Örneğimiz tabii ki Console üzerinden olacak ancak genel mimariyi anladığınızda örneği çok farklı ortamlarda çalıştırmak için fazla efor sarfetmeniz gerekmeyecek.

Örneğimizde iki tam sayı üzerinde matametiksel işlemler yapan bir uygulama yapacağız. Amacımız *exe* olan uygulamamız dağıtıldıktan sonra bizim sağladığımız şartları sağlayan herhangi bir bileşenin de programımız tarafından dinamik olarak yüklenebilmesini ve çalıştırılabilmesini sağlamaktır. Bu amaçla Hesaplama işlemi yapacak plugin'lerin bizim istediğimiz şartları sağlaması için bir **arayüz** (*interface*) hazırlayacağız. Bu arayüz bir *dll* olacak ve uygulamamız için plugin geliştirecek herkesin bu arayüzü içeren dll'i elde etmesi gerekecek. Zaten birçok uygulamadan programcılar için bir **SDK** (*Software Development Kit*) yayınlanır ve bu SDK içinde çeşitli arayüzler olur. Biz de buna benzer bir yaklaşımı benimsiyoruz.

Buna göre arayüzümüz aşağıdaki gibi olacaktır. İşlem yapan metot ve eklentinin adını temsil eden **Isim** isimli özellik içermektedir.

```
//Islem.dll

using System;

namespace Islem
{
    public interface IHesap
    {
        int Hesapla(int a, int b);
        string Isim { get; }
    }
}
```

Bu sınıfı bir dll şeklinde *C:\Plugins* isimli dizinde oluşturun.



Komut satırından dll oluşturmak için /t:library parametresi kullanılır. VS.NET 2005 ile çalışıyorsanız Class Library projesi açıp Build etmeniz yeterlidir.

Ardından uygulamamızın ilk olarak 2 tane eklentisi olacak. Bunlar Çarpma ve Toplama işlemi yapan iki dll olacaktır. Aşağıdaki iki sınıfı da iki farklı Dll olarak derleyip *C:\Plugins* dizinine kopyalayın.

//Toplama.dll

```
using System;
using Islem;

namespace Toplama
{
    public class Toplam:IHesap
    {
        public int Hesapla(int a, int b)
        {
            return a + b;
        }

        public string Isim
        {
            get
            {
                return "Toplam";
            }
        }
    }
}
```


//Carpma.dll

```
using System;
using Islem;

namespace Carpma
{
    public class Carpma:IHesap
    {
        public int Hesapla(int a, int b)
        {
            return a * b;
        }

        public string Isim
        {
            get {
                return "Carpma";
            }
        }
    }
}
```

Yukarıdaki sınıfı dll olarak derlerken *Islem.dll*'e referans vermeyi unutmayın. Komut satırı derleyicisi ile referans verme işi */r:* parametresi ile yapıldığını hatırlayalım.

csc /r:Islem.dll /t:library Toplama.cs

Sıra eklentilerimizi yönetecek ana uygulamamızı yazmaya geldi. Bu uygulamada bütün eklentiler Main metodu içinde reflection kütüphanesi ile yüklenecek ve kullanıcıya hangi eklentiyi çalıştırmak istendiği sorulmaktadır. Seçilen işlem eklentiler içinden aranarak sonucu kullanıcıya gösterilmektedir.

Örnek uygulamamızın kodları aşağıdaki gibidir.

//HesaplamaProgrami.exe

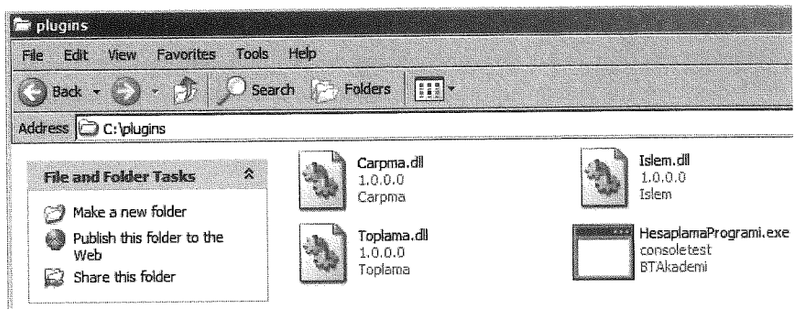
```
using System;
using System.Reflection;
using System.Collections;
using System.IO;
using Islem;

namespace consoletest
{
  class Program
  {
      public static object[] pluginler = new object[100];
```

```
        static void Main(string[] args)
        {
            PluginYukle("c:\\plugins");

            string key = "";
            do
            {
                Console.WriteLine("Birinci sayıyıyı giriniz:");
                int sayi1 = Convert.ToInt32(Console.ReadLine());

               Console.WriteLine("İkinci sayıyıyı giriniz:");
                int sayi2 = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Yapmak istediğiniz işlemi  giriniz.:");
                key = Console.ReadLine();

                if (key != "q")
                {
                    IHesap plugin = PluginAra(key);
                    if (plugin != null)
                    {
                        int sonuc = plugin.Hesapla(sayi1, sayi2);
                        Console.WriteLine("Sonuç: " + sonuc.
ToString());
                    }
                    else
                    {
Console.WriteLine("Yüklenmeyen bir işlem seçildi.");
                    }
                }

            } while (key != "q");
        }

        static IHesap PluginAra(string isim)
        {
            foreach (IHesap plugin in pluginler)
            {
                if (plugin.Isim == isim)
                    return plugin;
            }

            return null;
        }

        static void PluginYukle(string dizin)
        {
            string[] dosyalar = Directory.GetFiles(dizin, "*.dll");
            int index = 0;
            foreach (string dosya in dosyalar)
            {
                Assembly asm = Assembly.LoadFrom(dosya);

                Type[] typelar = asm.GetTypes();

                foreach (Type type in typelar)
```

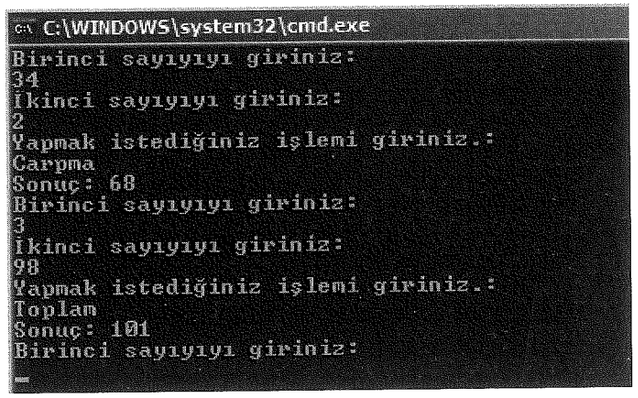


```
                {
            if (type.IsClass &&    typeof(IHesap).IsAssignableFrom(type))
                    {
object nesne = Activator.CreateInstance(type);
                        pluginler[index] = nesne;
                        index++;
                    }
                }
            }
        }
    }

}
```

Yukarıdaki programı derlerken dikkat edilmesi gereken nokta sadece **Islem.dll**'e referans verilmiş olmasına rağmen **Toplama.dll** ve **Carpma.dll** içerisindeki metotların proje içinden çağrılabilmesidir.



Bütün derleme işlemlerini başarılı bir şekilde yaptığınızda C:\\plugins dizini aşağıdaki görünmelidir.



Böylece herhangi bir programcı **Islem.dll**'i alarak bizim programa yeni eklentiler geliştirebilir. Programımızın yeni eklentiye adapte olması için gereken tek şey ilgili dll

C:\plugins dizinine kopyalanması olacaktır.

C# Dilinin Dinamik Özelliği (dynamic ve DLR)

Bilindiği üzere C, C++, C# ve Java gibi programlama dilleri statik programlama dilleri olarak anılmaktadır. Statik dil ile kastedilen ise genellikle tip tanımlarının ve tipler üzerinde bulunan üye elemanların (metot, olay, property vs) çağrım ifadelerinin derleme zamanında belirgin olarak bilinmesi zorunluluğudur. Örneğin C# dilinde bildirimini yapmadığınız bir metodun çağrım ifadesi derleme zamanında hata verecektir. Ya da derleme aşamasında bildirimi yapılmayan bir özellik çağrım ifadesi henüz derleme aşamasında hata verecektir ve kod derlenmeyecektir. Ancak Perl, Javascript, Rubby, Python gibi programlama dilleri ise dinamik dillerdir. Burada dinamiklikten kasıt derleme işlemi sırasında kontrolü yapılan tip uyumluluğu veya üye çağrım ifadelerinin geçerlilik kontrolünün çalışma zamanında yapılmasıdır. Dolayısıyla statik diller çoğunlukla dinamik dillere göre performans olarak daha öndedir. Ancak kodlama açısından baktığınızda dinamik dillerin sağlamış olduğu birçok kolaylık bulunmaktadır. Örneğin Javascript dilinde bir türe ait herhangi bir özellik bildirimini yapmadan türe ait özellik direkt olarak kullanılabilmektedir. Genelde script dili diye tanımlanan diller dinamik diller kategorisinde bulunumaktadır.

2010 yılında final sürümü yayınlanan C# 4.0 standartlarında ve .NET 4.0 çalışma ortamında artık dinamik dillerin desteklediği bir takım özellikler performans kaybı olmadan desteklenebilmektedir. Hatta .NET ortamında çalışan **Ruby** (*IronRuby*) ve **Python** (*IronPython*) dillerinin gerçek platformlarına göre daha performanslı çalıştıkları bile yapılan testlerde gözlemlenmiştir. Dinamik diller için eleştiri konusu olan performans problemi .NET platformunda **DLR** (*Dynamic Language Runtime*) dediğimiz ve CLR üzerinde çalışan yeni bir katman sayesinde kısmen de olsa çözülmüştür. DLR konusuna bu bölümün sonunda ayrıca değinilecektir.

Bu bölümde genel olarak C# dilinin dinamik özelliklerinden, kısaca bu işin DLR ile nasıl çözüldüğünden ve kendi dinamik nesnelerimizi ne şekilde geliştirebileceğimizden bahsedilecektir.

Önemli Not: C# diline dinamik özellikler C# 4.0 (2010 yılında) ile birlikte girmiştir.

Statik ve Dinamik Dillerin Karşılaştırılması

Esasında **Statik** ve **Dinamik** kelimeleri ifade anlamında tam olarak yukarıda bahsedilen ayrımları yansıtmamaktadır. C türevi diller, Pascal ve VB gibi dillerin derleme aşamasında tip güvenliğini önde tutan tutumu dolayısıyla statik dil olarak adlandırılıyor. Dinamik diller ise bu davranışı biraz öteleyip çalışma zamanına bırakmaktadır. Örneğin aşağıdaki gibi bir örnek kod (benzerlerinin diğer dillerde yazıldığını varsayıyoruz) herhangi bir statik dilde derleme hatası verecektir.


```
object o = new object();
o.Ad = "Sefer";
```

Ancak dinamik dil olarak kabul edebileceğimiz Javascript'de (ya da benzerlerinde) aşağıdakine benzer bir kod derleme aşamasında hata vermeyecektir. Hatta bazı dillerde (script dilleri) derleme diye bir kavram da olmayabilir.

```
var o = new object();
o.Ad = "Sefer";
```

Derleme aşamasında hata vermemesinin tek açıklaması statik dillerin derleme zamanında yaptığı işlemin çalışma zamanında yapılmasıdır. Bu yüzden dinamik diller genel olarak yavaşlık problemi ile karşı karşıya kalırlar. Ancak bu yavaşlık problemi gözardı edildiğinde programcıya oldukça kolaylık sağlanmış olunur ve birçok tür dönüşüm işlemi de çalışma zamanında yapılarak programcının daha az kod yazması sağlanmış olunur. Elbette dinamik olarak tanımlanan tiplerin çalışma zamanında ne şekilde davranacağı belli olduğu için birçok hatayı da ancak çalışma zamanında yakalanabiliyor. Bu da yazılan kodun güvenilirliğinin dinamik dillerde daha zor sağlanabilmesine neden oluyor. Aynı şekilde dinamik dillerde kod yazım aşamasından hemen sonraki derleme aşamasında tip dönüşümleri ve üye çağrım ifadelerinin davranışları belirgin olmadığından birçok geliştiricinin gözbebeği olan refactoring, intellisense gibi özellikler statik dillere göre çok daha kısıtlı yapılabilmektedir.

Her şeye rağmen dinamik dillerin .NET ile tam bütünleşik çalışabilmesinin sağlanması ve birlikte çalışabilirliğin daha kuvvetli olması için Microsoft, C# 4.0' dan itibaren bir takım dinamik özellikleri C# diline ve .NET platformuna eklenmesini sağlamıştır. Burada Microsoft'un üzerinde durduğu en önemli noktalardan biri dinamiklik ile birlikte gelen performans probleminin nasıl çözüleceğidir. Bu amaçla dinamiklik (dynamic anahtar sözcüğü ile tanımlanan değişkenler) özelliği geliştirilirken mevcut CLR'nin ana çalışma mimarisine dokunmadan bu özellikleri başka bir katman olan DLR ile çözümlemeye karar vermiştir. DLR klasik CLR üzerinde çalışan ve dinamik nesnelerin ne şekilde davranacağına çalışma zamanında karar veren bir katmandır. Dinamik nesnenin yapısı analiz edilip üye çağrımlarında ne şekilde bir işlem gerçekleştirileceğine karar verildikten sonra CRL ilgili işlemin icrası için devreye girer. Ancak DLR ilgili çağrım ifadesini bir önbelleğe alarak sonraki çağrım ifadelerinde DLR'in getirmiş olduğu yavaşlığın önüne geçer. Bu sayede dinamik tiplerde dahi standart CLR çağrımlarının performansına oldukça yakın bir sonuç elde edilir.

Bu kitap kapsamında DLR'in çalışma mekanizması ayrıntılı olarak ele alınmayacak olup DLR'in C# dili ile olan entegrasyonunu sağlayan dynamic anahtar sözcüğü DynamicObject isimli sınıftan bahsedilecektir.

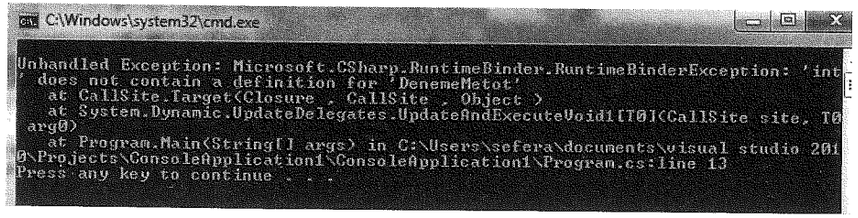
dynamic ile Tanımlanan Tipler

C# 4.0'dan itibaren dinamik tiplerin tanımlanması "dynamic" isimli bir anahtar sözcük ile yapılmaktadır. Herhangi bir tip dynamic olarak tanımlandığında artık o tip üzerinden çağrılacak herhangi bir üye çağrımı (metot, özellik çağrımı gibi) derleme aşamasında kontrol edilmemekte ve bir denetimden geçmemekte. Örneğin aşağıdaki kod bloğu C# derleyicisi açısından geçerli bir koddur.

```
dynamic a = 7;
a.DenemeMetot();
a.DenemeOzellik = 5;
```

Yukarıdaki kod bloğunda dinamik olarak bildirilen int tipli a değişkeni üzerinden çağrılacak herhangi bir metot ya da özelliğin tip güvenliği açısından kontrolü ya da geçerlilik kontrolü C# derleyicisi tarafından yapılmayacaktır. Bu işlem çalışma zamanında DLR tarafından yapılacaktır.

Yukarıdaki kod bloğunu içeren bir uygulamayı derleyip çalıştırdığınızda aşağıdaki gibi hata aldığınızı göreceksiniz.



Bu hatada doğal olarak "Dinamik" olarak tanımlansa dahi int tipindeki bir değişkende DenemeMetot isimli bir metodun olmadığı söylenmektedir. Bu hata DLR seviyesinde bir hatadır. Bu hatanın temel çıkış noktası DLR'nin dinamik olarak tanımlanan int tipinden veriler için üye çağrım ifadelerinin ne şekilde ele alacağını bilememesinden kaynaklanmaktadır.

Aynı şekilde aşağıdaki gibi kendi sınıfımızı yazıp bu sınıf tipinden bir dinamik nesne tanımladığımızda olmayan bir özelliği çağırdığımızda çalışma zamanında yukarıdaki hataya benzer bir hata alırız.

```
public class MyObject
{
}

//main metodunun içi
dynamic di = new MyObject();
di.Ad = "Ali";
```

Sonuç olarak klasik bir CLR tipinin dynamic olarak tanımlanması o nesne üzerinde

dilediğimiz özelliğin veya metodun çağrılabileceği anlamına gelmez. Ancak DLR bu özgürlüğü bize `IDynamicMetaObjectProvider` isimli interface ile verir. DLR bir nesnenin dinamik olarak metot ya da diğer üye çağrımlarını içerebilmesini sağlamak ve çalışma zamanındaki davranışını belirlemek için bu arayüzü kullanır. Dolayısıyla dinamik olarak tanımlanan bir değişken üzerinden yapılan herhangi bir çağrım ifadesi ile karşılaşıldığında C# derleyicisi bir kontrol yapmaz ve ifadeyi çalışma zamanında DLR'ye iletmek üzere işaretler. Çağrım bloğunun çalıştırma emrini alan DLR öncelikli olarak ilgili tipin `IDynamicMetaObjectProvider` arayüzünden türeyip türemediğini kontrol eder. Eğer nesne bu arayüzden türemişse direkt olarak bu arayüzün içinde bulunan GetMetaObject isimli metot çağrılır. Bu metot içinde ise zaten ilgili sınıfın dinamik tanımlandığı durumda üye çağrım ifadesinin ne şekilde davranacağı bildirilmiş oluyor. Eğer `IDynamicMetaObjectProvider` arayüzünden türetilmemişse bu durumda DLR standart olarak CLR Binder mekanizmasını devreye sokarak ilgili tipin standart reflection çağrım ifadesini otomatik olarak yapar. Yani InvokeMethod, InvokeProperty gibi reflection konusundan aşina olduğumuz metot çağrımları otomatik olarak yapılır.

`IDynamicMetaObjectProvider` arayüzünü uygulamayan bütün tiplerde dynamic anahtar sözcüğü ile normalde reflection ile yaptığımız çağrım ifadelerini artık rutin bir CLR çağrımı gibi yapabilmekteyiz. Örneğin Activator.CreateInstance ile yaratılan bir nesne eğer dinamik olarak çağrılırsa ilgili değişkenin bütün özellik ve metotlarına "." Operatörü ile rahatlıkla erişebiliriz.

Örnek: C# Reflection Binder

```
using System;

class Program
{
   static void Main(string[] args)
   {
     dynamic d = Activator.CreateInstance(typeof(String),'C',5);

     int uzunluk = d.Length; // CreateInstance metodu
geriye object dönmesine rağmen "." Operatörü ile Length
özelliğine eriştik.

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

Yukarıdaki program derlenip çalıştırıldığında ekrana 5 değerinin yazıldığı görülecektir. Halbuki VS.NET 2010 editörü ile "d." İfadesini yazdığımızda aşağıdaki gibi bir uyarı çıktığı görülür.

```
dynamic d = Activator.CreateInstance(typeof(String),'C',5);

int uzunluk = d.

Console.WriteLin
```

(dynamic expression)
This operation will be resolved at runtime.

Bu uyarıda "d değişkeni üzerinden nokta operatörü ile ulaşılmak istenen üye'nin çözülmesi çalışma zamanında yapılacaktır" denmektedir.

Yukarıdaki örnekten de anlaşılacağı üzere eğer dinamik olarak tanımlanan değişken tipi `IDynamicMetaObjectProvider` arayüzünden türetilmemişse DLR'nin varsayılan davranışı C# CLR Binder mekanizmasını devreye sokarak "reflection" sınıfları ile ilgili çağrımı standart CLR kuralları çerçevesinde çalışmasını sağlamaktır. Bu mekanizmaya Member Binding denir. .NET 4.0 içerisinde genel itibariyle 4 adet "Binder" hazır olarak gelmektedir.

Bunlar:

• **CLR Binder**: Yukarıda bahsedilen reflection mekanizmasının çalışmasını sağlar. Yani dinamik tip eğer `IDynamicMetaObjectProvider` arayüzünden türetilmemişse bu Binder devreye girer.

• **IronPython Binder:** Klasik .NET nesneleri ile dinamik bir dil olan Python'un .NET uyarlaması olan IronPython ile entegre çalışabilmek için geliştirilmiş bir Binder mekanizmasıdır.

• **IronRuby Binder:** Klasik .NET nesneleri ile dinamik bir dil olan Ruby'nin .NET uyarlaması olan IronRuby ile entegre çalışabilmek için geliştirilmiş bir Binder mekanizmasıdır.

• **COM Binder:** .NET programcılarının COM (eski nesil bileşen teknolojisi) nesnelerine daha kolay ve hızlı erişebilmesi için arka planda `IDispatch` arayüzünü kullanarak erişim yapan bir Binder mekanizmasıdır. COM Binder sayesinde COM **Wrapper** sınıfları oldukça kısalmakta ve programcıların gereksiz yere tür dönüşümü yapmaları ortadan kalkmaktadır. COM nesnelerine erişim bu kitabın kapsamında olmadığı için bu konuya değinilmeyecektir.

Bütün bu Binder yapılarının ortak özelliği `IDynamicMetaObjectProvider` arayüzünü kullanmalarıdır. Dolayısıyla programcılarda bu arayüzü kullanarak kendi Binder mekanizmalarını rahatlıkla oluşturabilmektedir.

.NET'in DLR katmanı genel olarak .NET dillerinin dinamik dillerle daha hızlı ve kolay entegre edilmesini sağlamak amacıyla geliştirilmiştir. Aynı zamanda statik dillerin sağlamış olduğu hızlı çalışma zamanı ortamını da koruyabilmek amacıyla CLR üzerinde radikal değişiklikler yapılmamış ve DLR üzerinde "CallSite Caching" isimli bir mekanizma ile çağrım ifadeleri ön belleğe alınarak aynı çağrım tekrar edildiğinde derleme aşamasında çözümleme yapılmış gibi bir hız kazandırılmaktadır.

dynamic Değişkenler ile İlgili Bilinmesi Gerekenler

dynamic anahtar sözcüğü kullanılarak tanımlanan değişkenler ile ilgili kısıtlar ve kullanım senaryoları aşağıda listelenmiştir;

1. Dinamik değişkenler metotlara parametre olarak gönderilebilir.

2. Dinamik değişkenler metotların geri dönüş değeri olabilir.

3. Dinamik tipler ile statik tipler arasında rahatlıkla tür dönüşümü yapılabilir. Tür dönüşümünün geçerliliği derleme aşamasında değil çalışma zamanında kontrol edilir. Geçerli değilse çalışma zamanında exception oluşturulur. Örneğin aşağıdaki kod derlenecektir ancak çalışma zamanında exception alınacaktır.

```
dynamic d = "ali";
int a = d;
```

CLR Binder devreye girerek normalde derleme aşamasında alınacak hata çalışma zamanına ertelenmiş olacaktır.

4. Çalışma zamanında geçerli olan tür dönüştürme işlemleri de açıkça belirtilmeden yapılabilir. Örneğin aşağıdaki kod hem derleme aşamasında hem de çalışma zamanında hatasız çalışacaktır.

```
dynamic d = 45;
decimal k = d;
```

5. Visual Studio.NET 2010 içerisinde dinamik tanımlı değişkenler üzerinde maalesef **Intellisense** (*kod tamamlama*) özelliği çalışmamaktadır. (VS.NET 2010 RC 1 sürümünde)

6. Eğer nesne yaratılacağı zaman tipe direkt erişilebiliyorsa dinamik olarak tanımlamanın programcıya bir faydası yoktur, aksine çalışma zamanında hata yapma olasılığı artacağı gibi **reflection binder** mekanizması kullanılacağı için ciddi bir performans problemi yaşanabilir.

7. Halihazırda **Reflection** kütüphanelerinin zaten kullanıldığı projelerinizde nesnelerinizi dinamik olarak tanımlayarak hızlı erişim yapabilirsiniz.

8. .NET Framework içinde bulunan Binder mekanizmaları gibi kendi Binding meka-

nizmamızı oluşturabiliriz.

9. Dinamik çağrım ifadelerinde **Extension Metotları** (*genişletme metotları*) bulunamaz.

10. Dinamik metot çağrım ifadelerinde lambda ifadeleri parametre olarak geçirilemez. (Lambda ifadeleri için LINQ konusuna bakınız.)

İpucu 1: dinamik değişkenler C# derleyicisinin bir kısım görevlerinin çalışma zamanına ertelenmesini sağlar.

İpucu 2: DLR, CLR üzerinde çalışan bir katmandır. Call Site Caching özelliği ile çağrım ifadelerini eski CLR'nin anlayacağı hale getirip saklar. Böylece dinamik özelliği neredeyse statik dil hızı ile çalışır.

DynamicObject Sınıfı ve IDynamicMetaObjectProvider Arayüzü

Önceki kısımlarda anlatıldığı üzere DLR bir dinamik tip üzerindeki çağrımın ne şekilde ele alınacağını `IDynamicMetaObjectProvider` ile belirlenen kurallar ile anlar. Bu arayüz içinde prototipi aşağıda verilen sadece bir metot bulunmaktadır.

```
DynamicMetaObject GetMetaObject(Expression parameter)
```

Normal şartlarda bu metot uygulandığında DLR otomatik olarak bu metodu çağırır. Geriye döndürülecek `DynamicMetaObject` ile işlemin nasıl ele alınacağı belirlenir. Programcıların kendi dinamik nesnelerini daha hızlı yapabilmeleri için belli başlı çağrım ifadelerinin hazır olarak implemente edildiği ve bunları override edebileceğimiz yine aynı şekilde `IDynamicMetaObjectProvider` arayüzünden türeyen System.Dynamic isim alanında `DynamicObject` isimli abstract bir sınıf bulunmaktadır. Bu sınıf kullanılarak çok hızlı bir şekilde kendimize özgü dinamik sınıflar geliştirebiliriz.

Örneğin basit bir şekilde property atama ve set etme işlemlerini dinamik hale getiren ancak diğer çağrımları desteklemeyen bir Entity sınıfı aşağıdaki gibi geliştirilebilir.

```
public class Entity : System.Dynamic.DynamicObject
{

    private Dictionary<string, object> ozellikler =
new Dictionary<string, object>();

    public override bool TrySetMember(SetMemberBinder binder,
object value)
    {
        if (!ozellikler.Keys.Contains(binder.Name))
            ozellikler.Add(binder.Name, value);
```



```
        else
            ozellikler[ binder.Name] = value;
        return true;
    }

    public override bool TryGetMember(GetMemberBinder binder, out
object result)
    {
        if (!ozellikler.Keys.Contains(binder.Name))
            throw new Exception("Özellik bulunamadı");

        result = ozellikler[ binder.Name] ;

        return true;
    }

    public override DynamicMetaObject
GetMetaObject(System.Linq.Expressions.Expression parameter)
    {
        return base.GetMetaObject(parameter);
    }
}
```

Yukarıdaki örnekte dinamik olarak belirlenen Property'ler bir dictionary koleksiyonunda tutularak `TryGetMember` ve `TrySetMember` fonksiyonlarında kullanılmaktadır. Entity nesnesi türünden bir değişken artık dinamik anahtar sözcüğü ile tanımlandığında herhangi Property erişiminde TryGetMember, herhangi bir Property'ye bir atama olduğunda TrySetMember metotları çalışacaktır.

Bu durumda aşağıdaki gibi kod ifadeleri tamamen geçerlidir;

```
dynamic de = new Entity();

de.Ad = "Sefer";
de.Soyad = "Algan";
de.No = 1;
Console.WriteLine(di.Ad);
```

Gerçekte Entity nesnesinde Ad, Soyad ve No gibi özellikler olmamasına rağmen Entity sınıfında biz ilgili atamalar yapıldığında bir listede bu atamaya ilişkin değerleri tutup gerektiğinde geri veriyoruz.

C# çoklu sınıf türetmesini (multiple inheritance) desteklemediği için kendi dinamik nesnelerinizi DynamicObject'den türettiğiniz zaman aynı anda başka taban nesneniz olamayacaktır.

Yukarıdaki örnekte DynamicObject sınıfında bulunan sadece TryGetMember ve TrySetMember metotlarını yeniden tanımladık. DynamicObject içinde diğer farklı çağrım ifadeleri için de ayrı ayrı metotlar bulunmakta. Bütün bu metotları override ederek bütün C# çağrım ifadelerini çalışma zamanına öteleyebilirsiniz.

DynamicObject içerisinde bulunan diğer metotlar:

Yukarıdaki bütün metotlar virtual tanımlıdır ve geri dönüş değerleri bool türündendir. Bu metotlar geriye işlemin başarılı olup olmadığı ile ilgili bilgi dönmelidir. Her bir metodun ilk parametresi ise ilgili işlem ile ilgili ayrıntılı bilgileri verir.

1. `TryBinaryOperation(BinaryOperationBinder binder, object arg, out object result)`

İki dinamik nesnenin bir binary operasyon içinde ne şekilde davranacağını belirler. Örneğin: `toplam = sayi1 + sayi2` ifadesinde sayi1 ve sayi2'nin dinamik nesne olduğu durumda işlemin ne şekilde gerçekleşeceğini belirler.

2. `TryConvert(ConvertBinder binder, out object result);`

Dinamik nesnelerin tür dönüşümlerinin ne şekilde yapılacağını belirler. Örneğin `(int)dinamiktip;`

3. `TryCreateInstance(CreateInstanceBinder binder, object[] args, out object result);`

Dinamik bir nesnenin ilk yaratılması anının ne şekilde olacağını belirler.

Örnek: `dynamic d = new DinamikSinif();`

4. `TryDeleteIndex(DeleteIndexBinder binder, object[] indexes);`

Python ve Ruby dillerinde bulunan index ile bir nesneyi null değere çekmek için kullanılan delete anahtar sözcüğü için geliştirilmiştir. C# dilinde karşılığı olmadığı için override etmek bir anlama gelmez.

5. `TryDeleteMember(DeleteMemberBinder binder);`

C# dili için anlam ifade etmez. Yine aynı şekilde Ruby ve Python için geliştirilmiştir.

6. `TryGetIndex(GetIndexBinder binder, object[] indexes, out object result);`

Indeksleyici erişimlerinin ne şekilde olacağını belirler.

Örnek: `int a = dinamikKoleksiyon[ 3] ;`

7. `TrySetIndex(SetIndexBinder binder, object[] indexes, object value);`

Indeksleyici atamalarının ne şekilde olacağını belirler.

Örnek: `dinamikKoleksiyon[ 3] = 10;`

8. `TryInvoke(InvokeBinder binder, object[] args, out object result);`

Temsilci çağrımı ve değişken üzerinde bir nesneye erişimin ne şekilde ele alınacağını belirler. Örnek : dinamikDegisken(10);

9. `TryInvokeMember(InvokeMemberBinder binder, object[] args, out object result);`

Metot çağrımlarının ne şekilde ele alınacağını belirler.

Örnek: `dinamikNesne.Metot();`

10. `TryUnaryOperation(UnaryOperationBinder binder, out object result);`

Dinamik nesne tek operandlı operatörlerle kullanıldığında işlemin ne şekilde ele alınacağını belirler.

Örnek: `int sonuc = dinamikNesne++;`

Çalışma Sorusu

Aşağıdaki çağrım ifadelerinin geçerli olduğu ve çalışma zamanında ekrana "insert into TabloAdi (Ad, Soyad, No) values ('Ahmet','Altan','15')" şeklinde bir ifade yazmayı sağlayan Entity sınıfını geliştiriniz.

```
dynamic di = new Entity();

di.Ad = "Ahmet";
di.Soyad = "Altan";
di.No = 15;
Console.WriteLine(di.Insert());
```

Entity sınfının bütün özellik ve metotları dinamik olmalıdır.

13

Şablon Tipler (Generics)

- Şablon Tiplere Örnek: Liste Koleksiyonu
  - Liste'nin Kullanımı
  - Liste'nin Şablon Tür Versiyonu
- Çalışma Zamanı ve Şablon Türler
- Sınıf Aşırı Yükleme (Class Overloading)
- Şablon Tipler Arasında Türetme
- Şablon Tipler ve Arayüzler (Interface)
- Şablon Tiplerin Metotlara Etkisi
- default Operatörü
- Kısıtlar (Constraints)
- Şablon Tipli Metotlar
  - Şablon Tipi Çıkarsama (Type Inference)
- Şablon Tipli Temsilciler
- Generic Koleksiyonlarda Covariance ve Contravariance
- null Değer Alabilen Değer Tipleri (Nullable Value Types)
  - C# Sözdizimi Entegrasyonu
  - ?? Operatörü
- Tuple(Sınıf) ve ValueTuple(Struct) Şablon Veri Tipleri
  - C# 7.0 ve Tuple Tipleri
  - Deconstruct Özel Metodu ile Tuple İlişkisi
- Discard (_) İşareti ile Optimizasyon

Profesyonel programlamada, türden bağımsız algoritma geliştirme önemli bir tekniktir. Türden bağımsız algoritmalar, geliştirici için büyük kolaylıklar sağlamaktadır. Söz gelimi iki int türünden sayının toplanmasını sağlayan bir fonksiyonu yazdıktan sonra, aynı işlemi iki double türünden sayı için tekrarlamak zaman kaybına sebep olacaktır. C++ dilinde türden bağımsız algoritma kurabilmek için **şablon** (*template*) fonksiyonları ve şablon sınıfları kullanılmaktadır. C#, türden bağımsız algoritma geliştirmeye doğrudan destek vermiyor olsa da dolaylı yollardan türden bağımsız işlemler yapabilmek mümkündür. Bu işlemler C#'ta "Her şey bir Object'tir" cümlesinin altında yatan gerçekle halledilmektedir. C#'ta her şeyin bir nesne olması ve her nesnenin ortak bir atasının olması ve bu atanın da Object sınıfı olması bu cümlenin altında yatan gerçektir. Dolayısıyla herhangi bir türe ait referansı Object referanslarına atayabiliriz. Yani bir bakıma türden bağımsız bir işlem gerçekleştirmiş oluyoruz. Sözgelimi Object türünden bir parametre alan bir metodu dilediğimiz bir nesne referansı geçebiliriz. **Temel** (*base*) sınıfa ait referanslara, **türeyen** (*inherited*) sınıf referanslarını atayabilmek nesne yönelimli programlama tekniğinin sunduğu bir imkandır.

C#'ta Object referanslarına istenilen türden referanslar atanabilir. Bu, büyük bir imkan gibi görünse de aslında bazı dezavantajları da beraberinde getiriyor. Çünkü çalışma zamanında Object türüne atanmış referanslar orjinal türe tekrar geri dönüştürülmektedir. Kısaca unboxing olarak bilinen bu işlem özellikle değer (value) ve referans (reference) türleri arasında yapıldığında önemsenecek büyüklükte bir performans kaybı meydana getirmektedir. Çünkü değer ve referans türleri belleğin farklı bölgelerinde saklanmaktadır. Bu durum boxing ve unboxing işlemlerinin çalışma zamanında farklı bellek bölgeleri arasında uzun sürebilecek veri transferlerine sebep olur. Bu tür bir performans kaybını bazı veri yapıları için önlemek üzere C# 2.0'dan itibaren dil tasarımcıları Generics (Şablon Tipler) isimli bir kavramın dile eklenmesini sağlamıştır. Bu sayede bazı veri yapılarında, özellikle .NET sınıf kütüphanesindeki koleksiyon tipi veri yapılarında epeyce performans kazancı elde edilmiştir. Birazdan Şablon Tip mimarisini detaylı olarak inceleyeceğiz.

Şablon Tiplere Örnek: Liste Koleksiyonu

İsterseniz basit bir koleksiyon nesnesi üzerinden **generics** kavramının sağlayacağı yararları, boxing/unboxing işlemlerinin etkisini ve tip **güvenliği** (*type safety*) kavramını inceleyelim.

.NET sınıf kütüphanesinde de bulunan **ArrayList** isimli bir sınıf yardımıyla içinde her türden veri bulunduran ve istenildiğinde bu verilere erişilen bir koleksiyon nesnesi vardır. Koleksiyonlar bir sonraki bölümlerde detaylı olarak işlenecektir. Ancak biz örnek olması açısından dizilerle ilgili mevcut bilgimizi kullanarak buna benzer bir koleksiyon sınıfı yazacağız. Bu amaçla **Liste** isimli sınıfımız ile bütün veri türlerine ait işlemleri yapabilmek için saklanacak verilerin tipi **Object** olarak seçilecektir. Eğer bu böyle olmasaydı **Liste** sınıfının her bir tür için ayrı ayrı yazılması gerekecekti. Bu

mümkün olsa bile her şey bitmiş olmayacaktı. Çünkü Liste sınıfı kullanıcının tanımlayacağı türleri barındıracak duruma gelmez. İşte bütün bu sebeplerden dolayı Liste veri yapısında saklanan veriler Object olarak seçilmiştir. Buna göre Liste sınıfının kodları aşağıdaki gibidir;

```
using System;

namespace nongenericlistornegi
{
    public class Liste
    {
        private object[ ] elemanlar;

        private static int varsayilankapasite = 4;

        private int buyukluk;

        public Liste ()
  {
          elemanlar = new object[ varsayilankapasite] ;
        }

        public virtual int Ekle(object deger)
        {
            if (this.buyukluk == this.elemanlar.Length)
            {
              this.GerekliyseKapasiteArtir(this.buyukluk + 1);
            }
            this.elemanlar[ this.buyukluk] = deger ;
            return this.buyukluk++;
        }

        public object this[ int index]
        {
            get
            {
                if ((index < 0) || (index >= this.buyukluk))
                {
                    throw new Exception("yanlış index");
                }
                return this.elemanlar[ index] ;
            }
            set
            {
                if ((index < 0) || (index >= this.buyukluk))
                {
                    throw new Exception("yanlış index");
                }
```

```
                this.elemanlar[ index]  = value;
            }
        }

        private void GerekliyseKapasiteArtir(int min)
        {
            if (this.elemanlar.Length < min)
            {
    int adet = (this.elemanlar.Length == 0) ?  varsayilankapasite
: (this.elemanlar.Length * 2);
                if (adet < min)
                {
                    adet = min;
                }
                this.Kapasite = adet;
            }
        }

        public int Kapasite
        {
            get
            {
                return elemanlar.Length;
            }
            set
            {
                if (value != this.elemanlar.Length)
                {
                    if (value > 0)
                    {
                        object[ ] yenidizi = new object[ value] ;
                        if (this.buyukluk > 0)
                        {
Array.Copy(this.elemanlar, 0, yenidizi,    0,this.buyukluk);
                        }
                        this.elemanlar = yenidizi;
                    }
                    else
                    {
this.elemanlar = new object[ varsayilankapasite] ;
                    }
                }
            }
        }
    }
}
```

Yukarıdaki Liste sınıfındaki amacımız boyu dinamik olarak artan bir koleksiyon nesnesi yapmak. Her türlü elemanı saklayabilmek için ise sınıfın içinde object tipinden bir dizi tanımlanmıştır. Liste sınıfının üye elemanları üzerinden kısaca geçecek olursak;

- Yapıcı metot ile liste koleksiyon nesnesi yaratıldığı anda varsayılan olarak 4 elemanlı bir dizi yaratılır.

- Ekle metodu ile yapıcı metot ile oluşturulan diziye bir eleman ekleniyor. Eleman eklemek için dizinin boyutu kontrol ediliyor. Eğer eleman eklediğimizde dizinin boyutu aşılıyorsa **IndexOutOfRange** hatası alacağımızdan dizinin uzunluğunu **GerekliyseKapasiteArtir** isimli metot ile yeniden düzenliyoruz.

- **GerekliyseKapasiteArtir** metodu içinde dizinin uzunluğunu artırmak amacıyla yeni dizi uzunluğu hesaplanarak yeni kapasite belirleniyor.

- Dizi içindeki indeksleyici yardımıyla eklediğimiz herhangi bir elemana indeks numarası ile erişiyoruz.

- **Kapasite** isimli özellik içinde dizinin uzunluğunu artırmak amacıyla istenen yeni uzunlukta bir dizi yaratılıyor ve eski dizinin elemanları yeni diziye kopyalanıyor.

Liste'nin Kullanımı

Yukarıdaki Liste sınıfını kullanarak aşağıdaki koleksiyonlar oluşturmamız mümkündür.

```
Liste listem = new Liste();

listem.Ekle(3);
listem.Ekle(5);
listem.Ekle("Sefer");
```

Ekle metodunun parametresinin tipi object olduğundan her türlü veriyi koleksiyona alabiliyoruz. Güzel ve faydalı gibi görünse de aslında bu durumda object ile çalışmanın iki önemli dezavantajı vardır.

Birincisi Ekle metodunun parametresi object tipli olduğundan herhangi bir tür güvenliğinden bahsetmek mümkün değildir. Söz gelimi sadece Personel nesnelerini tutması düşünülen bir listeye herhangi bir aşamada int ya da başka bir türden veri eklenmesi derleme aşamasında herhangi bir hataya yol açmayacağından mümkün olabilir. Bu da çalışma zamanında beklenmedik hatalara neden olabilir.

İkinci önemli dezavantaj ise performans kaybı. Listeye eklenen bir nesneyi özellikle bu nesne değer tipinden ise önemli sayılabilecek bir performans kaybı olacaktır. Çünkü **Ekle** metodu ile bir **unboxing** işlemi olmuştur (boxing ve unboxing konuları **Tür Dönüşümü** isimli konuda detaylı olarak anlatılmıştır.). Veriyi alırken de unboxing işlemi olacağından gereksiz yere veri kopyalamaları olacaktır. Bu da yoğun olarak kullanılan değer tipli Liste koleksiyonu için önemli bir performans kaybına neden olur. Birazdan nasıl bir performans farkı olduğunu test edeceğiz. Örneğin:

```
listem.Ekle(3);
```

şeklinde eklenen bir değişkeni tekrardan geri almak için mecburen tür dönüştürme operatörü kullanılarak aşağıdaki gibi unboxing işlemi yapılmalıdır.

```
int deger = (int)listem [ 0] ;
```

Bu işlemler kendi tanımlayacağımız özel sınıflar için de geçerlidir. Ancak **int** ve **double** gibi temel veri türlerindeki performans kaybı daha fazladır. Çünkü **Ekle(3)** şeklindeki bir çağrımda boxing işlemi gerçekleşirken **listem[3]** indeksleyicisinin çağrılmasında unboxing işlemi gerçekleşir. Üstelik bu durumda indeskleyicinin geri dönüş değerini farklı bir türe dönüştürmeye çalışırsak derleme zamanında herhangi bir hata almayız. Bu da çalışma zamanında haberimiz olmadan bazı veri kayıplarının olabileceğini gösterir. Kullanıcı tanımlı sınıflar ile ilgili bir liste kullanıyorsak, indeksleyicinin geri dönüş değerini farklı bir kullanıcı tanımlı sınıfa çeviriyorsak bu sefer de derleme zamanında hata alınmaz, ancak çalışma zamanında **invalid cast operation** istisnai durumu meydana gelecektir.

Liste'nin Şablon Tür Versiyonu

Yukarıda değinilen bütün dezavantajlardan (tip güvenliği ve performans) kurtulmak amacıyla C# 2.0'dan itibaren C# diline ve CLR'ye **şablon tür** (*generics*) özelliği eklenmiştir.

Önemli Not: Şablon türler sadece derleme değil aynı zamanda çalışma zamanıyla da ilgilidir.

Şablon türler sınıf, metot ya da temsilci bazında herhangi bir üye elemanın tipinin bildirim aşamasında belli olmadığı durumlarda kullanılmaktadır. Örneğin Liste sınıfı türünden nesnelerin hangi türden eleman saklayacağı sınıf bildirimi sırasında belli değildir. Bu ancak Liste türünden nesneler yaratıldığında kullanıcı tarafından belirlenmektedir. Bu durumda Liste sınıfı bazında bir şablon tür tanımlayıp bu türün kullanıldığı yerde bunu kullanmak gerekir. C# 2.0 ile birlikte bir sınıf içinde şablon tür aşağıdaki gibi belirlenmektedir.

```
class Sınıf<SablonTur>
{

}
```

Dolayısıyla Sınıf türünden bir nesne yaratıldığında SablonTur'un hangi tip olduğu da aşağıdaki gibi belirtilmelidir.

```
Sınıf<int> s = new Sınıf<int>();
```

Dolayısıyla SablonTur'un kullanıldığı her yer artık int anlamına gelecektir. Çalışma zamanında derleyici SablonTur gördüğü yere int kelimesini bizim yerimize koyar.

Önemli Not: C++'taki **template**'lerde (şablonlar) şablon tipler derleme aşamasında belirlenirken C#'ta çalışma zamanında belirlenir.

Liste isimli koleksiyon örneğimizde dizi içinde tutulan veri tiplerini object yerine şablon bir tür olarak belirlersek hem performansı artırmış oluruz hem de koleksiyona sadece belirlediğimiz veri tiplerini ekleyerek tip güvenliğini sağlamış oluruz. Aşağıdaki örnek sadece bildirimleri içermektedir.

```
public class Liste<T>
{
    private T[ ] elemanlar;

    public virtual int Ekle(T deger)
    {
..
    }

    public T this[ int index]
    {
            ..
    }

}
```

Yukarıdaki prototipte görüldüğü üzere artık verileri saklayacak dizimizin tipi object değildir. Sınıf düzeyinde T isimli bir şablon tür tanımlanarak T tipinin çalışma zamanında ne olacağı Liste isimli sınıfı kullanacak kişi tarafından belirlenmesi sağlanmıştır. Dikkat edilirse elemanlar isimli dizideki object tipini temsil eden her yer T isimli şablon tür ile yer değiştirilmiştir. Dolayısıyla Liste nesnesi int türünden elemanlar saklayacak şekilde oluşturulduğunda bütün T şablon türleri int olarak CLR tarafından değiştirilecektir.

Aşağıdaki kod, ise Liste isimli sınıfımızın tamamının şablon tür versiyonudur.

```
using System;

namespace nongenericlistornegi
{
    public class Liste<T>
    {
        private T[ ] elemanlar;
        private static int varsayilankapasite = 4;

        private int buyukluk;
```

```
        public Liste ()
        {
              elemanlar = new T[ 0] ;
        }

        public virtual int Ekle(T deger)
        {
            if (this.buyukluk == this.elemanlar.Length)
            {
                this.GerekliyseKapasiteArtir(this.buyukluk + 1);
            }
            this.elemanlar[ this.buyukluk]  = deger ;
            return this.buyukluk++;
        }

        public T this[ int index]
        {
            get
            {
                if ((index < 0) || (index >= this.buyukluk))
                {
                    throw new Exception("yanlış index");
                }
                return this.elemanlar[ index] ;
            }
            set
            {
                if ((index < 0) || (index >= this.buyukluk))
                {
                    throw new Exception("yanlış index");
                }
                this.elemanlar[ index]  = value;
            }
        }

        private void GerekliyseKapasiteArtir(int min)
        {
            if (this.elemanlar.Length < min)
            {

int adet = (this.elemanlar.Length == 0) ?   varsayilankapasite :
(this.elemanlar.Length * 2);

                if (adet < min)
                {
                    adet = min;
                }
                this.Kapasite = adet;
            }
```



```
        }

        public int Kapasite
        {
            get
            {
                return elemanlar.Length;
            }
            set
            {
                if (value != this.elemanlar.Length)
                {
                    if (value > 0)
                    {
                        T[ ] yenidizi = new T[ value] ;
                        if (this.buyukluk > 0)
                        {
Array.Copy(this.elemanlar, 0, yenidizi, 0, this.buyukluk);
                        }
                        this.elemanlar = yenidizi;
                    }
                    else
                    {
                        this.elemanlar = new
T[ varsayilankapasite] ;
                    }
                }
            }
        }
    }
}
```

Liste sınıfımızın şablon versiyonunu yaptığımız andan itibaren yeni koleksiyon nesneleri ancak aşağıdaki gibi tanımlanabilir.

```
//Sadece int türünden veriler saklayabilen Liste koleksiyonu..
Liste<int> listem = new Liste<int>();
listem.Add(3);
listem.Add(5);
```

Yukarıdaki tanımlama yapıldıktan sonra,

```
listem.Add("Sefer");
```

kod bloğu henüz derleme aşamasındayken hata verecektir. Eğer Visual Studio.NET 2005 kullanıyorsanız şablon türlerin **intellisense** ile entegre olduğunu görebilirsiniz. Yani metot çağrım sırasında **Ekle** metodunun parametresinin int türünden olduğu size gösterilecektir.

Aynı şekilde birden fazla Liste elemanı da aşağıdaki gibi tanımlanabilir. Her liste elemanı sadece şablon tür ile belirlenen tipteki elemanları parametre olarak alabilecektir.

```
//Sadece string türünden veriler saklayabilen Liste koleksiyonu..
Liste<string> listem1 = new Liste<string>();
//Sadece Personel türünden veriler saklayabilen Liste koleksiyonu..
Liste<Personel> listem2 = new Liste<Personel>();
//Sadece DateTime türünden veriler saklayabilen Liste koleksiyonu..
Liste<DateTime listem3 = new Liste<DateTime>();
```

Listede saklanan elemanları elde etmeye çalıştığımızda aşağıdaki gibi herhangi cast işlemine de gerek kalmayacaktır. Bu da değer tipleri kullanıldığında boxing işleminin olmayacağı anlamına gelir.

```
Liste<int> listem = new Liste<int>();
listem.Add(3);
//cast işlemi yok. indeksleyici şablon tür döndürüyor.
int eklenen = listem[0];
```

Şablon tiplerin tip güvenliğini sağladığı açıkça görülmektedir. Performans farkını gözlemlemek için bir test programı yazabiliriz. Bir döngü içinde Liste koleksiyonuna 100000 adet int tipinden veri saklayarak Liste sınıfının her iki versiyonunun kaç saniye sürdüğünü gözlemleyebiliriz.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace nongenericlistornegi
{
    class Program
    {
        static void Main(string[] args)
        {
            DateTime dtBaslangic1 = DateTime.Now;
            Liste<int> listem = new Liste<int>();
            for (int i = 0; i < 1000000; i++)
            {
                listem.Ekle(i);
                int eleman = listem[i];
            }
            DateTime dtSon1 = DateTime.Now;
```



```
            TimeSpan tsSure1 = dtSon1 - dtBaslangic1;

            Console.WriteLine("Şablon Tür Versiyonu : " +
tsSure1.TotalMilliseconds);

            DateTime dtBaslangic2 = DateTime.Now;
            ListeObject listem2 = new ListeObject();
            for (int i = 0; i < 1000000; i++)
            {
                listem2.Ekle(i);
                int eleman2 = (int)listem2[i];
            }
            DateTime dtSon2 = DateTime.Now;

            TimeSpan tsSure2 = dtSon2 - dtBaslangic2;

            Console.WriteLine("Klasik Version : "tsSure2.
TotalMilliseconds);
        }
    }
}
```

Yukarıdaki test programını çalıştırdığımda aşağıdaki sonucu aldım. Yukarıdaki örneği çalıştırabilmeniz için Liste sınıfının hem şablon tür versiyonu hem de klasik versiyonunu yazmış olmanız gerekir. Her iki versiyon da bir önceki sayfalarda bulunmaktadır.

```
C:\WINDOWS\system32\cmd.exe
Şablon Tür Versiyonu : 31,25
Klasik Version : 109,375
Press any key to continue . . .
```

Yukarıdaki test programı basit bir örnek olmasına rağmen yaklaşık üç kat daha fazla performans elde edildiği açıkça görülmektedir.

Çalışma Zamanı ve Şablon Türler

İçerisinde şablon tür bulunan sınıflar türünden nesneler yaratıldığında her bir türün hafıza yapısı ve gerçek nesne modeli çalışma zamanında belirlenir. Yani derleme sırasında kod içinde tanımlanan her farklı şablon tür için ayrı bir sınıf bildirimi yaratılmaz. Zaten bu davranış biçimi C#'taki şablon türlerin C++ dilindeki template kavramından ayrıldığı noktadır.

Şablon tip belirlenerek oluşturulan her bir nesneye constructed type (inşa edilen tip) denilmektedir. Oluşturulan her bir tipin değer ya da referans tipi olmasına göre de çeşitli farklılıklar mevcuttur. Sözgelimi aşağıdaki gibi değer tiplerinden veri saklamak üzere 3 adet liste koleksiyonu tanımlamış olalım.

```
Liste<int> listem = new Liste<int>();
Liste<double> listem = new Liste<double>();
Liste<bool> listem = new Liste<bool>();
```

Yukarıdaki kod bloğu ile birlikte performans artırımının olabilmesi ve tip güvenliğinin sağlanması amacıyla çalışma zamanında aslında 3 adet farklı versiyonda Liste sınıfı oluşturulur. Yani CLR herhangi bir değer tipli **constructed type** oluşturursa anında belleğe yeni bir Liste koleksiyonu oluşturur. Yukarıdaki örnekte bellekte Liste isimli koleksiyonun 3 farklı versiyonu bulunur. Ancak referans tiplerinde durum biraz farklıdır. Referans tipleri zaten direkt olarak stack bellek bölgesinde bulunduklarından bir referans tipi sadece bir adresi temsil etmektedir. Bu yüzden her bir referans tipi aslında stack bellek bölgesi için aynı uzunluktadır. Referans tipleri için bellekte sadece bir adet Liste koleksiyonu oluşturulur ve hangi referans tipi kullanılırsa kullanılsın aynı versiyon kullanılır. Dolayısıyla şablon türlerin referans türleri ile birlikte kullanılması performans kazandırmayacaktır. Ancak bu durum referans tiplerini şablon tür olarak kullanmamamız gerektiği anlamına gelmez. Nitekim tip güvenliği ve şablon özelliği de önemli kriterlerdir. Özellikle **delegate** nesnelerinde kullanılan referans tipli şablon tipler önemli faydalar sağlamaktadır.

Soru: Aşağıdaki tanımlamalar sonrasında çalışma zamanında kaç adet Liste sınıfı versiyonu bulunur?

```
Liste<int> listem = new Liste<int>();
Liste<double> listem = new Liste<double>();
Liste<string> listem = new Liste<string>();
Liste<object> listem = new Liste<object>();
Liste<DateTime> listem = new Liste<DateTime>();
```

Cevap: 4

Sınıf Aşırı Yükleme (Class Overloading)

Liste koleksiyonu örneğinde dikkat ettiyseniz şablon tip versiyonun adını **Liste** klasik versiyonun adını **ListeObject** yaptık. Halbuki buna gerek yoktu. İkisinin de aynı olabilirdi. İki sınıf aynı isim alanında olsa bile biri şablon tür içereceği gibi derleme aşamasında ikisi birbirinden ayırt edilebilecektir. Yani artık şablon türler sayesinde aynı isimli birçok sınıf tanımlanabilir.

Buna göre aşağıdaki sınıf bildirimler tamamen geçerlidir:

```
namespace classoverloading
{
    class Liste
    {
    }
```



```
    class List<T>
    {

    }
}
```

Bir sınıf içinde ya da şablon tür bildirilebilecek her türlü eleman (interface, struct, delegate ya da metod) içinde birden fazla şablon tür de bulunabilir. Aşağıdaki kullanım tamamen geçerlidir.

```
class CiftListe<T,K>
{

}
```

Gördüğünüz gibi birden fazla şablon tip olduğunda tipler virgül ile ayrılmaktadır. Bu şekildeki bir sınıf yardımıyla yine aynı şekilde aşağıdaki gibi nesne oluşturulabilir.

```
CiftListe<int,string> listem =  new CiftListe<int,string>()
```

Sonuç olarak şablon tip sayısı farklı olduğu sürece istenildiği kadar aynı isimli sınıf bildirimi yapılabilir.

Önemli Not: Şablon tiplerin isimleri **K, V, U** gibi tek harfli ve büyük olmak zorunda değildir. Değişken isimlendirme kurallarına uyan her isim verilir. Ancak genellikle tek haneli karakterler verilmektedir.

Şablon Tipler Arasında Türetme

Bir şablon tipli sınıf başka bir şablon sınıf içeren sınıftan türeyebilir. Türetme, polimorfizm gibi konular ele alındığında şablon tiplerle ilgili birçok permutasyondan bahsetmek mümkün ancak biz bu kitap kapsamında şablon tiplerle ilgili sık kullanılan desenlere değineceğiz.

Aşağıdaki iki türetme biçimide geçerlidir.

```
namespace genericinheritance
{
    public class TabanSınıf<U>
    {
        private U eleman;
        public TabanSınıf(U param)
        {
            this.eleman = param;
        }
    }
```

```
    public class TureyenSınıf1<T, U> : TabanSınıf<U>
    {
        private T tureyeneleman;

        public TureyenSınıf1(T p1, U p2): base(p2)
        {
            this.tureyeneleman = p1;
        }
    }

    public class TureyenSınıf2<T> : TabanSınıf<int>
    {
        private T tureyeneleman;

        public TureyenSınıf2(T p1, int p2): base(p2)
        {
            this.tureyeneleman = p1;
        }
    }
}
```

Şablon Tipler ve Arayüzler (Interface)

Sınıflarda olduğu gibi arayüzler ile birlikte de şablon tipler kullanılabilmektedir. Örneğin .NET Framework içerisindeki **System.Collections.Generic** isim alanında birçok şablon tipli arayüz bulunmaktadır. Örneğin daha önceki konularda değindiğimiz **IEnumerable** arayüzünün artık şablon türlü versiyonu da bulunmaktadır. Yeni versiyonu aşağıdaki gibidir.

```
interface IEnumerable<T>
{
    IEnumerator<T> GetEnumerator()
}
```

IEnumerable içindeki T isimli şablon tip yine aynı şekilde kendisi gibi şablon tip alabilen **IEnumerator** arayüzüne gönderilmiştir. **IEnumerable<T>** arayüzü bir sınıfta aşağıdaki gibi uygulanabilir.

```
public class Sayac : IEnumerable<int>
{
IEnumerator IEnumerable.GetEnumerator()
{
  ...
}

IEnumerator<int> IEnumerable<int>.GetEnumerator()
    {
        ..
}
}
```


Dikkat edilirse GetEnumerator metodunun hem şablon tipli versiyonu hem de klasik versiyonu yazılmıştır. Bunun nedeni aşağıdaki gibi IEnumerable<T> nin IEnumerable arayüzünden türemiş olmasıdır. Dolayısıyla her iki arayüzde bulunan metotlarda uygulanmalıdır. (implement)

```
interface IEnumerable<T> : IEnumerator
{
...
}
```

Şablon Tiplerin Metotlara Etkisi

Sınıf düzeyinde tanımlanan şablon tipler, metotların geri dönüş değerleri ya da metotların parametrelerinin tipi olabilmektedir. Şablon tipler nesne yaratma zamanında belli olduğu için özellikle metot aşırı yükleme söz konusu olduğunda dikkatli olmak gerekir. Nitekim aynı şablon tipli olmayan ama üretilen nesneye bağlı olarak aynı imzaya sahip metotların çalışma zamanında oluşması muhtemeldir. Örneğin aşağıdaki örnekteki sınıf bildirimi geçerli bir bildirimdir.

```
public class Islemler<T>
{
    public int Metot1(int a,int b)
    {
        return a + b;
    }

    public int Metot1(T a, T b)
    {
        return 0;
    }
 }
```

Ancak aşağıdaki gibi bir nesne tanımlamak derleme zamanında hataya neden olmasa da kullanan açısından karışıklıklara neden olacağından bu tarz kullanımlardan çekinmek gerekir.

```
Islemler<int> t = new Islemler<int>();
t.Metot1(4, 5);
```

Yukarıdan görüldüğü üzere şablon tip olarak int seçildiğinden sınıf içerisinde bulunan diğer Metot1 ile çakışma olacaktır.

Şablon tipler metotlar ile birlikte kullanıldığı gibi, özellik, indeksleyici ve olaylar ile birlikte de kullanılabilir.

default Operatörü

Bazen sınıflar içinde kullanılan şablon tipler türünden nesneler tanımlanma ihtiyacı doğabilir. Özellikle geriye **null**, **false** ya da **0** gibi değerler döndürülme ihtiyacı hissedildiğinde bir takım zorluklarla karşılaşmamız mümkündür. Çünkü her bir tip null ya da 0 değeri alamaz. Bu yüzden sınıf içinde tip ile ilgili herhangi bir bilgiye sahip olamıyoruz. Generics mimarisi ile birlikte C# 2.0'a yeni bir operatör daha eklenmiştir. **default** operatörü bir şablon tipin C#'taki varsayılan değerini geri döndürür. Kullanımı aşağıdaki gibidir.

```
class Islemler<T>
{
    public T IslemYap(T v,int param)
    {
        if (param >= 0)
            return v;
        else
            return default(T);
    }
}
```

Şablon tip referans tipi ise default operatörü null değer döndürür. Eğer değer tipi ise ilgili tipin varsayılan değerini döndürür (bool için false, int için 0 gibi).

Bir şablon türünden nesne yaratmak için aşağıdaki gibi tanımlamalar yapılamaz.

```
class Islemler<T>
{
    public void Metot()
    {
      T degisken = null;
      T degisken = 0;
      ...
    }
}
```

Ancak aşağıdaki gibi değişken bildirimi yapabiliriz.

```
class Islemler<T>
{
    public void Metot()
    {
      T degisken = default(T);
      ...
    }
}
```

Kısıtlar (Constraints)

Generic mimarisi her ne kadar tipten bağımsız algoritma geliştirmede önemli bir katkı sağlasa da bazı dezavantajları da vardır. Örneğin şablon tür içeren bir sınıf içinde o şablon tür ile ilgili herhangi bir bilgiye sahip olmadığımız için şablon tür nesneleri üzerinde istediğimiz gibi işlem yapamayız.

Şöyle bir senaryo düşünelim: Kedi ve Kopek nesnelerimizin olduğunu düşünelim. Her iki sınıf içinde de Konus isimli bir metot bulunmaktadır. Şablon türler kullanılarak acaba her iki sınıftaki Konus metodu çağrılabilir mi?

```
using System;

namespace constraints
{
    class Kedi
    {
        public void Konus()
        {
            Console.WriteLine("havv");
        }
    }

    class Kopek
    {
        public void Konus()
        {
            Console.WriteLine("miyav");
        }
    }

    class Konusturma<T>
    {
        private T memeli;
        public Konusturma(T t)
        {
            memeli = t;
        }
        public void Konustur()
        {
            memeli.Konus();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Konusturma<Kedi> k = new Konusturma<Kedi>(new Kedi());
            k.Konustur();

            Konusturma<Kopek> k = new Konusturma<Kopek>(new
Kopek());
            k.Konustur();

        }
    }
}
```

Yukarıdaki örneği çalıştırdığımızda Kedi ve Kopek nesnelerinde Konus isimli metot olsa bile derleme hatası aldığınızı göreceksiniz. Çünkü bu örneğimizde şablon tipi Kedi ve Kopek nesneleri olmasına rağmen şablon tip hakkında herhangi bir kısıtlama yoktur. Yani int, double ya da Personel gibi nesneler de olabilir. Bu durumda bu nesneler içinde **Konus()** metodu olmayacağından tutarsız bir durum meydana gelecektir. Bu tür durumlarda bir şablon tür hakkında bazı kısıtlamalar koyabilmek amacıyla **constraints** dediğimiz çeşitli kısıtlayıcılar kullanılabilmektedir. Mesela **constructed type** olarak kullanılacak her tipin mutlaka **Konus()** metodu içermesi gibi.

C# 2.0'dan itibaren C# diline 5 adet kısıt eklenmiştir. Bunlar aşağıdaki tabloda verilmiştir.

Koşul	Açı
struct	şablon tipin struct olması gerektiği bildirilir.
class	şablon tipin sadece sınıf olabileceği bildirilir.
new()	şablon tipin mutlaka varsayılan yapıcı metot içermesi gerektiği bildirilir.
türetme	şablon tipin belirtilen bir tipten türemek zorunda olduğu belirtilir.
interface	şablon tipin belirtilen bir arayüzü uygulamak zorunda olduğu belirtilir.

İlk örneğimizi yukarıdaki **Konusturma<T>** isimli sınıfta bulunan T şablon türünün mutlaka **Memeli** isimli bir sınıftan türemiş olması gerekiğini bildireceğiz. Bu amaçla Kedi ve Kopek sınıfın türediği yeni bir Memeli sınıfı yazmamız gerekecektir.

Şablon tiplere kısıtlar (constraints) C# diline eklenen where isimli bir anahtar sözcükle yapılmaktadır.

```
using System;

namespace constraints
{
    abstract class Memeli
    {
        abstract public void Konus();
    }

    class Kedi : Memeli
    {
        public override void Konus()
        {
            Console.WriteLine("havv");
        }
    }

    class Kopek : Memeli
```



```
    {
        public override void Konus()
        {
            Console.WriteLine("miyav");
        }
    }

    class Konusturma<T> where T : Memeli
    {
        private T memeli;
        public Konusturma(T t)
        {
            memeli = t;
        }
        public void Konustur()
        {
            memeli.Konus();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Konusturma<Kedi> k1 = new Konusturma<Kedi>(new Kedi());
            k1.Konustur();

Konusturma<Kopek> k2 = new Konusturma<Kopek>(new Kopek());
            k2.Konustur();

        }
    }
}
```

Yukarıdaki örnekten de görüldüğü üzere T şablon türü Memeli sınıfından türeyen herhangi bir nesne olmak zorunda olduğu için aşağıdaki gibi bir kullanım tamamen geçerli hale gelmiştir. Kısıtlama sayesinde şablon tür nesneleri içinde **Konus()** isimli metodun olması garanti altına alınmıştır. Böylece **constructed type**'lar yalnızca Memeliden türeyen nesneler tarafından oluşturulabilecektir. Örneğin

```
Konusturma<int> t = new Konusturma<int>();
```

gibi bir tanımlama artık derleme zamanında hata verecektir.

```
private T memeli;
public void Konustur()
{
    memeli.Konus();
}
```

Yukarıdaki kod bloğu geçerli olmasına rağmen aşağıdaki kod bloğu geçerli değildir.

```
public void MemeliKonustur()
{
    T memeli =  new T();
    memeli.Konus();
}
```

Yukarıdaki kod bloğunun geçerli olmamasının sebebi T şablon türünün **new** operatörü ile kullanılmasıdır. Bildiğiniz gibi new operatörünün kullanılabilmesi için o sınıfın varsayılan yapıcıya sahip olması gerekir. T şablon türünden new ile bir nesne yaratabilmek için bunu garanti altına almak gerekiyor. Bu amaçla **new()** isimli kısıtlaycı kullanılmaktadır.

new() kısıtlayıcı sayesinde şablon türler kullanılarak new operatör ile nesne yaratmak mümkün hale gelmektedir. Kullanımı aşağıdaki gibidir.

```
class Konusturma<T> where T : new()
{
}
```

Yukarıdaki bildiriminden sonra varsayılan yapıcı metodu olmayan veri tipleri **Konusturma<T>** sınıfındaki T şablon tipi yerine geçemez.

Diğer bir önemli kısıtlayıcı ise arayüz kısıtlayıcısıdır. Arayüz kısıtlayıcı bir şablon türün belirtilen arayüzden türemesi gerektiğini bildirir. Arayüz kısıtlarına bir örnek verelim.

```
using System;

namespace constraints
{
    class Karsilastirici<T> where T : IComparable<T>
    {
        public static int Karsilastir(T a, T b)
        {
            return a.CompareTo(b);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
           int s1=Karsilastirici<int>.Karsilastir(4, 5);
            int s2=Karsilastirici<float>.Karsilastir(2.3f, 2.3f);
           int s3=Karsilastirici<string>.Karsilastir("Veli", "Ali");
int s4=Karsilastirici<DateTime>.Karsilastir(DateTime.Now,
DateTime.Now.AddDays(1));
```



```
            Console.WriteLine(s1);
            Console.WriteLine(s2);
            Console.WriteLine(s3);
            Console.WriteLine(s4);
    }
    }
}
```

Yukarıdaki örnekte **Karsilastir<T>** isimli sınıfta bulunan T şablon türünün mutlaka **IComparable** arayüzünden türemiş olması gerektiği bildirilir ki **Compare** isimli metot kullanılabilsin. Böylece karşılaştırılmayan nesneler **Karsilastir<T>** isimli sınıfta kullanılamayacaktır.

Örneği çalıştırdığımızda ekrana

```
-1
 0
 1
-1
```

yazdığı görülecektir.

ICompareble arayüzü içindeki Compare metodunu uygulayan her nesne karşılaştırılabilir anlamına gelir. **0** iki nesnenin eşit olduğunu **1** değeri ilk nesnenin diğerinden büyük **-1** ise küçük olduğunu belirtmektedir.

Diğer kısıtlayıcılar şablon tipleri sadece **sınıf** veya **struct** olmaya zorlayan kısıtlayıcılardır. Bir şablon türün genel olarak sadece sınıf ya da struct olması gerektiği bildirilmek istenirse aşağıdaki gibi yapılabilir.

```
class AritmetikIslemler<T> where T : struct
{
}
```

ya da

```
class Hayvan<T> where T : class
{
}
```

Şablon türler hakkında elbette ki birden fazla kısıtlayıcı bildirmek mümkündür. Birden fazla kısıtlayıcı bildirmek için kısıtlayıcılar virgül ile ayrılır. Örneğin hem new() hem class hem de arayüz kısıtlayıcısı aşağıdaki gibi bildirilebilir.

```
class Hayvan<T> where T : class, IComparable, new()
{
}
```

Birden fazla şablon tür söz konusu olduğunda her biri için ayrı ayrı kısıtlamalar tanımlanabilir. Bunun için şablon tür sayısı kadar **where** anahtar kelimesi eklemek gerekmektedir. Birden fazla where kullanımına aşağıda bir örnek verilmiştir.

Önemli Not 1: Kısıtlamaların birkaçını bir arada kullanabiliriz. Buradaki tek şart, eğer varsayılan yapıcı kısıtlaması da var ise new anahtarının her zaman için en sonda belirtilmesi gerekliliğidir.

Önemli Not 2: Kısıtlamalar çalışma zamanı değil derleme zamanı ile ilgilidir.

```
public class İkilisListe<AnahtarTuru, DegerTuru>
        where AnahtarTuru : IComparable,IEnumerable
        where DegerTuru : Musteri
{
    public void Ekle(AnahtarTuru anahtar, DegerTuru deger)
    {
        switch (anahtar.CompareTo(x))
        {
          ...
        }
    }
}
```

Şablon Tipli Metotlar

Şimdiye kadar bütün örneklerimizde kullandığımız şablon tipler sınıf düzeyinde tanımlanmıştı. Ancak şablon tipler sadece sınıf seviyesinde değil metot ve temsilci seviyelerinde de tanımlanabilir. Bu bölümde metot bazında şablon tiplerin ne şekilde tanımlandığını inceleyeceğiz.

Metot seviyesinde şablon tip tanımlama biçimi, sınıf seviyesinde tanımla biçimiyle hemen hemen aynıdır. Örneğin iki değişkenden en büyüğünü veren **EnBuyuk()** isimli şablon bir metot aşağıdaki gibi hazırlanabilir.

```
using System;

namespace constraintsmetot
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(EnBuyuk<string>("Ali", "Veli"));
            Console.WriteLine(EnBuyuk <float>(3.2f, 2.4f));
        }

        public static T EnBuyuk<T>(T p1, T p2) where T : IComparable
```



```
        {
            T retval = p2;
            if (p2.CompareTo(p1) < 0)
                retval = p1;

            return retval ;
        }
    }
}
```

Yukarıdaki program çalıştırıldığında ekrana,

Veli
3.2

yazdığı görülecektir. Örnekten de anlaşıldığı gibi metotlarda bildirilen şablon türler üzerinde de aynen sınıflarda olduğu gibi çeşitli kısıtlamalar kullanılabilir. Şablon tür mimarisi olmasaydı da yukarıdaki metodu tek seferde yazabilirdik ancak bu kez metodun parametresi IComparable tipinden olması gerekecekti.

Şablon Tipi Çıkarsama (Type Inference)

Bir önceki örnekte bulunan EnBuyuk() isimli metot çağrılırken dikkat edilirse, T şablon tipi bilinçli bir şekilde belirtilmiştir. Ancak C# derleyicisi parametre olarak gönderilen şablon tipli nesnelerin özelliğine bakarak otomatik bir çıkarsama yapabilmektedir. Yani **EnBuyuk** isimli metotları da aşağıdaki gibi çağırmamız herhangi bir hataya neden olmayacaktır.

```
string buyuk1 = EnBuyuk("Ali", "Veli");
int buyuk2 = EnBuyuk(3.2f, 2.4f);
```

EnBuyuk() metoduna gönderilen parametrelere göre şablon tipin ne olacağına karar verilmektedir.

Şablon Tipli Temsilciler

Temsilciler de bildirildikleri noktada şablon tür alabilirler. Bu sayade parametre sayısı aynı olan bütün metotları tek temsilci ile temsil etme şansına sahip oluruz. Örneğin iki **int** sayıyı veya iki **float** sayıyı toplayan iki metodu temsil edecek tek bir temsilci aşağıdaki gibi bildirilebilir.

```
using System;

namespace constraintsmetot
{
    class Program
    {
        public delegate T Temsilci<T>(T s1, T s2);
```

```
        public static int Topla(int a, int b)
        {
            return a + b;
        }

        public static float Topla(float a, float b)
        {
            return a + b;
        }

        static void Main(string[] args)
        {
            Temsilci<int> t1 = new Temsilci<int>(Topla);
            Console.WriteLine(t1(3,4));

            Temsilci<float> t2 = new Temsilci<float>(Topla);
            Console.WriteLine(t2(3.3f, 4.4f));
        }
    }
}
```

Yukarıdaki program çalıştırıldığından her iki **Topla()** versiyonun da çalıştığını göreceksiniz. Böylece tek bir temsilci bildirimi ile farklı imzalara sahip metotları temsil edebilme şansına sahip oluyoruz. **Generic** temsilciler olmasaydı yukarıdaki her iki **Topla()** metodunu temsil edecek iki ayrı temsilci yazmak zorunda kalacaktır.

Temsilci şablon tipleri de diğer tipler gibi kısıtlayıcı alabilirler. Örneğin yukarıdaki örneğimizde T tipinin yalnızca **struct** olmasını istersek aşağıdaki gibi bir bildirim yapabiliriz.

```
public delegate T Temsilci<T>(T s1, T s2) where T : struct;
```

Şablon tipli temsilcilerin faydası özellikle olay tabanlı programlama modellerinde gözükmektedir. Özellikle nesneler arasında mesaj alışverişi yapabilmek için olaylar kullanılır. Olaylar, ateşlendiği nesnelere bir takım mesajlar da gönderir. Bu mesaj gönderimler event handler dediğimiz olay yöneticisi olan temsilciler yardımıyla yapılır. Birçok olay olabileceği için her biri için ayrı bir temsilci tanımlamak yerine olay parametresini (eventargs) şablon tür olarak tek bir şablon tipli temsilci tanımlamak daha mantıklıdır. .NET Framework 2.0'dan itibaren System isim alanında EventHandler<TEventArgs> isimli bir şablon tipli temsilci bildirilmiştir. Bu temsilci yardımıyla birçok olayı (event) yönetmek mümkün hale gelmiştir.

```
public delegate void EventHandler<TEventArgs>;
```

.NET Framework 2.0 kütüphanesi içerisinde bu kitapta değiştiremeyeceğimiz birçok şablon tipli temsilci bulunmaktadır. Daha detaylı bilgi edinmek isterseniz elbette MSDN kaynağına başvurmanızda fayda var.

Generic Koleksiyonlarda Covariance ve Contravariance

C# 2.0'dan itibaren şablon tipli koleksiyonlarda geliştirilmiştir. (Detaylı bilgi için Koleksiyonlar bölümünü inceleyiniz.) Ancak şablon tipli koleksiyonlarda önemli bir problem ancak C# 4.0 versiyonunda çözülebilmiştir. Sorun biribirinden türemiş tipler koleksiyon içinde bulunduğunda koleksiyon tipleri farklılaştığı için birbirine atanamıyordu. Sözgelimi içinde int türünden elemanların olduğu bir liste nesnesini içinde object türden nesne tutan bir koleksiyona referans vermek mümkün değildi. Yani aşağıdaki kod bloğu C# 4.0 öncesi versiyonlarda derleme hatası vermektedir.

```
IList<string> strings = new List<string>();
IList<object> objects = strings;
```

Yukarıdaki kod bloğunun çalışmaması bir bakıma tür güvenliğinin de garantisidir. Çünkü yukarıdaki kod eğer geçerli olsaydı aşağıdaki ifade de geçerli olmak zorunda olurdu ama bu sefer de tür güvenliği sağlanamadığı için ikinci satır çelişkili bir durum oluşturacaktı. Çünkü int türü string türüne doğal olarak dönüştürülemez.

```
objects[1] = 5;
string s = strings[0];
```

Eğer bu dönüşüm işleminden sonra elemanlar üzerinde oynama yapmamayı garanti edecek bir sınıf ya da arayüz olursa bu atama mantıklı olabilir. Örneğin `IList<T>` yerine `IEnumerable<T>` kullanılabilir. IEnumerable üzerinden koleksiyonda değişiklik yapılamayacağı için yukarıdaki şaibeli kod bloğu da otomatikman yazılamaz duruma gelecektir.

C# 4.0'dan itibaren bu atamayı (string listesinin object listesine atanabilmesi) geçerli yapabilmek için şablon tipli arayüzlerde kullanmak şartıyla özel bir bildirim şekli gelmiştir. Buna göre şablon tip "out" anahtar sözcüğü ile aşağıdaki gibi bildirildiğinde bu imkan sağlanmaktadır. Bu işleme "covariance" denilmektedir.

```
public interface IEnumerator<out T> : IEnumerator
{
    bool MoveNext();
    T Current { get; }
}
```

Yukarıdaki işlemin tam tersine ise contravariance denilmektedir. Çoğunlukla şablon tipin aynı zamanda input parametre olduğu durumlarda özellikle şablon tipli temsilcilerde ihtiyaç duyulmaktadır. Bu işlem şablon tipin "in" anahtar sözcüğü ile işaretlenmesi ile gerçekleştirilir. Örnek bir bildirim aşağıdaki gibi yapılabilir;

```
public delegate Temsilci<in T>(T param);
```

Yukarıdaki bildirime göre `Temsilci<object>` tipindeki bir temsilci, `Temsilci<string>` tipindeki bir temsilci değişkeni olarak da kullanılabilir. Yani aşağıdaki kullanım geçerli olmaktadır.

```
Temsilci<object> func = new Temsilci<object>(Metot1);

func(7) // metot1 çağrılır.
func("Ali"); // metot1 çağrılır

Metot1(object obj){ ...}
```

in ve out ile bildirilen şablon tipler ancak interface ve delegate'ler üzerinde tanımlanabilir. Diğer bir kısıt ise dönüşüm işleminin referans dönüşüm işlemi olma zorunluluğudur. out ile bildirilen şablon listede, int elemanlar içeren koleksiyon object eleman içeren koleksiyona atanamaz. Çünkü int-object dönüşümü bir referans dönüşümü değil boxing işlemidir.

null Değer Alabilen Değer Tipleri (Nullable Value Types)

Generic mimarisine verilebilecek belki de en güzel örneklerden biri .NET Framework 2.0'a eklenmiş Nullable<T> isimli bir yapıdır. Bu yapı değer tiplerine null değer alabilme özelliği katmayı hedefleyen bir sarmalayacı (wrapper) yapıdır. Şablon tür üzerinde struct kısıtlayıcı olduğundan Nullable<T> sadece değer tipleri ile birlikte kullanılabilir. Nullable veri tipleri daha çok veritabanı uygulamalarında ön plana çıkmaktadır. Çünkü hemen hemen bütün veritabanı sistemlerinde bulunan int, decimal gibi veri tipleri null olabilmektedir. Bu tür değerler null olabildiği için C# kodu ile bu alanlara erişilmek istendiğinde bir takım null kontrolleri yapmak gerekir. Nullable<T> tipi ile bu işlemin biraz daha kolaylaştırılması hedeflenmiştir.

System.Nullable<T> yapısı genel olarak aşağıdaki gibi bildirilmiştir.

```
public struct Nullable<T> where T: struct
{
    private T value;
    private bool hasValue;

    public T Value { get { ... } }
    public bool HasValue { get { ... } }
    public T GetValueOrDefault() { ... }

}
```

`Nullable<T>` yapısı iç yapısında çok karışık değildir. Aslında verilerin değerleri yine normal bildiğimiz tiplerde tutulmaktadır. İçerdeki bool türünden bir değişken

vasıtası ile değerin **null** olup olmadığı bir şekilde kontrol ediliyor.

Buna göre **null** değer alabilen **int** ve **double** türünden değişkenler aşağıdaki gibi tanımlanabilir.

```
Nullable<int> a = new Nullable<int>();
a = 5;
Nullable<double> d = 5;
```

Tabiki buradaki önemli nokta a ve d değişkenlerine aynı zamanda null değeri de atayabileceğimizdir. Aşağıdaki kullanımlar da tamamen geçerlidir.

```
Nullable<int> a = new Nullable<int>();
a = null;
Nullable<double> d = null;
```

Nullable<T> yapısı içerisinde bulunan **GetValueOrDefault()** isimli metot ile nullable tipe atanan değeri elde edebiliriz. Eğer değer null ise bize o tipin varsayılan değerine verecektir. Örnek vermek gerekirse;

```
Nullable<int> a = new Nullable<int>(4);
Nullable<double> d = null;
Console.WriteLine(a);
Console.WriteLine(d.GetValueOrDefault());
```

Yukarıdaki kod bloğunu derleyip çalıştırdığınızda ekrana,

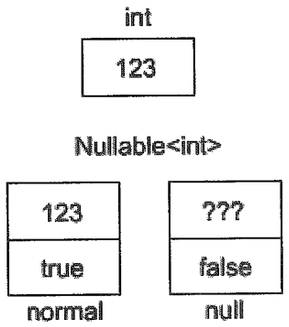
4
0

yazdığını göreceksiniz.

Value isimli özellik ise kullanılan şablon tip türünden bir değişkendir ve değişkenin değerini temsil eder. Eğer değişkenin değeri null ise varsayılan değer yerine null vereceğinden aşağıdaki kod çalışma zamanında istisna oluşturacaktır.

```
Nullable<double> d = null;
Console.WriteLine(d.Value);
```

Kısaca değer tipleri ve null alabilen değer tiplerini yandaki şema ile özetleyebiliriz.



Nullable tipler

C# Sözdizimi Entegrasyonu

Nullable<T> yapısının çok fazla kullanılacağı tahmin edilerek bu yapının kullanımı C# dilinin sözdizimine entegre edilmiştir. **?** takısı ile değer tipleri kullanıldığında derleyici o ifadeyi otomatik olarak Nullable yapısına dönüştürmektedir. Kısaca null değer alabilen bir double değişkeni

```
Nullable<double> d = 10;
```

şeklinde tanımlanabileceği gibi kısa yoldan aşağıdaki gibi de tanımlanabilir.

```
double? d = 5;
```

Önemli Not: double ile **double?** tamamen farklı veri tipleridir.

double? tipi aslında çalışma zamanında **Nullable<double>** yapısına dönüşeceği için **Nullable<double>** için söyleyeceğimiz bütün kurallar **double?** için de geçerli olacaktır. Dolayısıyla double? türden bir değişkene aşağıdaki gibi null bir değer ataması yapılabilir.

```
double? d = null;
```

Nullable veri tipleri normal veri tiplerine dönüştürülebilir. Ancak bunun için tür dönüştürme operatörünü açıkça kullanmak zorundayız. Örneğin **int?** tipinden bir değişkeni **int** türüne aşağıdaki gibi dönüştürebiliriz.

```
int? a = 5;
int b = (int)a;
```

Aynı şekilde tür dönüştürme kurallarına uymak şartı (*bkz: Tür Dönüşümü Konusu*) ile farklı türlere de dönüştürmek mümkündür. Yani aşağıdaki tür dönüşümü tamamen geçerlidir.

```
int? a = 5;
double b = (double)a;
```

Ancak bu şekildeki bir tür dönüşümünde eğer dönüşecek tür null değer içeriyorsa çalışma zamanında hata alınacaktır.

```
int? a = null;
int b = (int)a; //çalışma zamanı hatası
```

Nullable ve normal tipler arasında ters taraflı dönüşüm de mümkündür. Yani normal bir tipin nullable bir tipe dönüştürülmesi de geçerlidir. Bu işlem için açıkça tür dönüşüm operatörünü kullanmamız gerekmiyor. Yani aşağıdaki gibi bilinçsiz tür dönüşümü uygulanabilmektedir.

```
int a = 5;
int? b = a;
```


Nullable veri tiplerini bir ifade içinde birlikte kullanmak da mümkündür. Örneğin iki nullable değer tipini + operatörü ile aşağıdaki gibi kullanabiliriz.

```
int? a = 5;
int? b = 6;

int c = (int)(a + b);
```

+ operatörün ürettiği değer yine **int?** olduğundan sonucu **int** türünden bir değişkene atayabilmek için bilinçli bir şekilde tür dönüştürme operatörü kullanıldı.

?? Operatörü

C# 2.0 ile birlikte dile Nullable<T> yapısında bulunan **GetValueOrDefault()** metodunun yaptığı işe benzer bir iş yapan yeni bir operatör eklenmiştir. Bu operatör sayesinde **T?** tiplerinin null olması durumunda **if-else** bloklarını yazmadan farklı bir değer olarak elde etmek için kullanılır. Örneğin **int?** tipindeki bir değişkenin **null** olması durumunda **0** değerini alması için **??** operatörü aşağıdaki gibi kullanılabilir.

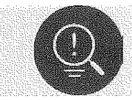
```
int? a = null;
int b = a ?? 0;
int? c = 50;
int d = c ?? 0;
Console.WriteLine(b);
Console.WriteLine(d);
```

Yukarıdaki program çalıştırıldığında ekrana;

0
50

yazdığı görülecektir.

?? Operatörünün ikinci operatörü tiplerin varsayılan değeri olmak zorunda değildir. Örneğin **c ?? 100** ifadesi de geçerlidir.



Null Condition Operatörü (?)

Sadece C# 6.0'dan itibaren geçerli olmak üzere yeni bir operator daha gelmiştir. Bu operatör birden fazla kod bloğunda yaptığımız işlemi tek bir satırda yapmamızı sağlamaktadır. Kod içerisinde sıklıkla karşılaştığımız "null" değer kontrol etme blokları bu operator sayesinde artık gereksiz olmuştur. Aşağıdaki gibi bir sınıfmızın olduğunu düşünelim;

```
public class Personel
{
    public int No { get; set; }
    public string Ad { get; set; }
    public Personel Yonetici { get; set; }
}
```

Bir metot düşünelim ve parametre olarak Personel nesnesi almış olsun. Normal şartlarda bu nesne ile çalışırken bu nesnenin "null" olup olmadığını aşağıdaki gibi kontrol ederiz. (Elbette çalışma zamanında hata almak istemiyorsak)

```
public void BilgiGoster(Personel p)
{
    //eski yöntem
    if(p != null)
       Console.WriteLine(p.Ad);
}
```

C# 6.0'dan itibaren ? operatörünü kullanarak bu kod bloğunu aşağıdaki gibi değiştirebiliriz.

```
public void BilgiGoster(Personel p)
{
    Console.WriteLine(p?.Ad);
}
```

Yukarıdaki kod bloğunda "p?" ifadesi ile eğer p nesnesi null ise "." operatörü devreye girmez ve null değer üretir. Böylece Null olan bir nesne üzerinden nokta operatoü ile özelliklere ya da diğer üye elemanlara erişilmez.

? operatörü "." operatorü ile birlikte iç içe de aşağıdaki gibi kullanılabilir. (Bu örnekte bir personelin iki üst yöneticisine erişilmeye çalışılıyor)

```
Console.WriteLine( p?.Yonetici?.Yonetici?.Ad );
```

Eğer ? operatorü olmasaydı yukarıdaki tek satır kod bloğunu şu şekilde yazmalıydık;

```
if(p!= null)
    if(p.Yonetici !=null)
       if(p.Yonetici.Yonetici!=null)
          Console.WriteLine(this.Yonetici.Yonetici.Ad);
```

? operatörünü atama opertörleri ile kullanıdğımızda dikkat etmemiz gereken bir durum vardır. Örneğin aşağıdaki ifadeyi nasıl bir değikene atayabileceğimizi tahmin etmeye çalışın;

```
Personel p = new Personel();
.... =  p?.No;
```


İkinci satırdaki ifade p nesnesinin durumuna göre int değer ya da null değer üretebilir. Bu durumda bu ifadeyi int tipinden bir değişkene atayamayız. Tabi ki bu durumda kullanmamız gereken nesne Nullable<int> yada int? dir. Doğru kullanım aşağıdaki gibi olacaktır;

```
Personel p = new Personel();
int? no =  p?.No;
```

nullable<> tipi sadece değer tiplerinde kullanılır. Çünkü referans tipler zaten null değer alabilmektedir. Dolayısıyla ? işaretinin kullanıldığı bir ifade değer tipi değil de referans tipi dönüyorsa bu durumda nullable<> kullanılmaz.

Örneğin aşağıdaki ifade tamamen geçerlidir, çünükü Personel nesnesi zaten null değer alabilir.

```
List<Personel> personeller = new List<Personel>();
Personel  p =  personeller?[0];
```

? operatörü ile ?? operatörünü birlikte kullanarak güzel bir kullanım elde edebiliriz.

Örneğin;

```
public void(List<Personel> personeller)
{
        int personelSayisi = personeller?.Count ?? 0;
}
```

Tahmin edeceğiniz gibi **şablon tür** (*generic*) mimarisi CLR ve C# dilinde oldukça fazla değişikliğin oluşmasına neden oldu. Bu bölümde ana hatlarıyla şablon türlerle ilgili en çok karşılaşılabilecek durumları inceliyoruz. Projelerinizde şablon türleri kullandıkça ya da varolan projelerin kodlarını okumaya başladıkça şablon türlerin faydasını daha iyi anlayacağınızı tahmin ediyorum.

Tuple(Sınıf) ve ValueTuple(Struct) Şablon Veri Tipleri

Tuple tipi 2010 yılında, Microsoft .Net 4.0 sürümüyle beraber .NET Framework'e bir kütüphane olarak eklendi. C# 7.0 sürümüyle beraber Tuple tiplerinde önemli geliştirmeler yapıldı. Bu geliştirmelere değinmeden önce genel olarak Tuple tipinin ne olduğunu ve eski sürümlerde nasıl kullanıldığını inceleyeceğiz.

Tuple matematikte sonlu serilere verilen isimdir. Programlamada ise bir tuple, belirli bir sayı ve eleman sırasına sahip bir veri yapısıdır. Örneğin bir tuple içerisinde, ilk eleman bir kişinin adı, ikinci eleman kişinin doğum yılı ve üçüncü eleman telefon nu-

marası olabilir. Bu üç öğeye sahip Tuple tipine 3-tuple veya üçlü (triple) denilir. Tuple aynı zamanda bir şablon tiptir.

.Net Framework 8-tuple veya sekizli (octuple) tuple tipine kadar destekler. Bu tarz çoklu Tuple nesneleri Tuple sınıfının static Create isimli aşağıdaki metotları ile yaratılabileceği gibi Tuple<string,string>, ya da Tuple<string,string,int> şablon tipleri kullanılarak da yaratılabilir.

Name	Description
Create<T1>(T1)	**1-tuple** veya tekli yaratır.
Create<T1, T2>(T1, T2)	**2-tuple veya çiftli yaratır.**
Create<T1, T2, T3>(T1, T2, T3)	**3-tuple** veya üçlü yaratır.
Create<T1, T2, T3, T4>(T1, T2, T3, T4)	**4-tuple** veya dörtlü yaratır.
Create<T1, T2, T3, T4, T5>(T1, T2, T3, T4, T5)	**5-tuple** veya beşli yaratır.
Create<T1, T2, T3, T4, T5, T6>(T1, T2, T3, T4, T5, T6)	**6-tuple** veya altılı yaratır.
Create<T1, T2, T3, T4, T5, T6, T7>(T1, T2, T3, T4, T5, T6, T7)	**7-tuple** veya yedili yaratır.
Create<T1, T2, T3, T4, T5, T6, T7, T8>(T1, T2, T3, T4, T5, T6, T7, T8)	**8-tuple** veya sekizli yaratır.

Tuple çoğunlukla aşağıdaki amaçlar için kullanılır;

- Tek bir veri kümesini temsil etmek. Örneğin, bir tuple bir veritabanı kaydını temsil edebilir ve elemanları kaydın alanlarını temsil edebilir.
- Bir veri tipini kolay bir şekilde modellemek için kullanılabilir.
- out parametresi kullanmadan bir metotdan birden fazla değer dönmek için kullanılabilir.

ValueTuple yapısı ise Tuple'ın aynı özelliklerine sahiptir ancak Tuple gibi sınıf değil yapı(struct) olarak tanımlanmıştır. ValueTuple kullanabilmek için minimum .NET Framework 4.7 kurulması gerekmektedir. ValueTuple'ın nerede kullanıldığını bu bölümde inceleyeceğiz.

Tuple yaratırken nesne içerisine eklediğimiz verilere ise Item1,Item2,Item3 şeklinde önceden tanımlanmış isimlerle ulaşabilmekteyiz. Genel hatlarıyla ikili bir Tuple'ın aşağıdaki gibi tanımlandığını söyleyebiliriz.


```
public class Tuple<T1, T2>
{
    public T1 Item1 { get; }
    public T2 Item2 { get; }
}
```

Tuple kullanımına bir örnek verelim. Herhangi bir kaynaktan (veritabanı ya da dosya) çektiğimiz bir kullanıcı verisini bir Tuple tipine aşağıdaki gibi aktarabiliriz.

```
using System;

public class Program
{
    public static void Main()
    {
        var user = KullaniciGetir(2);

        int id = user.Item1;
        string isim = user.Item2;
        string soyisim = user.Item3;
        string eposta = user.Item4;

        Console.WriteLine("Id : " + id);
        Console.WriteLine("İsim : " + isim);
        Console.WriteLine("Soyisim : " + soyisim);
        Console.WriteLine("Eposta : " + eposta);
    }

    public static Tuple<int, string, string, string>
KullaniciGetir(int kullaniciId)
    {
        //Kaynaktan(dosya,veritabanı vs) veri alma işlemleri
        Tuple<int, string, string, string> kullanici = Tuple.
Create<int, string, string, string>(1, "Sefer", "Algan", "sefer.
algan@btakademi.com");
        //yada
        //var kullanici = new Tuple<int, string, string,
string>(1, "Sefer", "Algan", "sefer.algan@btakademi.com");
        return kullanici;
    }
}
```

Yukarıdaki kodu incelediğimiz zaman KullaniciGetir metodu içerisinde Tuple.Create yardımıyla bir 4-tuple veya dörtlü yaratıldığını görüyoruz.

Her ne kadar Tuple kullanımı bir veri tipini modellememizi kolaylaştırsa da, aşırı kullanımı kod okunabilirliğini ve bakım yapılabilirliğini düşürmektedir. Örneğin Item1, Item2 vb. elemanların neyi ifade ettiklerini ilk bakışta anlamak zordur ve yukarıdaki gibi değişkenlere atama yapılmadığı sürece kafa karışıklığına sebep olabilir. Ayrıca Tuple elemanlarının kendilerinin de Tuple alabildikleri de göz önünde bulundurulursa, dikkatli olunmadığı durumlarda kodun karmaşıklığı artabilir ve hata ayıklamak zorlaşabilir.

C# 7.0 ve Tuple Tipleri

C# 7.0 sürümüyle beraber Tuple tiplerinde geliştirmeler yapıldı. Yukarıdaki metodu aşağıdaki gibi geliştirebiliriz. Ancak bu durumda geri dönen değer Tuple değil .NET Framework 4.7 ile birlikte gelen ValuTuple tipinden olmaktadır. ValueTuple sınıf değil yapı olarak tasarlandığını daha önce belirtmiştik.

```
    public static (int, string, string, string)
KullaniciGetir(int kullaniciId)
    {
        int id = 1;
        string isim = "Sefer";
        string soyisim = "Algan";
        string eposta = "sefer.algan@btakademi.com ";

        return (id, isim, soyisim, eposta); // Tuple Literal

       //ya da

      //ValueTuple<int, string, string, string> kullanici =
ValueTuple.Create<int, string, string, string>(1, "Sefer",
"Algan", "sefer.algan@btakademi.com");

        return kullanici;
    }
```

Metot bildiriminde geri dönüş elemanlarının tiplerini belirttiğimiz yeni yapıya Tuple Type denilmetedir. Metodun geri dönüş kısmında hangi elemanları döneceğimizi belirttiğimiz yere de Tuple Literal denilmektedir.

Bu kullanımda da yukarıdaki örnekteki gibi, Tuple elemanlarına ulaşırken Item1, Item2 vb. kullanmamız gerekiyor. C# 7.0'da yapılan güncelleme ile birlikte Tuple Type kısmında geri dönüş elemanlarına aşağıdaki gibi anlamlı isimler verebilmekteyiz.

```
    public static (int Id, string Isim, string Soyisim, string
Eposta) KullaniciGetir(int userId) // Tuple Type
    {
        int id = 1;
```



```
        string isim = "Sefer";
        string soyisim = "Algan";
        string eposta = "sefer.algan@btakademi.com ";

        return (id, isim, soyisim, eposta); // Tuple Literal
    }
```

Tuple Literal kısmında geri dönüş elemanları sırayla vermek yerine, Tuple Type kısmında tanımlanan elemanlara eşleme yolu ile de verilebilmektedir.

```
return (Id:id, Isim:isim, Soyisim: soyisim, Eposta: eposta);
// Tuple Literal
```

Deconstruct Özel Metodu ile Tuple İlişkisi

C# 7.0 versiyonuyla beraber sınıflarımız içerisinde bulunan herhangi bir bölüm veriyi özel isimli Deconstruct metodu ile dış dünyaya sunabilme seçeneği gelmiştir. Bu işlemi yapabilmek için sınıfı çerisinde Deconstruct metodu tanımlanır. İşlemin yapıldığı kod satırına ise Deconstructer denilmektedir. Deconstruct isimli metot derleyici tarafından tanınır ve ilgili sınıfa ait nesne bir Tuple değişkenine aktarıldığı anda otomatik olarak çalışır. Deconstruct metodunun geri dönüş değeri aşağıdaki örnekte görüleceği üzere void olmalıdır ve tanımlanan bütün parametreleri out ile işaretlenmelidir.

```
public class Kullanici
{
    public int Id { get; set; }
    public string Isim { get; set; }
    public string Soyisim { get; set; }
    public string Eposta { get; set; }

    public void Deconstruct(out string isim, out string soyisim)
    {
        isim = Isim;
        soyisim = Soyisim;
    }
}
```

Kullanici sınıfının Isim ve Soyisim özellikleri Deconstruct metodu ile tanımı yapılarak doğrudan ValueTuple değişkenine atanma özelliği kazandırılmıştır.

```
class Program
{
    static void Main(string[] args)
    {
        Kullanici kullanici = new Kullanici()
        {
            Id = 1,
```

```
            Isim = "Sefer",
            Soyisim = "Algan",
            Eposta = "sefer.algan@btakademi.com"
        };

        //deconstruct ifadesi
        (string isim, string soyisim) = kullanici;

        Console.WriteLine("Kullanıcı " + isim + " " + soyisim);
    }
}
```

Deconstruct metodunun kullanımı belli durumlarda önemli özelliklere erişimi standart hale getirebilir ve kod okunurluğunu arttırabilir.

Deconstruct özelliği ValueTuple tiplerinde de aşağıdaki gibi kullanılabilmektedir.

```
    public (int, string, string, string) KullaniciGetir(int
kullaniciId) // Tuple Type
    {
        // Veritabanı işlemleri

        int id = 1;
        string isim = "Sefer";
        string soyisim = "Algan";
        string eposta = "sefer.algan@btakademi.com ";

        return (id, isim, soyisim, eposta); // Tuple Literal
    }
```

KullaniciGetir metodunun Tuple Literal kısmında tanımlanan geri dönüş elemanlarını Deconstructer ifadesi yardımıyla aşağıdaki gibi kullanabiliriz.

```
(int id, string isim, string soyisim, string eposta) =
KullaniciGetir(2);
```

Önemli Not: Decontruct metodunun kullanılabilmesi için .NET Framework 4.7 'nin kurulması gerekmektedir. Çünkü ValueTuple isimli yapı bu sürümle birlikte gelmiştir.

Discard (_) İşareti ile Optimizasyon

C# 7.0 versiyonuyla birlikte bellek optimizasyonu amacıyla Discard özelliği eklenmiştir. Discard ilgilenmediğimiz değişkenlerin gereksiz yere bellekte yer kaplamaması amacıyla geliştirilmiş bir dil özelliğidir. Örnek verecek olursak; Aşağıdaki Personel sınıfında kişinin iletişim bilgilerinin geriye döndüren IletisimBilgisiGetir() metodun-

dan geriye ValueTuple olarak GSM, EvNo ve IsNo olmak üzere 3 adet bilgi dönmektedir. Eğer belli bir anda bizim sadece GSM'e ihtiyacımız var ise diğer iki değişkeni deyim yerindeyse görmezden gelmek için için "_" karakterini kullanıyoruz. Bu sayede ilgili faaliyet alanında EvNo ve IsNo için bellekte gereksiz yere alanlar ayrılmayacaktır.

```
class Personel
{
    private string IsNo, GSM, EvNo;
    private int Id;
    private string Ad, Soyad;

    public (string IsTel,string GSM,string EvTel)
IletisimBilgisiGetir()
    {
        return (IsNo,GSM,EvNo);
    }
}
Personel p = new Personel();
 (_,string gsmNo,_) = p.IletisimBilgisiGetir();
```

Discard karakterinin genel olarak kullanım yerleri aşağıdaki durumlarla sınırlıdır;

- Tuple ve Deconstructer Kullanımı (Yukarıda verilen örnek)
- Is operatöründe desen eşleştirmede değişkeni yok saymak için :

```
Örnek : if (obj is var _)
```

```
Out parametreli metot çağrımı : if (DateTime.TryParse("1.1.2017",
out _))
```

- Metot çağrımından dönen değerin önemsenmediği durumlar.

```
Örnek :  _ =geriyeIntDonenBirMetot();
```

Yukarıdaki durumlar dışında kullanılan "_" karakteri standart bir isim(değişken adı) olarak algılanacaktır.

14

Koleksiyonlar

- System.Collections İsim Alanı
- ArrayList Sınıfı
- Hashtable Sınıfı
- SortedList Sınıfı
- BitArray Sınıfı
- Stack Sınıfı
- Queue Sınıfı
- Şablon Tipli Koleksiyon Nesneleri
- Koleksiyonlar ile İlgili Diğer Konular
- Yeni Bir Iterasyon Tekniği (yield)

Çoğu zaman bir grup nesne ile çalışmamız gerekir. Bir grup nesneyi organize etmek ve yönetmek için çeşitli algoritmalar mevcuttur. Bu algoritmaları kullanarak nesne gruplarını yönetebiliriz. Ancak .NET Framework, System.Collections isim alanında koleksiyon diye isimlendirdiğimiz birtakım sınıflar sunar. System.Collections isim alanındaki en önemli sınıflar, ArrayList, SortedList, BitArray, Hashtable, Stack ve Queue sınıflarıdır. Bu bölümde her bir sınıfın altında yatan algoritmayı, bize ne gibi faydalar sunduğunu ve çeşitli özelliklerini inceleyeceğiz.

System.Collections İsim Alanı

Koleksiyon tabanlı sınıflar aynı türden ya da farklı türden birden fazla nesneyi bir arada tutup bu nesnelere farklı yollardan erişmemizi sağlarlar. Örneğin, diziler konusunda gördüğümüz Array sınıfı aynı türden nesneleri bir arada tutmak ve gerektiğinde indeksleyici operatörü ile Array nesnesinde tutulan dizi elemanlarına ulaşmak için kullanılır. Şu ana kadar gördüğümüz tek koleksiyon tabanlı sınıf Array sınıfıdır. Array sınıfı, C# dilinin temel sentaksı içinde bulunan dizi tanımlama ile uyumlu çalışır. Yani bir dizi tanımladığımızda aslında System.Array sınıfı türünden bir nesne tanımlamış oluyoruz. Bu durum temel diziler ile, Array sınıfının kullanışlı metotlarını ve özelliklerini kullanabilmemizi sağlar. Bu bölümde göreceğimiz koleksiyon tabanlı nesneler Array nesnelerinden farklıdır. Her şeyden önce bu koleksiyon nesnelerinin tanımlan-

ması için new anahtar sözcüğü koleksiyon ismi ile kullanılmalıdır. İkincisi bu nesneler Array sınıfındaki gibi System isim alanında bulunmazlar. Bu bölümde göreceğimiz bütün Koleksiyon sınıfları System.Collections isim alanında bulunur.

System.Collections isim alanında bulunan sınıflar C++ dilindeki konteyner sınıfları ile aynı işleri yapmaktadır. Bu yüzden C++ programcılarına, özellikle de STL denilen standart şablon kütüphanesini kullananlara bu bölümdeki koleksiyon tabanlı sınıflar yabancı gelmeyecektir.

Bir koleksiyon nesnesi içinde bulunan elemanlara foreach döngüsü ile erişebilmemiz mümkündür. Bu erişim, koleksiyon tabanlı sınıfların IEnumarable arayüzünü uygulamış olmasının bir sonucudur. IEnumarable arayüzündeki GetEnumerator() metodunun geri dönüş değeri olan IEnumarator arayüzü ile koleksiyondaki her bir elemana erişmek için metotlar ve özellikler tanımlıdır. Buradan her koleksiyon sınıfının IEnumerable arayüzünü uyguladığı sonucunu çıkarabiliriz. Hatırlarsanız arayüz kalıtımı konusunda IEnumerable arayüzünün nasıl uygulandığına bir örnek vermiştik. Ayrıca her koleksiyon tabanlı sınıf, kendi çalışma prensibine uygun diğer arayüzleri de uygulamış olabilir. Örneğin biraz sonra göreceğimiz ArrayList sınıfı ayrıca IList ve ICollection arayüzlerini de uygulamıştır.

System.Collections isim alanında birçok arayüz bulunmaktadır. Bu bölümde bu arayüzlerin nasıl uygulanacağı anlatılmamaktadır. Bu bölümde sadece bu arayüzleri uygulayan çeşitli sınıfları inceleyecek, bu arayüzlerin her birinin ne amaçla kullanılabileceği kısaca açıklayacağız.

Aşağıdaki listede System.Collections isim alanında bulunan arayüzler maddeler halinde listelenmiştir. Arayüzlerde bildirilen metotlar ve özellikler detaylı bir şekilde anlatılmamıştır. Sadece koleksiyon arayüzleri hakkında ön bilgi edinmeniz amaçlanmıştır. Çünkü bütün koleksiyon sınıfları bu arayüzlerden bir ya da birkaçını uygulamıştır. İleride göreceğimiz koleksiyon sınıflarındaki ortak özelliklerin sebebi, uygulanan arayüzlerin aynı olmasıdır.

1. ICollection: Bu arayüz bir kolleksiyonda olabilecek en temel metotları ve özellikleri içerir. System.Collections isim alanında bulunan bütün sınıflar bu arayüzü uygulamıştır. ICollection arayüzü, IEnumerable arayüzünü de kalıtım yolu ile elde etmiştir. Bu yüzden ICollection arayüzünü uygulayan sınıflar aynı zamanda IEnumerable arayüzünü de uygulamalıdır.

2. IComparer: Bu arayüz iki nesnenin ne şekilde karşılaştırılacağını uygulayan Compare() isimli metodu bulundurur. Bu metodun prototipi aşağıdaki gibidir.

```
int Compare(object o1, object o2);
```


Bu arayüzü uygulayan sınıfın nesnelerinin karşılaştırılma şekline göre Compare() metodunun geri dönüş değeri –1, 0 ve 1 olmalıdır. Karşılaştırma genellikle büyüklük ve küçüklüğe göre yapılır.

3. IDictionary: Bu arayüzü bir anahtar-değer ikilisi gibi çalışan koleksiyon sınıfları uygular. Bu konu içinde göreceğimiz HashTable sınıfını anlatırken anahtar ve değer ikili çiftinin ne demek olduğunu daha iyi anlayacaksınız. IDictionary arayüzü ICollection arayüzünden türemiştir. Bu yüzden IDictionary arayüzünü uygulayan bir sınıf ICollection arayüzündeki metot ve özellikleri de uygulamalıdır.

4. IDictionaryEnumerator: Bu arayüz ile anahtar değer ikili çiftlerden sadece değer, sadece anahtar ya da hem değer hem anahtar çiftlerini elde etmek için Value, Key ve Entry isimli 3 tane özellik bulunur.

5. IEnumerable: Bir koleksiyon tabanlı sınıfın elemanları arasında ilerlemek için kullanılan GetEnumerator() metodunu tanımlar.

6. IEnumerator: Daha önce de gördüğümüz gibi bir koleksiyon sınıfı için koleksiyondaki her bir elemana erişmek ve diğer işleri yapmak için uygulanması gereken arayüzdür. Bu metotlar MoveNext() ve Reset() metotlarıdır. Current ise IEnumerator arayüzündeki tek özelliktir. IEnumerable ve IEnumerator arayüzlerinin kalıtımına onbirinci bölümde örnek verilmişti.

7. IList: Bu arayüz bir koleksiyondaki elemanların her birine sıfır tabanlı bir indeks ile erişebilmek için çeşitli metotlar ve özellikler içerir. IList, ICollection arayüzünden türetilmiştir.

8. IHashProvider: Bu arayüz Object sınıfındaki GetHashCode() metodunu tanımlar. Bu metot ile HashTable sınıfı ile çalışabilmek için bir kod üretilir. Hash kodu denilen bu kodun ne anlama geldiğini HashTable sınıfını anlatırken daha iyi anlayacaksınız.

System.Collections isim alanındaki arayüzleri kısaca inceledikten sonra koleksiyon sınıflarını inceleyebiliriz. Bu bölümde temel olarak ArrayList, Hashtable, SortedList, BitArray, Stack ve Queue koleksiyon sınıfları anlatılacaktır.

ArrayList Sınıfı

ArrayList, klasik dizilerle çalışırken karşılaştığımız birçok sorunu ortadan kaldıran yeni bir koleksiyon sınıfıdır. Klasik dizilerle çalışırken dizinin boyutunun derleme aşamasında bilinme zorunluluğu vardı. ArrayList ile oluşturacağımız nesneler için böyle bir kısıtlama yoktur. Çalışma zamanında koleksiyona istediğimiz gibi yeni nesneler ekleyebilir ve yeni nesneler çıkarabiliriz. Klasik diziler aynı tür elemanları içeren koleksiyonlardır. Fakat ArrayList ile farklı türden nesneleri aynı koleksiyon içinde

saklayabiliriz. Farklı türden nesnelerin aynı koleksiyon içinde bulunması **boxing** ve **unboxing** işlemleri ile etkili bir biçimde kullanılabilir.

Klasik diziler ile çalışırken çoğu zaman diziyi ne kadar büyüklükte açacağımız belli olmaz. Bu tür durumlarda dizi yeteri kadar büyüklükte açılır. Açılan dizinin elemanları çoğu durumda kullanılmayan gereksiz bilgiler de olabilir. Bu durumda performans düşüklüğü meydana gelir. Halbuki ArrayList ile bir dizinin boyutunu çalışma zamanında belirleyebiliriz. Biz diziye eleman ekledikçe dizi otomatik olarak büyütülür. Klasik dizilerde olmayan diğer bir özellik ise diziden eleman çıkararak dizinin boyutunu küçültmektir. Normal dizilerden farklı olarak ArrayList nesnelerinden çalışma zamanında bir eleman da çıkarabiliriz.

Bir ArrayList sınıfının kapasitesi artırılacağı zaman kapasite iki katına çıkarılır. Böylece her eleman eklendiğinde kapasite artırma gibi ek bir işlem yapılmaz.

ArrayList sınıfı, ICollection, IEnumerable, IList ve ICloneable arayüzlerini uygulamaktadır. ArrayList sınıfının 3 tane aşırı yüklenmiş yapıcı metodu vardır. Bu metotlar aşağıda belirtilmiştir.

```
ArrayList alist = new ArrayList();
```

Bu yapıcı metot ile varsayılan kapasitede bir ArrayList nesnesi oluşturulur. Bu varsayılan kapasite 16 elemanı içerecek şekildedir. Çalışma zamanında bu kapasite aşılırsa dizinin kapasitesi otomatik olarak artırılır.

```
ArrayList alist = new ArrayList(50);
```

Bu yapıcı metot ile parametre ile verilen kapasitede bir ArrayList dizisi oluşturulur. Birinci yapıcı metotta olduğu gibi diziye eleman eklendikçe eğer kapasite aşılmışsa, kapasite artırılır.

```
ArrayList alist = new ArrayList(ICollection);
```

Bu metot ile ICollection arayüzünü uygulamış herhangi bir sınıf referansı ile ArrayList dizisi oluşturulur.

Bir ArrayList'e eklenen elemanlar object türüne aktarılarak saklanır. Bu da her türlü sınıf nesnesinin ArrayList dizilerine eklenebileceğini gösterir, ancak dizideki elemanlara tekrar ulaşabilmek için object nesnelerinden orijinal sınıf türüne geri dönmemiz gerekir. Bunun için object referanslarına unboxing işlemi uygulanmalıdır.

ArrayList dizilerine yeni eleman eklemek için Add() metodu kullanılır. Bu metot parametre olarak bir object referansı almaktadır. Aynı şekilde bir ArrayList dizisinden bir elemanı çıkarmak için `Remove()` metodu kullanılır. Parametre olarak verilen object referansı, diziden çıkarılacak elemanı temsil etmektedir. Aşağıda bir ArrayList dizisinin bildirimi, diziye yeni eleman ekleme ve diziden eleman çıkarma işlemlerinin nasıl yapıldığı gösterilmektedir.

Bir dizinin herhangi bir andaki kapasitesini öğrenmek için ArrayList sınıfının Capacity özelliği kullanılabilir.

```
using System;
using System.Collections;

class Koleksiyon
{
      public static void Main()
      {
          ArrayList AList = new ArrayList();

          Console.WriteLine("Varsayılan kapasite: " +
                                          AList.Capacity);

          AList.Add(6);
          AList.Add("CSharp");
          AList.Add('A');
          AList.Add(236.58f);
          AList.Add(true);

          EkranaYaz(AList);

          AList.Remove('A');
          AList.Remove(true);
          AList.Add("Deneme");

          EkranaYaz(AList);
      }

      public static void EkranaYaz(ArrayList al)
      {
          foreach(object i in al)
              Console.Write(i.ToString() + " ");
          Console.WriteLine();
      }
}
```

Yukarıdaki programın ekran çıktısı aşağıdaki gibidir.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum13\kod>ArrayList
Varsayilan kapasite: 16
6  CSharp  A  236,58  True
6  CSharp  236,58  Deneme

H:\cskitap\kitap\Bolum13\kod>
```

Görüldüğü gibi bir ArrayList dizisine her türden nesneler eklenebilmektedir. Bu da ArrayList dizilerinin en önemli özelliğidir. Burada dikkat edilmesi gereken nokta diziye eklenen her elemanın boxing işlemi ile object referanslarına dönüştürülmesidir. Bu yüzden dizinin elemanlarına erişmek için boxing işleminin tersi olan unboxing işlemini uygulamamız gerekir. Farklı türden nesneleri ArrayList dizilerine eklediğimizde foreach döngüsü yardımıyla unboxing işlemini açıkça yapamayız. Çünkü her eleman için ancak boxing işleminin yapıldığı türe ilişkin unboxing işlemi gerçekleşir. Örneğin, aşağıdaki ArrayList sınıfının elemanlarına ortak bir dönüşüm yapılamaz.

```
using System;
using System.Collections;

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

class Koleksiyon
{
      public static void Main()
      {
          ArrayList AList = new ArrayList();
          AList.Add("Sefer");
          AList.Add("Algan");
          AList.Add("Gökçen");
          AList.Add("Yıldırım");
          Deneme d = new Deneme(5);
          AList.Add(d);

          foreach(object i in AList)
              Console.WriteLine((string)i);
      }
}
```


Bu programı derlediğimizde InvalidCastException istisnai durumunun oluştuğunu görürüz. Bunun sebebi dizinin son elemanının bizim bildirdiğimiz Deneme türünden olmasıdır yani Deneme türünün string türüne dönüştürülememesidir. Eğer açıkça elemanları string türüne dönüştürmemiş olsaydık bu sefer dizinin son elemanı Deneme türünden olmasına rağmen ekrana sınıfın ismi olan "Deneme" yazacaktı.

ArrayList sınıfı ile farklı türden verileri organize edebilmemize rağmen açıkça bu amaç için çok nadir kullanılır. Birbirleriyle bağlantılı olan temel veya türemiş tür referanslarını bu tür dizilerde saklamak daha mantıklı olacaktır. Birbirleriyle çok yakından ilişkisi olmayan türden nesneleri aynı çatı altında toplamak algoritmik açıdan çok sık karşılaşacağımız bir durum değildir.

ArrayList sınıfı ile ilgili ilginç bir nokta da, dizinin kapasitesinin artırılması ile ilgilidir. Daha önce de denildiği gibi dizinin kapasitesi artırılacağı zaman mevcut kapasite iki katına çıkarılır.

Bu bilgilerle aşağıdaki programın ekran çıktısını tahmin etmeye çalışın.

```
using System;
using System.Collections;

class Koleksiyon
{
      public static void Main()
      {
          ArrayList AList = new ArrayList(2);
          AList.Add(1);
          AList.Add(2);
          AList.Add(3);
          AList.Add(4);
          AList.Add(5);

          Console.WriteLine(AList.Capacity);
      }
}
```

Cevabınız 5 ise yanlış!. Çünkü Add(3) metodu ile dizinin kapasitesi 2'den 4'e çıkarılıyor. Add(5) ifadesi ile de 4'ten 8'e çıkarılıyor. Remove() metodu ile bir elemanı diziden çıkarmak dizinin kapasitesini değiştirmez. Ayrıca Remove() ile diziden çıkarılmak istenen eleman eğer dizide mevcut değilse hiçbir iş yapılmadan işlem devam eder.

ArrayList sınıfı klasik dizilerle gerçekleştirilmesi zor olan birtakım işler için çok uygundur. Örneğin, bir dizinin herhangi bir yerine bir eleman eklemek için prototipi

```
void Insert(int indeks, object deger)
```

olan metot kullanılabilir. Bu metot ile dizinin indeks nolu elemanına ikinci parametre ile verilen değer yerleştirilir. Dolayısıyla dizi indeks elemanından sonra sona doğru bir eleman ötelenir. Add() metodunun, eklenecek elemanları dizinin sonuna ekledi-ğini de hatırlatmak gerekir.

Sabit kapasiteli bir ArrayList dizisi de oluşturmak mümkündür. Bunun için ArrayList sınıfının statik metodu olan FixedSize() kullanılır. Parametre olarak verilen alist sınıfının elemanlarını kapsayan bir ArrayList referansı döndürür. Kullanımı aşağıdaki gibidir.

```
ArrayList alist = ArrayList.FixedSize(ArrayList alist);
```

Kapasitesi sabit olan diziler klasik diziler gibi davranır. Dizinin indeksi aşıldığı zaman ArgumentOutOfRangeException hatası fırlatılır.

ArrayList sınıfının elemanlarına klasik dizilerde olduğu gibi indeksleyici aracılığıyla da erişebiliriz.

Şimdi isterseniz ArrayList sınıfının diğer kullanışlı metot ve özelliklerini inceleyelim.

ArrayList sınıfının önemli özellikleri aşağıdaki gibidir.

1. Count: ArrayList koleksiyonundaki toplam eleman sayısını int türü olarak geri döndürür. Count özelliğinin Capacity özelliğinden farklı olduğunu unutmayın. Bu özelliğin sadece get bloğu bildirilmiştir.

2. IsFixedSize: ArrayList dizisinin sabit kapasiteli olup olmadığı bilgisini bool olarak geri döndürür. Eğer kapasite sabitse true değeri geri döner.

3. IsReadOnly: Bu özellik ile bir ArrayList dizisinin sadece okunabilir olup olmadığı kontrol edilebilir. Sadece okunabilir ArrayList dizilerini elde etmek için ArrayList sınıfının statik ReadOnly() metodu kullanılır. Bu metodun kullanımı statik FixedSize() metodunun kullanımı ile aynıdır.

ArrayList sınıfının IsSynchronized ve SyncRoot özellikleri bu kitabın konularıyla alakalı olmadığı için ele alınmamıştır.

ArrayList sınıfının önemli metotları ise aşağıdaki gibidir.

1. AddRange(ICollection ic): Bu metot ile ic arayüz referansı ile tutulan koleksiyon nesnesinin elemanları, metodu çağıran ArrayList nesnesinin sonuna eklenir. Bu me-

tot ile ICollection arayüzünü uygulamış olan bütün koleksiyon sınıf nesneleri kullanılabilir. Metodun parametresinin ICollection arayüzü türünden olmasının sebebi de budur.

2. Sort(): Bu metot ile ArrayList dizisinin elemanları IComparer arayüzü ile belirlenen kurala göre sıralanır. Sıralama algoritması olarak QuickSort kullanılır. Sıralanacak dizi sadece okunabilir ise NotSupportedException istisnai durum sınıfı fırlatılır. Sort() metodunun 3 tane aşırı yüklenmiş versiyonu bulunmaktadır. Bunlar aşağıdaki gibidir.

3. void Sort(): Bu metot dizin elemanlarını varsayılan IComparer arayüzünde belirtilen kurala göre sıralar. Bu metot daha çok temel veri türlerini içeren dizileri sıralamak için kullanılır.

4. void Sort(IComparer ic): Bu metot ile belirlenen IComparer karşılaştırma yöntemine göre sıralama işlemi yapılır.

5. void Sort(int indeks, int adet, IComparer ic): Bir önceki metot ile aynı işlemi yapar, ancak sıralama işlemi dizinin tamamında değil de indeks elemanından itibaren adet kadar eleman üzerinde yapılır.

Sort() metodunun etkisini görmek için string türden elemanlar içeren bir ArrayList dizisini sıraladıktan sonra ekrana yazdırmayı deneyin.

6. int BinarySearch(object o): System.Array sınıfında olduğu gibi sıralanmış diziler içinde parametre olarak verilen nesne aranır. Eğer aranan eleman bulunursa elemanın indeks numarası ile geri döner. Eğer aranan eleman bulunamaz ise negatif bir değere geri döner. Bu metodun aşağıdaki gibi iki versiyonu daha vardır.

a. int BinarySearch(object o, IComparer ic): Arama yaparken o nesnesi ile dizinin elemanlarının karşılaştırılmasının yapılacağı kural olarak ic arayüzü referansı alınır. Bu metot daha çok programcının tanımladığı türden eleman içeren koleksiyonları sıralamak için kullanılır. Bu bölümün sonunda IComparer arayüzüne benzeyen IComparable arayüzünü uygulayan bir sınıf bildirimine örnek verilecektir.

b. int BinarySearch(int indeks, int adet, object o, IComparer ic): Arama işlemi bir önceki metot ile aynıdır. Ancak arama işlemi dizinin indeks elemanından itibaren adet sayıda eleman içinde yapılır.

7. void Reverse(): Bu metodu çağıran dizinin elemanları ters çevrilir. Yani sonuncu eleman birinci eleman, birinci eleman da sonuncu eleman olacak şekilde değiştirilir. Reverse() metodunun ikinci kullanım şekli de aşağıdaki gibidir.

8. void Reverse(int indeks, int adet): Ters çevirme işlemi sadece indeks elemanından itibaren adet kadar eleman üzerinde gerçekleşir.

9. void Clear(): Bu metodu çağıran ArrayList dizisinin bütün elemanları diziden çıkarılır. Dolayısıyla dizinin Count özelliği sıfır olur. Clear() metodunun dizinin kapasitesine herhangi bir etkisi yoktur.

10. void TrimToSize(): ArrayList dizisinde kapasitenin eleman sayısından fazla olduğu durumda kapasiteyi eleman sayısına eşitlemek için kullanılır. Böylece kullanılmayacak fazla bellek alanları boş yere tutulmamış olur. TrimToSize() metodu boş diziler ile kullanıldığı zaman kapasite sıfırlanmaz, kapasite varsayılan değer olan 16 olur.

11. bool Contains(object o): Parametre ile verilen nesnenin dizi elemanları içerisinde bulunup bulunmadığı bilgisi true ya da false olarak geri döndürülür. Eğer aranan eleman varsa true yoksa false değerine geri döner.

12. void CopyTo(): Bu metot ile ArrayList sınıfının bütün elemanları klasik dizilere kopyalanır. Kopyalamanın yapılacağı Array dizisinin ArrayList dizisindeki elemanlara tür bakımından uyumlu olması gerekir. Bu metodun 3 kullanım şekli vardır. Bunlar aşağıdaki gibidir.

13. void CopyTo(Array a): Bu metodu çağıran ArrayList dizisinin bütün elemanları tek boyutlu a dizisine kopyalanır. Kopyalama işlemi a dizisinin ilk elemanından itibaren yapılır.

14. void CopyTo(Array a, int indeks): Birinci metot ile aynı işlemi yapar, tek değişen a dizisine kopyalama işleminin, a dizisinin indeks nolu elemanından itibaren olmasıdır.

15. void CopyTo(int indeks, Array a,int dizi_indeks, int adet): Bu metodu çağıran ArrayList nesnesinin indeks elemanından itibaren adet kadar elemanı a dizisinin dizi_indeks elemanından itibaren a ile belirtilen diziye kopyalanır.

16. ArrayList GetRange(int indeks, int adet): Bu metodu çağıran dizinin indeks elemanından itibaren adet kadar elemanı yeni bir ArrayList nesnesi olarak geri döndürülür.

17. int IndexOf(object o): Parametre ile belirtilen nesnenin ilk olarak bulunduğu elemanın indeks numarasını geri döndürür. Eğer eleman bulunmaz ise -1 değerini döndürür.

18. object[] ToArray(): Bu metot ArrayList dizisinin elemanlarını yeni bir object türünden diziye aktarıp bu dizinin referansı ile geri döner. ToArray() metodunun diğer bir versiyonu ise aşağıdaki gibidir.

19. Array ToArray(Type t): Bu metot ile parametre olarak belirlenen t türünden bir diziye kopyalama işlemi yapılır ve kopyalamanın yapıldığı dizinin referansı ile geri dönülür.

20. void RemoveAt(int indeks): Bu metot ArrayList dizisindeki indeks nolu elemanı diziden çıkarır. Eğer belirtilen indeks 0'dan küçük ya da ArrayList'in Count özelliğinden büyük ise ArgumentOutOfRangeException istisnai durumu meydana gelir.

ArrayList sınıfının bu metotları ile çeşitli örnekler yaparak her bir metodun etkisini daha iyi görebilirsiniz. Burada bu metotların her biri ile ilgili tek tek örnek vermek mümkün değildir. Zaten büyük çaplı sınıf kütüphaneleri ile çalışırken önemli olan neyin nerede olduğunu bilmek ve olanaklardan haberdar olmaktır. Bu kitabın amacı sizi bu olanaklardan haberdar etmektir. Siz ihtiyaçlarınıza göre bu metotları dilediğiniz zaman kullanabilirsiniz. Bu metotları kullanırken en sık başvuracağınız kaynaklardan birinin de MSDN dokümanlarının olduğunu unutmayın.

Hashtable Sınıfı

ArrayList ve Array sınıflarında elemanlara erişmek için bir indeks numarası kullanıyorduk. Ancak bazen indeks numarası yerine bir koleksiyondaki elemanlara bir anahtar değer ile erişmek isteyebiliriz. Bu durumu simüle eden sınıf Sytem.Collections isim alanında bulunan Hashtable sınıfıdır. HashTable sınıfında koleksiyonlar bir anahtar (key)-deger (value) ikilisi olarak saklanır. Yani belirlenen anahtarlara ilişkin çeşitli değerleri tutmak için Hashtable sınıfını kullanabiliriz. Hashtable ile anahtar ve değer ikilisi hash tablosu denilen bir tabloda saklanmaktadır. Her bir ikiliyi temsil eden bir hash kodu vardır. Bu hash kodu bir anahtar değer ikilisinin tek olmasını sağlamaktadır. Biz bir hash tablosundaki bir anahtara karşılık gelen değeri elde etmek istediğimizde bu hash kodu kullanılır. Dolayısıyla bir anahtara karşılık gelen değeri elde etmek için maksimum performans sağlanır. Hash tablosu sayesinde çok sayıda anahtar değer ikilisi içeren koleksiyonlarda arama ve anahtar değerlerinin karşılığını elde etme gibi işlemler daha kısa sürede gerçekleştirilir.

Bir Hashtable nesnesinin yapısını yandaki gibi gösterebiliriz.

Anahtar1	Deger1
Anahtar2	Deger2
Anahtar3	Deger3
Anahtar4	Deger4
Anahtar5	Deger5

Yandaki şekilde de görüldüğü üzere her anahtarın bir değeri vardır. Hashtable sınıfında anahtarlar Key, değerler de Value olarak adlandırılır.

Bir Hashtable koleksiyonu oluşturduktan sonra koleksiyona bir anahtar-değer çifti eklemek için ya Hashtable sınıfını Add() metodunu ya da Hashtable sınıfındaki indeksleyiciyi kullanırız. Buna göre bir Hashtable nesnesine yeni bir anahtar-değer çifti aşağıdaki gibi eklenebilir.

```
Hashtable hashtable = new Hashtable();

hashtable.Add("Anahtar1", "Deger1");

hashtable[ "Anahtar2"] = "Deger2";
```

Add metodunun iki parametresi vardır. Bu parametrelerden ilki anahtarı, ikincisi ise değeri temsil etmektedir. Bu iki parametre de object türündendir. Yani her türden nesneyi anahtar ya da değer olacak şekilde düzenleyebiliriz. Aynı şekilde indeksleyicideki indeks değeri de object türündendir. İndeksleyiciyi kullanarak bir anahtara ilişkin değer değiştirilebilir. Örneğin Anahtar2'ye ilişkin değeri değiştirmek için

```
hashtable[ "Anahtar2"] = "Yeni Değer";
```

yazılabilir.

Add() metodu ile daha önceden eklenmiş bir anahtar eklenmeye çalışılırsa ArgumentException istisnai durumu oluşur.

Hashtable sınıfının Count özelliği ile bir koleksiyondaki toplam anahtar-değer çifti sayısını elde edebiliriz.

Bir Hashtable nesnesindeki anahtar ve değerlerin tamamına erişmek için ICollections arayüzünü kullanabiliriz. Hashtable sınıfının Keys özelliği ICollections arayüzü referansı ile Hashtable nesnesi içindeki anahtarları bir koleksiyon şeklinde döndürür. Aynı şekilde Values özelliği ICollections arayüzü referansı ile Hashtable nesnesi içindeki anahtar değerlerini bir koleksiyon şeklinde döndürür. ICollection referansı üzerinden bu anahtar-değer çiftlerine foreach döngüsü yardımı ile tek tek erişebiliriz. Örneğin aşağıdaki programda Hashtable nesnesine eklediğimiz anahtar-değer çiftleri daha sonra tek tek elde edilip ekrana yazdırılıyor.

```
using System;
using System.Collections;

class hashtable
{
    public static void Main()
    {
        Hashtable Sozluk = new Hashtable();

        Sozluk.Add("Araba", "Car");
        Sozluk.Add("Masa", "Table");
        Sozluk.Add("Kalem", "Pencil");
        Sozluk[ "Kitap"] = "Book";
```



```
        Sozluk[ "Bilgisayar"] = "Computer";

        ICollection anahtarlar = Sozluk.Keys;

        foreach(string anahtar in anahtarlar)
            Console.WriteLine(anahtar + " = " +
                                        Sozluk[ anahtar] );

        Console.WriteLine("Toplam kelime: " + Sozluk.Count);
    }
}
```

Yukarıdaki programda basit bir sözlükteki karşılıklı kelimeleri tutmak için Hashtable sınıfını nasıl kullanabileceğimize bir ipucu verilmiştir. Bu programda Hashtable sınıfının Keys özelliğinin ICollection arayüzü referansına nasıl aktarıldığına özellikle dikkat edin.

Programı derleyip çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum13\kod>Hashtable
Araba = Car
Bilgisayar = Computer
Masa = Table
Kalem = Pencil
Kitap = Book
Toplam kelime: 5

H:\cskitap\kitap\Bolum13\kod>
```

Görüldüğü gibi Hashtable nesnelerinin dizilerden en büyük farkı koleksiyondaki elemanlara bir indeks ile değil de bir anahtar ile erişebilmemizdir.

Bir Hashtable nesnesindeki her anahtar-değer çifti arasında dolaşmak için IDictionaryEnumerator arayüzünü de kullanabiliriz. Bu arayüzün Key ve Value özellikleri ile anahtar-değer çiftlerine erişebiliriz. Ayrıca IDictionaryEnumerator arayüzündeki MoveNext() metodu ile anahtar-değer çiftlerinin içinde tek tek ilerleyebiliriz. Aşağıdaki programda bir Hashtable koleksiyonunun elemanlarına IDictionaryEnumerator arayüzü ile nasıl ulaşıldığı gösterilmektedir.

Hashtable nesnesi için bir numaralı koleksiyon elde etmek için GetEnumerator() metodu kullanılmaktadır.

```
using System;
using System.Collections;

class hashtable
{
      public static void Main()
      {
          Hashtable Sozluk = new Hashtable();

          Sozluk.Add("Araba", "Car");
          Sozluk.Add("Masa", "Table");
          Sozluk.Add("Kalem", "Pencil");
          Sozluk["Kitap"] = "Book";
          Sozluk["Bilgisayar"] = "Computer";

          IDictionaryEnumerator ide = Sozluk.GetEnumerator();

          while(ide.MoveNext())
              Console.WriteLine(ide.Key + " = " + ide.Value);

          Console.WriteLine("Toplam kelime: " + Sozluk.Count);
      }
}
```

Bu programın çıktısı bir önceki programın çıktısı ile aynı olacaktır.

Bir Hashtable koleksiyonundan bir anahtar-değer çiftini silmek için Remove() metodu kullanılır. Remove() metodu ile belirtilen anahtar değer eğer null referans ise ArgumentNullException istisnai durumu oluşur. Eğer belirtilen anahtar sadece okunabilir ya da Hashtable sınıfı sabit kapasiteli ise NotSupportedException istisnai durumu oluşur. Örneğin bir önceki programda Kitap anahtarına ilişkin anahtar-değer çiftini silmek için

```
Sozluk.Remove("Kitap");
```

deyimi yazılır.

IsReadOnly ve IsFixed özellikleri, ArrayList sınıfında olduğu gibi koleksiyonun sadece okunabilir ve sabit kapasiteli olup olmadığı bilgisini bool türünden bir değer olarak geri döndürür.

Şimdiye kadar değindiğimiz metotların yanı sıra Hashtable sınıfının birçok metodu vardır. Bu metotlardan sık kullanılanları aşağıda verilmiştir.

void Clear(): Hashtable koleksiyonundaki bütün anahtar-değer çiftlerini siler. Dolayısıyla Count özelliği 0 olur.

bool ContainsKey(object anahtar): Bu metot parametre ile verilen anahtarın Hashtable nesnesi içinde bulunup bulunmadığını kontrol eder. Eğer anahtar değeri varsa true değerine geri döner. Ayrıca Contains() metodu da ContainsKey() metodu ile aynı işleve sahiptir.

bool ContainsValue(object deger): Bu metot parametre ile verilen değerin Hashtable nesnesi içinde bulunup bulunmadığını kontrol eder. Eğer değer varsa true değerine geri döner.

void CopyTo(Array dizi, int indeks): Hashtable içindeki anahtar-değer çiftlerini bir boyutlu Array dizisine indeks elemanından itibaren kopyalar. Kopyalama işleminin yapıldığı dizi DictionaryEntry yapısı türündendir. DictionaryEntry yapısı ile bir anahtar-değer ikilisi temsil edilir. DictionaryEntry yapısındaki Key ve Value özellikleri ile anahtar ve değerlere erişilebilir. Sadece Hashtable nesnesindeki anahtarları bir diziye kopyalamak için,

```
hashtable.Keys.CopyTo(dizi, indeks)
```

metodu, aynı şekilde sadece değerleri kopyalamak için

```
hashtable.Values.CopyTo(dizi, indeks)
```

metodunu kullanabilirsiniz.

SortedList Sınıfı

SortedList, HashTable ve ArrayList yapılarının karışımı gibidir. Hem anahtar-değer ikilileri kullanılıyor hem de bu anahtar-değer çiftine bir indeks numarasına göre erişebiliyoruz. SortedList yapısının en belirgin özelliği ise adından da anlaşılacağı üzere sıralı olmasıdır. Sırlama işlemi temel veri türleri için büyüklük ve küçüklüğe göredir. Ancak kendi karşılaştırma modelimizi de IComparer arayüzünü kullanarak oluşturabiliriz. Genel olarak bir SortedList koleksiyon nesnesini aşağıdaki yapıcı metotlardan biriyle oluşturabiliriz.

```
SortedList slist = new SortedList ();
```

Bu yapıcı metot ile varsayılan kapasitede bir SortedList nesnesi oluşturulur. Bu varsayılan kapasite 16 elemanı içerecek şekildedir. Çalışma zamanında bu kapasite aşılırsa koleksiyonun kapasitesi otomatik olarak artırılır.

```
SortedList slist = new SortedList(50);
```

Bu yapıcı metot ile, parametre ile verilen kapasitede bir SortedList nesnesi oluşturulur. Birinci yapıcı metotta olduğu gibi diziye anahtar-değer ikilisi eklendikçe kapasite artırılır.

```
SortedList slist = new SortedList(IDictionary id);
```

Bu metot ile IDictionary türünden olan id koleksiyonunun elemanları ile yeni bir SortedList koleksiyonu oluşturulur.

SortedList yapısında sıralı olan eleman anahtarlardır.

SortedList sınıfında, Hashtable sınıfında bulunan Add(), Clear(), Contains(), ContainsKey(), ContainsValue(), GetEnumerator() ve Remove() metotları bulunur.

Bu metotların yanısıra SortedList sınıfında aşağıdaki metotlar da bildirilmiştir.

object GetByIndex(int indeks): Bu metot SortedList koleksiyonunda bulunan indeks numaralı elemanı geri döndürür.

object GetKey(int indeks): Bu metot ile sıralanmış SortedList koleksiyonundaki indeks nolu anahtar-değer çiftinden anahtar, object nesnesi olarak geri döndürülür.

IList GetKeyList(): Bu metot ile sıralanmış SortedList koleksiyonundaki anahtarlar IList arayüzü referansı ile geri döndürlür.

IList GetValueList(): Bu metot ile sıralanmış SortedList koleksiyonundaki değerler IList arayüzü referansı ile geri döndürülür.

int IndexOfKey(object anahtar): Parametre ile belirtilen anahtarın SortedList koleksiyonunda kaç numaralı elemana denk geldiğine geri döndürür. Eğer anahtar bulunamazsa –1 değerine geri döner.

int IndexOfValue(object deger): Parametre ile belirtilen anahtar değerinin SortedList koleksiyonunda kaç numaralı elemana denk geldiğine geri döndürür. Eğer anahtar değeri bulunamazsa –1 değerine geri döner.

void SetByIndex(int indeks, object değer): Bu metot indeks ile belirtilen sıradaki anahtarın değerini ikinci parametre ile verilen değere atar. Eğer yeni atanan değer sırayı değiştiriyorsa SortedList nesnesi kendi içinde yeniden düzenlenir. SortedList nesnesine her eleman eklendiğinde ya da her eleman değişikliği yapıldığında SortedList kendi içindeki sırayı yeniden düzenler.

void TrimToSize(): SortedList koleksiyonuna artık eleman eklenmeyecekse bu metot kullanılır. Metodun kapasitesi metottaki eleman sayısına eşitlenir.

SortedList sınıfının Capacity, Count, IsFixed, IsReadOnly, Keys ve Values isimli özellikleri de mevcuttur. Keys ve Values özellikleri ile Hashtable sınıfında olduğu gibi ICollection referansı elde edilir. Bu özelliklerin bir kısmını Hashtable bir kısmını da ArrayList sınıfını anlatırken incelemiştik. Bu ortak özelliklerin çoğu koleksiyon sınıflarının uyguladıkları arayüzlerden kaynaklanmaktadır.

SortedList koleksiyonunun elemanlarına ulaşmak için Keys ve Values özellikleri kullanılabileceği gibi DictionaryEntry yapısı da kullanılabilir. Bu yapı her bir anahtar-değer ikilisini temsil etmek için kullanılır.

Aşağıdaki örnekte DictionaryEntry yapısının ve SortedList koleksiyon sınıfının çeşitli metotlarının kullanımına örnekler verilmiştir.

```
using System;
using System.Collections;

class sortedlist
{
      public static void Main()
      {
          string[ ] Kelimeler = { "Araba","Masa","Defter",
                                  "Kitap","Okul" } ;

          SortedList Sozluk = new SortedList();

          Sozluk.Add("Araba", "Car");
          Sozluk.Add("Masa", "Table");
          Sozluk.Add("Kalem", "Pencil");
          Sozluk[ "Kitap"] = "Book";
          Sozluk[ "Bilgisayar"] = "Computer";

          EkranaYaz(Sozluk);

          for(int i=0; i < Kelimeler.Length; ++i)
              if(Sozluk.ContainsKey(Kelimeler[ i] ))
                    Console.WriteLine(Kelimeler[ i] + "=" +
                                          Sozluk[ Kelimeler[ i]] );
              else
                    Console.WriteLine(Kelimeler[ i] + ",
                                        sözlükte bulunamadı.");
```

```
        Console.WriteLine();

        Sozluk.Add("Doğru","Line");

        EkranaYaz(Sozluk);

        Console.WriteLine();
        Console.WriteLine("Sıralı listede toplam {0} eleman
                            bulunmaktadır.",Sozluk.Count);
        Console.Write("Bu elemanlar: ");

        ICollection anahtarlar = Sozluk.Keys;

        foreach(string str in anahtarlar)
            Console.Write(str + "-");
    }

    public static void EkranaYaz(SortedList slist)
    {
        foreach(DictionaryEntry de in slist)
            Console.WriteLine("{0,-12} :{1,- 12}"
                                ,de.Key,de.Value);

        Console.WriteLine();
    }
}
```

Programı derleyip çalıştırdığınızda aşağıdaki ekran görüntüsünü elde ederiz.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum13\kod>SortedList
Araba        :Car
Bilgisayar   :Computer
Kalem        :Pencil
Kitap        :Book
Masa         :Table

Araba=Car
Masa=Table
Defter, sözlükte bulunamadi.
Kitap=Book
Okul, sözlükte bulunamadi.

Araba        :Car
Bilgisayar   :Computer
Dogru        :Line
Kalem        :Pencil
Kitap        :Book
Masa         :Table


Sirali listede toplam 6 eleman bulunmaktadir.
Bu elemanlar: Araba-Bilgisayar-Dogru-Kalem-Kitap-Masa-
H:\cskitap\kitap\Bolum13\kod>
```


Ekran görüntüsünden de görüldüğü gibi slist koleksiyonuna eklenen "Doğru-Line" ikilisi koleksiyonun en sonuna eklenmemiştir. Bunun yerine anahtarlar alfabetik sıraya göre tekrar düzenlenmiştir.

Contains() metodu kullanılarak bir string dizisinin slist koleksiyonundaki anahtarlar ile nasıl karşılaştırıldığına özellikle de

```
Sozluk[ Kelimeler[ i] ]
```

ifadesinin kullanımına dikkat edin.

BitArray Sınıfı

BitArray, bitlerle işlem yapmak için özelleştirilmiş bir koleksiyon sınıfıdır. BitArray nesnelerinde 1 değerini ifade eden true, 0 değerini ifade eden false değerleri saklanır. BitArray ile bir sayının bitlerine tek tek erişebileceğimiz gibi bitsel AND ve XOR gibi bitsel işlemler de gerçekleştirebiliriz. BitArray ile ilgili örneklere geçmeden önce bir BitArray koleksiyonunu nasıl oluşturabileceğimizi inceleyelim. BitArray sınıfının kullanabileceğimiz 6 tane yapıcı metodu vardır. Bu metotlar aşağıda sıralanmıştır.

1. BitArray(BitArray bArray): Bu yapıcı metot ile bArray koleksiyonunun elemanları kullanılarak yeni bir BitArray koleksiyon nesnesi oluşturulur.

2. BitArray(bool[] bDizi): Bu yapıcı metot ile bool türden dizi olan bDizi'nin elemanları kullanılarak yeni bir koleksiyon nesnesi oluşturulur. Bu durumda bDizi dizisinin elemanları ile oluşturulan BitArray koleksiyonunun elemanları aynı olur.

3. BitArray(byte[] bDizi): Bu yapıcı metot ile byte türünden dizi olan bDizi'nin elemanları kullanılarak yeni bir koleksiyon nesnesi oluşturulur. Bu durumda oluşturulan koleksiyon nesnesinin eleman sayısı

```
8*bDizi.Length
```

olacaktır. Bunun sebebi byte türünün 8 byte büyüklüğünde olmasıdır. Yani bDizi'sinin ilk elemanı koleksiyondaki ilk 8 elemanı temsil edecektir. Bitleri 1 olanlar true 0 olanlar ise false olacak şekilde düzenlenir.

4. BitArray(int[] bDizi): Bu yapıcı metot ile int türünden dizi olan bDizi'nin elemanları kullanılarak yeni bir koleksiyon nesnesi oluşturulur. Bu durumda oluşturulan koleksiyon nesnesinin eleman sayısı

```
32*bDizi.Length
```

olacaktır. Bunun sebebi ise int türünün 32 byte büyüklüğünde olmasıdır. Yani bDizi'sinin ilk elemanı koleksiyondaki ilk 32 elemanı temsil edecektir. Bitleri 1 olanlar true, 0 olanlar ise false olacak şekilde düzenlenir.

5. BitArray(int Boyut): Bu yapıcı metot ile Boyut kadar false değerinde elemanı olan bir BitArray koleksiyonu oluşturulur.

6. BitArray(int Boyut, bool Deger): Bu yapıcı metot ile Boyut kadar **Deger** değerinde elemanı olan bir BitArray koleksiyonu oluşturulur. Genellikle bütün elemanları true olacak bir BitArray koleksiyonu oluşturmak için kullanılır.

BitArray sınıfı, ICollection ve IEnumerable arayüzlerini uyguladığı için koleksiyonların elemanlarına erişmek için şu ana kadar kullandığımız yöntemler BitArray sınıfı için de geçerlidir. BitArray koleksiyonunun elemanları bir biti temsil eder. Bu bit değeri true ya da false olacak şekildedir. İndeksleyici ile BitArray koleksiyonunun elemanlarına ulaştığımızda bool türünden bir değer elde etmiş oluruz.

Aşağıdaki örnekte bir BitArray koleksiyonunun oluşturulma şekilleri ve her bir bitin değerine ne şekilde ulaşıldığı gösterilmektedir.

```
using System;
using System.Collections;

class bitarray
{
    public static void Main()
    {
        int[] bdizi1 = {1};
        BitArray bArray1 = new BitArray(bdizi1);

        byte[] bdizi2 = {1,2};
        BitArray bArray2 = new BitArray(bdizi2);

        BitArray bArray3 = new BitArray(5,true);

        bool[] bdizi3 = {true,false,false,true,true};
        BitArray bArray4 = new BitArray(bdizi3);

        EkranaYaz(bArray1);
        EkranaYaz(bArray2);
        EkranaYaz(bArray3);
        EkranaYaz(bArray4);
    }
```



```
    public static void EkranaYaz(BitArray bArray)
    {
        IEnumerator ie = bArray.GetEnumerator();

        int i=0;
        while(ie.MoveNext())
        {
            i++;
            Console.Write("{0,-5} ",ie.Current);
            if(i%8==0)
                Console.WriteLine();
        }

        Console.WriteLine();
    }
}
```

Bu programı derleyip çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum13\kod>BitArray
True  False False False False False False False
False False False False False False False False
False False False False False False False False
False False False False False False False False

True  False False False False False False False
False True  False False False False False False

True  True  True  True  True
True  False False True  True

H:\cskitap\kitap\Bolum13\kod>
```

BitArray sınıfında Count ve Length özellikleri vardır. Count özelliği ile BitArray koleksiyonundaki elemanların yani bitlerin sayısı döndürülür. Bu özelliğin sadece get bloğu bildirilmiştir, dolayısıyla sadece okunabilirdir. Length özelliğinin ise hem set hem de get bloğu bildirilmiştir. Dolayısıyla Length özelliği ile bir BitArray koleksiyonunun uzunluğunu artırıp azaltabiliriz. Length özelliğini kullanarak bir BitArray koleksiyonunun uzunluğunu azaltırsak son bitler koleksiyondan silinecektir. Eğer uzunluğunu artırsak artırılan eleman sayısı kadar bit değeri koleksiyonun sonuna eklenir.

BitArray koleksiyonlarının elemanlarına erişmek için ayrıca Get() ve Set() metotları vardır. Bu metotların parametrik yapıları aşağıdaki gibidir.

1. bool Get(int indeks): Bu metot koleksiyondaki indeks nolu elemanın değerini geri döndürür.

2. void Set(int indeks, bool deger): Bu metot koleksiyondaki indeks nolu elemanın değerini ikinci parametre ile verilen değer ile değiştirir.

Ayrıca

void SetAll(bool deger)

metodu ile de bir koleksiyondaki bütün elemanlar true ya da false değerine çekilebilir.

BitArray sınıfının önemli bir metot grubu daha vardır. Bu metotlar koleksiyondaki bitler üzerinde bitsel işlemler yapar ve sonucu yeni bir koleksiyon referansı olarak geri döndürür. Bu metotların prototipleri aşağıdaki gibidir.

1. BitArray And(BitArray koleksiyon): Bu metot ile koleksiyondaki her bir elemanın değeri ile metodu çağıran koleksiyonun elemanları karşılıklı olarak bitsel VE işlemine tabi tutulur. Eğer her iki koleksiyonun elemanları eşit değilse ArgumentException istisnai durumu meydana gelir. Bitsel VE işleminin sonucunda, işlemin sonucunun atandığı nesnenin referansı ile geri dönülür.

2. BitArray Or(BitArray koleksiyon): And() metodu ile aynı mantıkla çalışır. Değişen tek şey bitsel işlemin 'VE' yerine 'VEYA' olmasıdır.

3. BitArray Xor(BitArray koleksiyon): And() ve Or() metotları ile aynı mantıkla çalışır. Değişen tek şey ise bitsel işlemin 'özel VEYA' olmasıdır.

4. BitArray Not(): Diğer metotlardan farklı olarak bir parametre almasına gerek yoktur. Metodu çağıran koleksiyondaki her bitin bitsel değilini alır ve yeni bir koleksiyon nesnesi referansı ile geri döner.

Aşağıda iki BitArray koleksiyonu bu metotlar ile bitsel işlemlere tabi tutulmuştur. Bitsel işlemlerden sonra koleksiyonun elemanlarının ne şekilde değiştiğini inceleyin.

```
using System;
using System.Collections;

class bitarray
{
      public static void Main()
      {
          byte[] bdizi1 = {165};
          BitArray bArray1 = new BitArray(bdizi1);

          byte[] bdizi2 = {98};
```



```
          BitArray bArray2 = new BitArray(bdizi2);

          EkranaYaz("BitArray1",bArray1);
          EkranaYaz("BitArray2",bArray2);
          EkranaYaz("And",bArray1.And(bArray2));
          EkranaYaz("Or",bArray1.Or(bArray2));
          EkranaYaz("Xor",bArray1.Xor(bArray2));
          EkranaYaz("Not",bArray1.Not());
      }

      public static void EkranaYaz(string str,BitArray bArray)
      {
          IEnumerator ie = bArray.GetEnumerator();

          int i=0;
          Console.WriteLine("[ "+str+"] ");
          while(ie.MoveNext())
          {
              i++;
              Console.Write("{0,-5} ",ie.Current);
              if(i%8==0)
                    Console.WriteLine();
          }

          Console.WriteLine();
      }
}
```

Programı derleyip çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum13\kod>BitselMetotlar
[BitArray1]
True  False True  False False True  False True

[BitArray2]
False True  False False False True  True  False

[And]
False False False False False True  False False

[Or]
False True  False False False True  True  False

[Xor]
False False False False False False False False

[Not]
True  True  True  True  True  True  True  True

H:\cskitap\kitap\Bolum13\kod>
```

Stack Sınıfı

Stack sınıfı ile System.Collections isim alanında tanımlanan veri türü bilgisayar bilimlerinde yığın olarak adlandırılır. Yığınlar bir veri grubudur; öyle ki bu gruptan belli bir anda ancak gruba en son eklenen elemanın değeri elde edilebilir. Değeri elde edilebilecek bu eleman gruba eklenen son elemandır. Dolayısıyla gruba giren ilk elemanı elde etmek için ilk eleman dışındaki bütün elemanların gruptan çıkması gerekir. Yığın ve benzeri veri gruplarında Pop() ve Push() gibi iki önemli metot bulunur. Pop() metodu bir yığından, yığına son eklenen elemanı çıkarır ve bu çıkarılan elemanın değeri ile geri döner. Push() metodu ise yığına yeni bir eleman eklemek için kullanılır. Yığın ve benzeri veri gruplarının çalışma prensibine kısaca Son Giren İlk Çıkar (Last In First Out) mantığı da denilmektedir. Bu mantığı kafanızda daha iyi canlandırmak için şöyle bir örnek verelim. Bir masaya sırayla üstüste tabakların dizildiğini düşünün. Bu masanın üzerinde 10 tabak olduğunu farzedelim. 5. tabağa erişebilmek için son eklenen 4 tabağın kaldırılması gerekir.(Bunu yapmazsanız tabakları kırma ihtimaliniz var!) Dolayısıyla tabaklardan oluşan kümedeki ilk tabağa erişmek için bütün tabakları önceden kaldırmış olmamız gerekir. Gördüğünüz gibi masaya konulan ilk tabak, gruptan en son çıkmıştır. Gruba son eklenen tabak ise gruptan ilk çıkacak elemandır.

Stack sınıfı ArrayList gibi dinamik bir veri yapısıdır. Yani Push() metodu ile yeni bir eleman eklendiğinde eğer kapasite yetersiz ise o anki kapasite 2 katına çıkarılır.

Bir Stack nesnesi oluşturmak için aşağıdaki yapıcı metotlardan birini kullanabiliriz.

1. Stack(): Bu yapıcı metot ile varsayılan kapasite olan 10 eleman içerecek şekilde bir yığın oluşturulur.

2. Stack(int kapasite): Bu yapıcı metot ile parametrede verilen kapasite kadar eleman içerecek şekilde yeni bir yığın nesnesi oluşturulur. Yığına ekleme yapıldığında kapasite aşılırsa, kapasite otomatik olarak iki katına çıkarılır. Birinci yapıcı metot ve bu metot ile oluşturulacak yığınlar boştur.

3. Stack(ICollection ic): Bu yapıcı metot ile ic koleksiyonunda bulunan elemanlardan oluşan bir yığın nesnesi oluşturulur.

Stack sınıfının kullanımına örnek vermeden önce önemli metotlarını inceleyelim. Pop() metodu yığında bulunan en üstteki elemanı geri döndürür ve bu elemanı yığından çıkarır. Peek() metodu ise son elemanı yığından silmeden geri döndürür. Bir yığına yeni bir eleman eklemek için Push() metodu kullanılır. Push() metodunun object türünden parametre aldığı için bir yığına istediğimiz türden nesneler ekleyebiliriz.

Eğer yığın boşken Pop() metodu kullanılırsa InvalidOperationException istisnai durumu meydana gelir.

Count özelliği yığındaki eleman sayısını verir. Stack sınıfının Clear() metodu yığındaki bütün elemanları yığından siler, dolayısıyla Count özelliği bu işlemden sonra 0 olur. Stack sınıfının diğer önemli metotları ise ToArray() ve Contains() metotlarıdır. ToArray() metodu ile yığındaki elemanlar klasik object türden bir diziye aktarılır. Contains() metodu ise object türden parametre olarak verilen değerin yığında bulunup bulunmadığını bildirir. Eğer belirtilen nesne yığında varsa true değerine geri döner.

Bir yığındaki elemanların tamamına bir koleksiyon gibi erişmek için Stack sınıfının GetEnumerator() metodunu kullanarak Stack sınıfına ilişkin IEnumerator referansını elde etmeliyiz.

Aşağıdaki örnekte Stack sınıfına ait metotların kullanımı gösterilmiştir.

```
using System;
using System.Collections;

class stack
{
      public static void Main(string[] args)
      {
          //Yeni bir yığın tanımlanıyor.
          Stack yigin = new Stack();

          // Yığına çeşitli değerler ekleniyor.
          yigin.Push(5);
          yigin.Push(98);
          yigin.Push(17);
          yigin.Push(4);
          yigin.Push(88);
          yigin.Push(65);

          Console.WriteLine("Yığınımızın ilk hali...");
          EkranaYaz(yigin);

          //Yığının tepesinden bir sayı alınıyor.
          //Bu değer sayi değişkenine atanıp ekrana
                                          yazdırılıyor.
          int sayi = (int)yigin.Pop();
          Console.WriteLine("\nYığından {0} sayısını
                                          aldık", sayi);

          // Yigininin tepesinden bir sayı daha aldık
          // ve bunu sayi değişkenine atayıp ekrana
```

```
                                        yazdırıyoruz.
        sayi = (int)yigin.Pop();
        Console.WriteLine("\nYığından {0} sayısını
                                        aldık", sayi);

        //Şimdi yığının tepesindeki sayıya bakalım
        //bu sayıyı yığından çıkarmıyoruz. Sadece ne
        //olduğuna bakıyoruz..
        sayi = (int) yigin.Peek();
        Console.WriteLine("\nYığının tepesindeki sayı şu
                                    anda: {0}", sayi);

    }

    public static void EkranaYaz(Stack yigin)
    {
        object obj = new Object();
        Stack yeniYigin = (Stack)yigin.Clone();

        if(yigin.Count!=0)
        {
            while(yeniYigin.Count>0)
                Console.WriteLine("\t"+
                            yeniYigin.Pop().ToString());

        }
        else
            Console.WriteLine("Yığın boş!");
    }
}
```

Programı derleyip çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum13\kod>Stack
Yigimizin ilk hali...
        65
        88
        4
        17
        98
        5

Yigindan 65 sayisini aldik

Yigindan 88 sayisini aldik

Yiginin tepesindeki sayi su anda : 4

H:\cskitap\kitap\Bolum13\kod>
```


Bu programdaki EkranaYaz() metodunu IEnumerator arayüzünü kullanarak aşağıdaki gibi de yazabilirdik.

```
public static void EkranaYaz(Stack yigin)
{
    IEnumerator ie = yigin.GetEnumerator();

    while(ie.MoveNext())
        Console.WriteLine("\t" + ie.Current);
}
```

Queue Sınıfı

Queue, veri yapısı çalışma biçimi olarak Stack sınıfına benzer. Ancak algoritma aynı değildir. Queue bir kuyruğu temsil eder. Kuyrukta ilk giren ilk çıkar mantığı vardır. Gerçek hayattaki kuyruklarda da böyledir. Mesela otobüs sırasına giren ilk kişi otobüse ilk önce biner. Queue veri yapısında da kuyruğa eklenen eleman ilk önce gruptan çıkarılır. Queue sınıfında kuyruğun en başındaki elemanı çıkarmak ve bu eleman ile geri dönmek için Dequeue() metodu kullanılır. Bu metot Stack sınıfının Pop() metodunun karşılığı gibi düşünülebilir. Enqueue() metodu ile de parametre olarak verilen object nesnesi kuyruğun sonuna eklenir. Bu da Stack sınıfındaki Push() metodunun karşılığıdır. Diğer bir metot olan Peek() metodu ise kuyruktaki ilk elemanın değerini döndürür ancak kuyruktan bu elemanı çıkarmaz. Stack sınıfındaki Peek() metodu ile aynı işlevdedir.

Boş bir kuyruk nesnesi ile Dequeue() metodunu kullanmak InvalidOperationException istisnai durumunu oluşturur.

Queue sınıfı standart bir koleksiyon özelliklerine sahip olduğu için GetEnumerator() metodu ile bir IEnumerator referansı oluşturup kuyruktaki bütün elemanlara ulaşabiliriz. Ya da Queue sınıfının Count özelliği kadar Dequeue() metodunu kullanıp her defasında bu metot ile geri dönen değer elde edilerek kuyruktaki bütün elemanlara ulaşılabilir.

Bir kuyruk nesnesi, aşağıdaki yapıcı metotlardan biri ile oluşturulabilir.

Queue(): Bu yapıcı metot ile varsayılan kapasite olan 32 eleman içerecek şekilde bir kuyruk nesnesi oluşturulur.

Queue(int kapasite): Bu yapıcı metot ile varsayılan kapasite yerine kapasite ile verilen sayıda eleman içerecek şekilde bir kuyruk nesnesi oluşturulur.

Queue(ICollection ic): Bu yapıcı metot ile ic koleksiyonunda bulunan elemanlardan oluşan bir kuyruk nesnesi oluşturulur.

Queue sınıfının diğer önemli iki metodu ise ToArray() ve TrimToSize() metotlarıdır. ToArray() metodu Stack sınıfında olduğu gibi kuyruğun elemanlarını klasik bir diziye aktarır. TrimToSize() metodu ise kuyruğun kapasitesini kuyruktaki eleman sayısına düşürür. Yani Count özelliği ile Capacity özelliği birbirine eşitlenir.

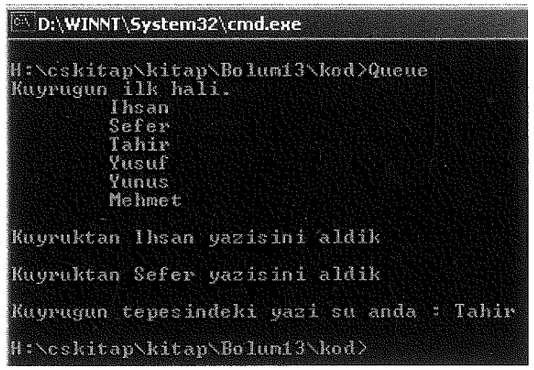
Aşağıdaki örnekte Queue sınıfının ve metotlarının kullanımına bir örnek verilmiştir.

```
using System;
using System.Collections;

class kuyruk
{
      public static void Main()
      {
          //Yeni bir kuyruk tanımlanıyor.
          Queue kuyruk = new Queue();

          //Kuruğa çeşitli değerler ekleniyor.
          kuyruk.Enqueue("İhsan");
          kuyruk.Enqueue("Sefer");
          kuyruk.Enqueue("Tahir");
          kuyruk.Enqueue("Yusuf");
          kuyruk.Enqueue("Yunus");
          kuyruk.Enqueue("Mehmet");

          Console.WriteLine("Kuyruğun ilk hali.");
          EkranaYaz(kuyruk);

          //Kuyruğun başından bir nesne aldık
          //ve bunu sayi değişkenine atayıp ekrana yazdıralım
          string isim = (string)kuyruk.Dequeue();
          Console.WriteLine("\nKuyruktan {0} yazısını aldık",
                                                  isim);

          //Kuyruğun tepesinden bir nesne daha aldık
          //ve bunu isim değişkenine atayıp ekrana yazdıralım
          isim = (string)kuyruk.Dequeue();
          Console.WriteLine("\nKuyruktan {0} yazısını aldık",
                                                  isim);
```



```
          //Şimdi, kuyruğun tepesindeki sayıya bir bakalım
          //bu sayıyı kuyruktan çıkarmıyoruz. Sadece ne
          //olduğuna bakıyoruz.
          isim = (string)kuyruk.Peek();
          Console.WriteLine("\nKuyruğun tepesindeki yazı şu
                                          anda : {0}", isim);

      }

      public static void EkranaYaz(Queue kuyruk)
      {
          IEnumerator ie = kuyruk.GetEnumerator();

          while(ie.MoveNext())
              Console.WriteLine("\t" + ie.Current);
      }
}
```

Programı derleyip çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.



Şablon Tipli Koleksiyon Nesneleri

C# 2.0'a eklenen yeni şablon tip özelliği sayesinde klasik tipli koleksiyon nesnelerinin kullanımı artık önerilmemektedir. Şablon türler konusunda da söylendiği gibi buraya kadar değindiğimiz koleksiyon nesnelerinin tamamı object veri tipi ile çalıştığı için hem performans kaybı meydana gelmekte hem de tür güvenliği sağlanamamakta. Buna çözüm olarak C# 2.0 ile birlikte System.Collections.Generic isim alanı altında bütün koleksiyonlar şablon tipli versiyonları geliştirilmiştir.

Şablon tipli yeni koleksiyonların kullanımı ve amaçları klasik koleksiyonlarla tamamen aynıdır. Tek fark saklanacak nesnelerin tiplerinin tek olması ve koleksiyon yaratıldığı anda belirlenmesidir.

List<T> Koleksiyonu

List<T> koleksiyonu, klasik ArrayList'in karşılığıdır. Hemen hemen birçok özelliği ortak olduğundan burada ayrıca değinilmeyecektir. Örnek bir liste tanımı aşağıdaki gibi yapılabilir.

```
List<int> listem =  new List<int>();
```

ya da

```
List<string> listem2 = new Liste<string>();
```

Dictionary<K,V> Koleksiyonu

Dictionary<K,V> koleksiyon klasik Hashtable'ın karşılığıdır. Hemen hemen birçok özelliği ortak olduğundan yine burada ayrıca değinilmeyecektir. Örnek bir nesne tanımı aşağıdaki gibi yapılabilir.

```
Dictionary<int,string> personeller =  new
Dictionary<int,string>();

personeller.Add(31,"Ali");

personeller.Add(53,"Veli");

personeller.Add(23,"Ahmet");

string ad = personeller.Add[ 3] ;
```

SortedDictionary<K,V> Koleksiyonu

SortedDictionary<K,V> koleksiyon klasik **SortedList**'in karşılığıdır. Hemen hemen birçok özelliği ortak olduğundan yine burada ayrıca değinilmeyecektir. Örnek bir nesne tanımı aşağıdaki gibi yapılabilir.

```
SortedDictionary<int,string> personeller =  new
SortedDictionary<int,string>();

personeller.Add(31,"Ali");

personeller.Add(53,"Veli");

personeller.Add(23,"Ahmet");

string ad = personeller.Add[ 3] ;
```


IEnumerable<T> ve IEnumerator<T> Arayüzleri

Klasik versiyonlarından türetilmiş olan şablon tipli bu arayüzler, şablon tipli koleksiyonlar için arayüz niteliğindedir. Kendi koleksiyonlarımızı yazarken artık bu arayüzleri kullanmamız gerekiyor. Eski arayüzler elbetteki geriye dönük uyumluluk açısından halen varlıklarını devam ettirmektedirler.

Queue<T>, Stack<T> ve LinkedList<T> Koleksiyonları

Bu bölümde incelediğimiz **Queue** ve **Stack** koleksiyonlarınında şablon tip versiyonları bulunmaktadır. Buna ek olarak .NET 2.0 ile birlikte koleksiyon elemanlarını zincir şeklinde tutabileceğimiz **LinkedList<T>** isimli yeni bir koleksiyon daha gelmiştir. Bu bölümde bu koleksiyon hakkında detaylı bilgi yoktur ancak MSDN dökümanlarından yeteri kadar kaynak koda erişmeniz mümkündür.

.NET Framework 2.0 ile birlikte object veri tipi ile çalışan koleksiyonlar artık demode olmuştur. Yeni geliştirilecek bütün projelerde şablon tipli koleksiyonlar kullanılmalıdır.

Koleksiyonlar ile İlgili Diğer Konular

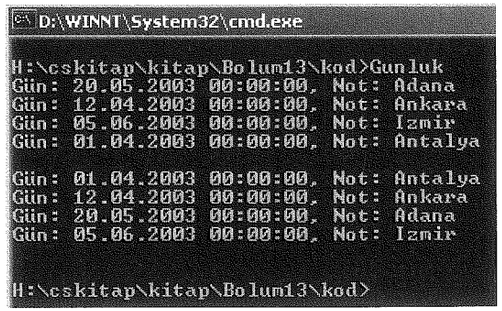
System.Collections isim alanında bazı özel durumlar için geliştirilmiş birkaç koleksiyon sınıfı daha vardır. Ancak bu sınıfları burada incelemeyeceğiz. **System.Collections** isim alanındaki diğer sınıflar için MSDN kütüphanesine başvurabilirsiniz.

Konuyu bitirmeden önce koleksiyon sınıflarında kendi bildirdiğimiz sınıfları nasıl kullanabileceğimize bir örnek verelim. Aşağıdaki örnekte bildirilen **Gunluk** sınıfı ile bir kişinin gideceği yerlerin günlük listesi tutulmaktadır. Bir **ArrayList** sınıfına eklenen **Gunluk** türünden nesnelerin **Sort()** metodu ile tarihe göre sıralamasını yapmak için Gunluk sınıfı, **CompareTo()** metodunun bildirildiği **IComparable** arayüzünden türetilmiştir. Aslında .NET kütüphanesindeki **DateTime** sınıfı ile karşılaştırma operatörleri kullanılarak iki tarih bilgisi karşılaştırılabilir, bizim **CompareTo()** metodunu uygulamamızın sebebi; sadece **Sort()** metodunun **Gunluk** sınıfındaki hangi elemana göre sıralama yapacağını bildirmektir.

```
using System;
using System.Collections;

class Gunluk : IComparable
{
      public DateTime Tarih;
      public string Not;
```

```
        public Gunluk(string not, DateTime tarih)
        {
            Tarih = tarih;
            Not = not;
        }

        public int CompareTo(object o)
        {
            Gunluk g = (Gunluk)o;
            return Tarih.CompareTo(g.Tarih);
        }
}

class CompareToTest
{
        public static void Main()
        {
            ArrayList notlar = new ArrayList();

            notlar.Add(new Gunluk("Adana",new DateTime(2003,5,20)));
            notlar.Add(new Gunluk("Ankara",new
DateTime(2003,4,12)));
            notlar.Add(new Gunluk("İzmir",new DateTime(2003,6,5)));
            notlar.Add(new Gunluk("Antalya",new
DateTime(2003,4,1)));

            EkranaYaz(notlar);

            notlar.Sort();

            EkranaYaz(notlar);
        }

        public static void EkranaYaz(ArrayList alist)
        {
            foreach(Gunluk g in alist)
            {
                Console.Write("Gün: " + g.Tarih + ", ");
                Console.Write("Not: " + g.Not);
                Console.WriteLine();
            }

            Console.WriteLine();
        }
}
```


Programı derleyip çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.



Ekran görüntüsünden de görüleceği üzere **ArrayList** sınıfının **Sort()** metodu kullanıldıktan sonra **Gunluk** nesneleri Tarihe göre sıralanmıştır. Diğer arayüzleri de kendi sınıflarımıza bu şekilde uygulayarak .NET platformuna tam uyumlu tasarımlar oluşturabiliriz.

Bu bölümde koleksiyon sınıfları ile çalışarak programlarımızı daha etkili bir biçimde yazabileceğimizi gördük. Bundan sonra koleksiyon sınıflarını sık sık kullanacağınızı düşünüyorum. Çünkü bu sınıflar gerçekten çok kullanışlıdır.

Yeni Bir Iterasyon Tekniği (yield)

Nesne Yönelimli Programlama ve Kalıtım isimli bölümünde (arayüzlerin anlatıldığı bölüm) hatırlayacağınız üzere bir nesnenin içinde bulunan herhangi bir koleksiyonu **foreach** döngü yapısı ile erişilebilir hale getirmek için ilgili sınıfa **IEnumerable** arayüzünü uygulamıştık. Bu işlemin standart bir yapısı olsa da bir hayli kod yazmamız gerekiyordu. Dilin yapısına C# 2.0 ile birlikte yeni bir iterasyon tekniği daha eklendi. Yine bu yapıda **IEnumerable** ve **IEnumerator** isimli arayüzlerle işbirliği yapıyor olsa da biz programcılar olarak bununla çok fazla ilgilenmemiz gerekmiyor.

Bu bölümü daha iyi anlayabilmek için yukarıdaki bahsedilen bölümdeki kısmı okumanızı öneriyorum.

C# 2.0 ile ile birlikte dile **yield return** ve **yield break** isimli iki anahtar sözcük eklendi. Bu anahtar sözcükler yalnızca geriye **IEnumerator** ya da **IEnumerator <T>** döndüren metotlar içerisinde kullanılabilir. **yield return** belli bir sıra ile çağrılırsa çağrılma sırasına göre **yield return** deyimi ile geriye döndürülen değerler bir koleksiyon kümesi oluştur. Buradaki önemli nokta elbette ki kümenin **IEnumerator** türünden olması ve dolayısıyla **foreach** yapısı ile direkt olarak çalışabilmesidir.

Önemli Not: yield return ifadesi bir döngü içerisinde kullanılabileceği gibi arda arda tek başına da çağrılabilir.

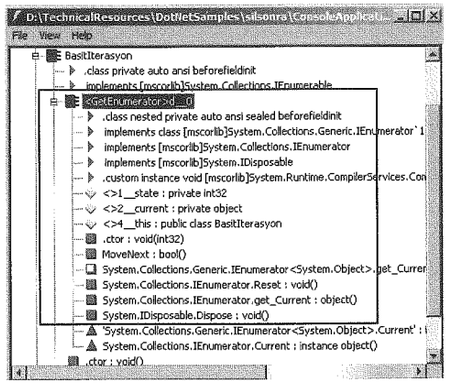
Öncelikle basit bir örnek ile **yield return** ile nasıl foreach uyumlu bir koleksiyon oluşturulduğunu görelim ardından daha güzel bir örnek yapacağız.

```
using System;
using System.Collections;

class BasitIterasyon : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        yield return 1;
        yield return 2;
        yield return 3;
    }
}

class Program
{
    static void Main(string[] args)
    {
        BasitIterasyon b = new BasitIterasyon();
        foreach (int sayi in b)
        {
            Console.WriteLine(sayi.ToString() + " ");
        }
    }
}
```

Yukarıdaki örnekte görüldüğü üzere **IEnumerable** arayüzünde bulunan **GetEnumerator** metodunda peşpeşe üç defa **yield return deyimi** çağrılmıştır. **yield return deyimi** ile birlikte her türlü nesne tipi döndürülebilir.

Önemli Not: yield return ile **return** deyimleri birbirleriyle karıştırılmamalıdır. yield return deyimi metodun sonlanması ve geriye bir şey döndürülmesi anlamına gelmemektedir.

Oluşturduğumuz programın *exe* dosyasını .NET ile birlikte gelen **ILDASM** (*Intermadiate Language Disassembler*) aracı ile incelediğimizde **yield return** deyimlerini içeren kısımların aslında bizim daha önceki yaptığımız iterasyon tekniği ile yapıldığını aşağıdaki gibi görürüz.



Yukarıdaki örnekte dikkat edilirse **IEnumerable** arayüzünü kullandık. Halbuki C# 2.0 ile birlikte gelen **şablon türler** (*generics*) sayesinde koleksiyonlarımızı ve dolayısıyla iterasyonları da hem tür güvenli hemde daha performanslı yazabiliriz. Bu amaçla **System.Collections.Generic** isim alanı altında bütün koleksiyon arayüzlerinin şablon tip alan versiyonları bulunmaktadır.

Örneğin, IEnumerable<T>, ICollection<T>, IEnumerator<T> ve IDictionary<T> gibi...

İkinci örneğimizde ise **foreach** ile uyumlu çalışan basit bir sayaç yapacağız. Sayaç nesnelerimizin maksimum değeri ve kaçar kaçar sayacağı, nesneye, nesneyi kullanan yerde verilerek parametrik yapılacaktır. Örneğin 0'dan 68'e kadar üçer üçer sayıp bir koleksiyon olarak bize veren kod blokları aşağıdaki gibi yazılabilecektir.

```
AdimSayici adimci =  new AdimSayici();

adimci.AdimSayisi = 3;

adimci.Maksimum = 68;


foreach (int sayi in adimci)

{

Console.WriteLine(sayi.ToString() + " ");

}
```

Dikkat ederseniz burada önemli olan algoritma değil bir nesneyi foreach içerisinde rahatlıkla kullanabilmemizdir. Bu örneğin geçerli olabilmesi için **AdimSayici** isimli sınıfın bildirimi aşağıdaki gibi olmalıdır. AdimSayici içerisindeki nesnelerimizin tipi int olarak belirlendiğinden iterasyonumuzun daha performanlı çalışabilmesi için bu defa sınıfımızı **IEnumerable<T>** arayüzünden türeteceğiz.

```
using System;
using System.Collections.Generic;
using System.Collections;

class AdimSayici :  System.Collections.Generic.IEnumerable<int>
{
    public int AdimSayisi;
    public int Maksimum;

    IEnumerator<int> IEnumerable<int>.GetEnumerator()
    {
        for (int i = 0; i < Maksimum; i+=AdimSayisi)
        {
            yield return i;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return null;
    }
}

class Program
{
    static void Main(string[] args)
    {
        AdimSayici adimci =  new AdimSayici();
        adimci.AdimSayisi = 4;
        adimci.Maksimum = 40;

        foreach (int sayi in adimci)
        {
            Console.WriteLine(sayi);
        }

        Console.WriteLine("İlk Adımcı Bitti");

        adimci.AdimSayisi = 6;
        adimci.Maksimum = 65;

        foreach (int sayi in adimci)
        {
            Console.WriteLine(sayi);
        }
    }
}
```


Örneği çalıştırıp derlediğinizde AdimSayici içinde yaptığımız iterasyon sayesinde ilk önce 0'dan 40'a kadar 4'er 4'er sonra da 0'dan 65'e kadar 6'şar 6'şar sayıp ekrana yazacaktır.

Şablon türler ile ilgili daha detaylı bilgi almak için lütfen **Şablon Türler** isimli bölümü okuyunuz.

Yukarıdaki örneği dikkatle incelediyseniz içi boş olan bir **GetEnumerator()** metodu bulunmaktadır. Bunu yazmamızın tek sebebi **IEnumerator<T>** arayüzünün aynı zamanda **IEnumerator** arayüzünden türemesidir. Kısaca bir sınıf içinde türediği arayüzdeki bütün üye elemanları uygulamak zorunda olduğumuz için yazdık.

Konuyu pekiştirmek için arayüzlerin anlatıldığı konuda verdiğimiz örneği bu yöntem ile tekrar yazmanızı tavsiye ediyorum.

Son olarak konuyu bitirmeden **yield break** deyimi ile ilgili bir noktaya değinmek istiyorum. Koleksiyonu oluşturduğumuz döngü içinde belirli bir koşul altında koleksiyonun oluşma işlemini sonlandırıp aynı zamanda metodu da sonlandırmak istersek bu durumda yield break deyimi kullanılır. Örneğin bir önceki örneğimizde sayı 50 değerine geldiğinde koleksiyonu bitirmek istersek yield break deyimini aşağıdaki gibi kullanabiliriz.

```
IEnumerator<int> IEnumerable<int>.GetEnumerator()
{
  for (int i = 0; i < Maksimum; i+=AdimSayisi)
  {
     if( i == 50)
       yield break;

     yield return i;
  }
}
```

15

Veritabanı ve XML İşlemleri

- **ADO.NET ile Veritabanı İşlemleri**
  - **ADO.NET Nesne Modeli**
  - **Veritabanına Bağlanmak**
  - **OleDbCommand ve OleDbDataReader**
  - **DataSet, DataAdapter, DataTable, DataRow, DataColumn**
- **ADO.NET 2.0 ve DbProviderFactory (Veri Sağlayıcı Bağımsız Erişim)**
- **XML İşlemleri ve System.Xml**
  - **XML Belgeleri**
  - **System.Xml İsim Alanı**
  - **DOM (Document Object Model)**

Bu bölümde .NET sınıf kütüphanesinin en önemli iki isim alanını giriş niteliğinde inceleyeceğiz. Bu isim alanları, veritabanı işlemlerini gerçekleştirmek için kullanılan System.Data ve XML dosyalarıyla çalışmak için çeşitli sınıflar içeren System.Xml'dir. Veritabanı ve XML işlemleri çok geniş bir konudur. Bu kitapta bu konulara sadece giriş niteliğinde değinilecektir. Ancak temel kavramları öğrendiğinizde MSDN kütüphanesinden faydalanarak bu isim alanlarındaki diğer sınıfları rahatlıkla kullanabilir hale gelirsiniz. Ayrıca bu bölüm yazılırken sizin veritabanı ve XML gibi konularda en azından temel düzeyde bilgi sahibi olduğunuz varsayılmıştır. Bu bölümde XML dosyalarının oluşturulması, XML sözdizimi, veritabanı oluşturma ve sorgulama gibi işlemler anlatılmayacaktır. Bu bölümdeki amacımız, C# ile System.Data ve System.Xml isim alanlarındaki yapıların nasıl kullanıldığını göstermektir.

ADO.NET ile Veritabanı İşlemleri

ADO.NET, veritabanlarındaki bilgileri elde etmek veya veritabanında bulunan verileri değiştirmek için geliştirilmiş bir veri erişim aracıdır. ADO.NET, .NET sınıf kütüphanesindeki System.Data isim alanındaki sınıfların tamamını temsil eder. ADO.NET'in klasik ADO'ya göre getirdiği yenilikler ile ADO kütüphanesinin geliştirildiği açıktır. Ayrıca klasik ADO'da olmayan birçok özelliğin ADO.NET'te bulunduğunu da söylemek gerekir.

Burada .NET platformunun getirdiği en önemli yeniliklerinden biri olan veriye ulaşım tekniklerinden bahsedeceğiz. Microsoft firmasının geliştirdiği veritabanı erişim sınıflarının genel adı ADO idi. ActiveX teknolojisi ile geliştirilmiş olan ADO sınıflar topluluğu yerel ya da uzaktaki makinalarda bulunan verilere erişmemizi sağlıyordu. ADO.NET ile değişmeyen tek felsefe veriye erişim dersek doğru söylemiş oluruz, zira ADO.NET 'in klasik ADO'dan çok büyük farklılıkları vardır. Bu farklılıklardan yeri geldikçe bahsedeceğiz. Konunun ilerleyen bölümlerinde ADO.NET'in en büyük yeniliklerinden olan offline çalışmanın temelleri de anlatılacaktır.

ADO.NET'i eski ADO'nun bir üst versiyonu gibi görmemek gerekir. Her ne kadar eski ADO'da bulunan "connection" ve "command" gibi eş kavramlara sahipse de, eski ADO da olmayan "DataSet" gibi birçok yeni kavramı da vardır. Kısaca ADO.NET, .NET paltformu için geliştirilmiş, offline çalışma sistemini destekleyen, web uygulamaları gibi n-katmanlı uygulamalarda büyük kolaylık sağlayan yeni bir veri erişim teknolojisidir diyebiliriz. Eski ADO'nun çalışma mantığı olan sunucu/istemci kavramı yerini, offline çalışmayı sağlayan DataSet kavramına bırakmıştır (Bu, eski ADO'daki sistemin kaldırıldığı anlamına gelmemelidir.) ADO.NET'in diğer önemli özelliği ise doğrudan XML desteği vermesidir. İçsel bir mekanizma ile veriler XML formatına dönüştürülür. Bu içsel mekanizma HTTP protokolü ile sağlandığı için herhangi bir firewall kısıtlamasından bahsetmek mümkün değildir.

ADO ile ADO.NET arasındaki temel farkları bir tablo halinde inceleyelim:

ADO	ADO.NET
ADO nesne modeli 6 temel yapıdan oluşmaktadır. Bunlar: - Connection Nesnesi - Recordset Nesnesi - Command Nesnesi - Error Nesnesi - Field Nesnesi - Parametre Kolleksiyonu	ADO.NET modeli veriye erişim olarak iki ana metot kullanır: Online Erişim (Connected Access) Offline Erişim (Disconnected Access)
ADO, Recordset nesnesi ile offline çalışma modeline sınırlı bir imkan sağlar.	ADO.NET iki yeni nesne modeli sunar. Bunlar DataSet ve DataAdapter nesneleridir. Bu yapıların ikisi de offline çalışma için gereklidir. DataSet tasarım olarak offline çalışacak şekilde düzenlenmiştir
ADO modelinde çok fazla kullanılmayan yapılar uygulamanın performansını düşürür.	ADO.NET modeli ise daha yüksek performans sağlar.
Doğrudan bir XML desteği yoktur.	Doğrudan XML desteği verir. ADO.NET, XML ile entegre çalıştığı için bir tablodaki verileri XML formatına dönüştürmek çok basittir. Eski ADO'da ise böyle bir durum söz konusu değildir.

ADO.NET'i oluşturan sınıfların çoğu bir DataSet oluşturmak ve bu DataSet üzerinde işlemler yapmak içindir. Peki nedir bu DataSet? DataSet, veritabanında bulunan bir veya daha fazla tablonun içinde barındığı tablolar kolleksiyonudur. DataSet içinde bulunan tablolar asıl veri kaynağından tamamen bağımsız bir şekilde çalışır. Yani DataSet nesnesinin içini bir kere doldurduğunuzda veritabanı bağlantınızı kesseniz bile DataSet ile sanki bir veritabanına bağlıymışsınız gibi DataSet üzerinden tablolar arasında ilişki kurabilir, yeni bir veri ekleyebilir, yeni sorgulamalar yapabilir ve hatta kayıtlar silip işiniz bittiğinde tekrar veritabanına kaydedebilirsiniz. İşte DataSet nesnesinin bu çalışma prensibine **disconnected (offline) çalışma** denmektedir. İşin ilginç tarafı ağda bulunan bir veritabanınındaki bütün tabloları DataSet içine alıp sanki kendi makinamızdaki veritabanı üzerinde çalışıyormuşuz gibi de yapabiliriz. Bu da doğaldır ki büyük bir kaynak tasarrufu ve performans artışı sağlamaktadır. Eski ADO'da bulunan Recordset nesnesi ile hatırlarsanız sadece bir tablo üzerinde çalışabiliyorduk. ADO.NET ile online çalışma DataAdapter ve DataSet dediğimiz nesneler ile gerçekleştirilir.

ADO.NET veri erişimi olarak iki tür destek sağlar: birincisi SQL Server ile yüksek performanslı erişim, ikincisi ise OLEDB protokolunu destekleyen diğer veritabanlarına bağlanmaktır. Eğer veritabanı olarak SQL Server kullanıyorsak SQL Server için özel yazılmış sınıfları kullanarak daha yüksek performans elde edebiliriz. Bu sınıflar SQL Server'ın TDS (Tabular Data Stream) dediğimiz yapısı ile doğrudan ilişkili olduğu için tercih edilir. Yani bu sınıflar ile SQL sunucusundaki verilere arada OLEDB gibi başka API'ler olmadan direkt erişilir.

ADO.NET Nesne Modeli

ADO.NET'in altyapısını oluşturan sınıflar System.Data isim alanının içinde bulunurlar. ADO.NET'e ilişkin bütün sınıflar System.Data.dll isimli tek bir assembly'de toplanmıştır. Bu yüzden ADO.NET kapsamındaki sınıfları programlarımız içinde kullanırsak derleyiciye Data.dll assembly'sinin referansını vermemiz gerekir. Aşağıda ADO.NET ile ilgili isim alanları ve bu isim alanlarındaki sınıfların ne amaçla kullanıldıkları belirtilmektedir.

System.Data

ADO.NET in en temel isim alanıdır. Veriyi sembolize eden veri türleri burada bildirilmiştir. Mesela, tablo, sütun, satır, koşullar ve DataSet gibi veriyle doğrudan ilişkisi olan türler bu isim alanındadırlar. Veri kaynağına bağlanmak için gerekli veri türleri bu isim alanında değildir. Kısacası ADO.NET mimarisinin temel yapı taşlarını oluşturan sınıflar bu isim alanında bulunur.

System.Data.Common
Bu isim alanı managed provider dediğimiz veri erişim kaynaklarını kullanan sistemlere yönelik sınıfları içerir.

System.Data.OleDb
Buradaki sınıfların yapısı eski ADO'ya biraz yakınlığı ile bilinir. OLEDB protokolünü destekleyen veritabanı sistemlerine erişmek için kullanılan sınıflar bu isim alanı içindedir.

System.Data.SqlClient
OleDb servislerini kullanmadan Sql Server sunucusuna doğrudan bağlanıp performansı artırmak istiyorsak bu isim alanı altında bulunan ve MS SQL Server için özel olarak geliştirilmiş sınıfları kullanmalıyız.

System.Data.SqlTypes
Bu isim alanında bulunan veri tipleri, MS SQL Server içinde desteklenen doğal veri türlerini içerir. Tabii ki bu veri tiplerinin CLR (common language run-time) deki karşılıklarını da kullanabiliriz, fakat sql veri tiplerini kullanmak SQL Server verilerine erişmede artı bir performans sağlayacaktır.

Veritabanına Bağlanmak
Günümüzde çeşitli veritabanı sistemleri geliştirilmiştir. Ancak popüler veritabanı sistemleri OLEDB dediğimiz veriye erişim modelini destekledikleri için her bir veritabanı için ayrı işlem yapmamıza gerek yoktur. ADO.NET ile SQL Server için ayrı bir model getirilmesi ise Microsoft firmasının kendi ürününe olan doğal ve beklenen bir yaklaşımıdır. Bu bölümde daha genel olması açısından OLEDB API'lerini kullanarak Access veritabanlarına erişeceğiz. System.OleDb isim alanında bulunan sınıflar OLEDB API'lerini kullanarak, bu yöntemi destekleyen veri depolayan sistemlere erişmemizi sağlar. Bu veri kaynağı illa bir veritabanı olmak zorunda değildir. Örneğin bir Excel belgesindeki verilere de OLEDB ile erişmek mümkündür.

Bir veri kaynağına erişmek için veri kaynağı ile bir bağlantının sağlanması gerekir. Bu bağlantıda bağlantı türü ve bağlantının yapılacağı veri kaynağı belirtilir. Veri kaynağına olan bağlantıyı sembolize eden sınıflar `SqlConnection` ve `OleDbConnection`'dır. Bu konuda OleDbConnection sınıfını kullanacağız. Çünkü OleDb ile hem SQL sunucusuna hem de diğer veritabanı sistemlerine bağlanabiliriz. Farklı veritabanı sistemlerine bağlanmadaki tek fark OleDbConnection nesnesini oluşturma sırasında görülür. Aşağıda SQL Server ve Access veritabanlarına SqlConnection ve OleDbConnection ile nasıl bağlanıldığı gösterilmektedir.

Aşağıdaki bağlantı nesnesi OLEDB ile SQL Server sunucusuna bağlantı gerçekleştirir. OleDbConnection sınıfı System.Data.OleDb isim alanında bulunur.

```
//veri kaynağına erişmek için çeşitli bilgiler hazırlanır.
string kaynak="Provider=SqlOleDb;server=SunucuAdı;uid=Kullanıcı;
pwd=Sifre; database=Northwind;";
//bağlantı nesnesi kaynak ile oluşturulur.
OleDbConnection baglanti = new OleDbConnection(kaynak);
//bağlantı nesnesinin Open() metodu kullanılarak bağlantı //
açılır
baglanti.Open();
```

Aşağıdaki bağlantı biçimi ise OleDb üzerinden Access veritabanlarına erişmek için kullanılır.

```
string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=vt.
mdb";
OleDbConnection baglanti = new OleDbConnection(kaynak);
baglanti.Open();
```

Görüldüğü gibi değişen tek şey veri kaynağına erişmek için kullanılan string türden olan kaynak nesnesidir. Diğer bir bağlantı biçimi ise System.SqlClient isim alanında bulunan ve SQL Server sunucusuna erişmek için özelleştirilmiş SqlConnection sınıfını aşağıdaki gibi kullanmaktır.

```
string kaynak="server=SunucuİSmi;uid=Kullanıcı; pwd=;Şifre
database=Northwind;";
SqlConnection baglanti = new SqlConnection(kaynak);
baglanti.Open();
```

Bu kitap kapsamında ikinci yöntemde gösterilen OleDb üzerinden Access veritabanına erişim yöntemi kullanılacaktır. Zaten OleDb API'lerini destekleyen diğer veritabanı sistemleri ile çalışmak, ADO.NET ile tamamen aynıdır. Daha önce denildiği gibi değişen tek şey bağlantı nesnesinin oluşturulma biçimidir.

Şimdi bir Access veritabanını aşağıdaki alanlara sahip olacak şekilde oluşturun ve alanlara aşağıdaki gibi çeşitli veriler girin.

	Alan Adı	Veri Türü
🔑	KullanıcıID	Otomatik Sayı
	Adı	Metin
	Soyadı	Metin
	Yaş	Sayı
	KullanıcıAdı	Metin
	Şifre	Metin
	Puan	Sayı

KullanıcıID	Adı	Soyadı	Yaş	KullanıcıAdı	Şifre	Puan
1	Sefer	Algan	23	algans	SJKAMSJN	251
2	Gökcen	Yıldırım	22	y_gokcen	kkkoI569	120
3	İnan	Erdem	23	erdemi	9089yhYL	99
4	İhsan	Algan	20	madenci	56569874	265
5	A.Faruk	Nacaroğlu	22	yakamoz	yaKAYTA	25
6	Ahmet	Hançer	35	hancer	OKUHGTF	0
7	Cemal	Öztürk	26	mcemalo	2365987412	14
8	Ebru	Salmaner	28	ebruli	ndbdbOIU89	130
9	Fatih	Azaklı	26	afatih	58987458	198
10	Melih	Kalafat	22	kalafatme	IKJU7898	25
11	Cezmi	Kara	54	karac	op989898	325

Oluşturduğunuz bu tabloyu Uyeler ismiyle, veritabanını ise veri.mdb ismiyle programları yazdığınız klasöre kaydedin. Daha sonra aşağıdaki program ile veritabanına OleDb üzerinden bağlantı sağlayın.

```
using System;
using System.Data.OleDb;

public class AccessOleDb
{
        public static void Main()
        {
                string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data
                                                Source=veri.mdb";

                OleDbConnection baglanti = new OleDbConnection(kaynak);

                baglanti.Open();

                Console.WriteLine("Bağlantı sağlandı...");

                baglanti.Close();
        }
}
```


Bağlantı nesnesi ile işimiz bittiğinde Close() metodunu kullanarak bağlantı için ayrılan kaynakların iade edilmesini sağlıyoruz.

Programı aşağıdaki gibi derleyip çalıştırdığınızda ekrana "Bağlantı sağlandı..." yazıyorsa veritabanına bağlantı işlemini başarıyla gerçekleştirdik anlamına gelir.

```
Visual Studio .NET Command Prompt

H:\cskitap\kitap\Bolum14\kod>csc OleDbAccess.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

H:\cskitap\kitap\Bolum14\kod>OleDbAccess
Baglanti saglandi...

H:\cskitap\kitap\Bolum14\kod>
```

Bundan sonra yapmamız gereken ise System.Data isim alanında bulunan çeşitli sınıfları kullanarak veritabanındaki bilgileri elde etmektir.

Eğer derleme işlemi başarısız olursa derleyiciyi

```
csc OleDbAccess.cs /r:System.Data.dll
```

şeklinde çalıştırın.

OleDbCommand ve OleDbDataReader

OleDbCommand nesnesi bir veri kaynağındaki tablodan SQL cümleciği kullanarak sorgulama yapmamızı sağlar. OleDbDataReader sınıfı ise OleDbCommand nesnesinin çalıştırılması ile elde edilen sonuçları okumak için kullanılır. OleDbDataReader sınıfındaki çeşitli metotlar yardımıyla sorgulama sonucu elde edilen bilgiler okunur.

OleDbCommand nesnesini 3 değişik şekilde oluşturabiliriz. Bunlar aşağıda sırayla açıklanmıştır.

1. OleDbCommand nesnesi bir bağlantı nesnesi ve bir SQL cümleciği ile oluşturulabilir. SQL cümleciği veritabanından bir sorgulama yapmamızı sağlar. SQL cümleciği ile Uyeler tablosundan sadece işimize yarayan sütunları ya da belirlenen bir kritere uyan kayıtları elde edebiliriz. Örneğin, Uyeler tablosundan Adı ve Soyadı sütunlarını elde etmek için aşağıdaki SQL cümleciği kullanılabilir.

```
SELECT Adı, Soyadı FROM Uyeler
```

Yaşı 30'un üzerinde olan Uyelerin bütün bilgilerini elde etmek için ise aşağıdaki gibi bir SQL cümleciği kullanılabilir.

```
SELECT * FROM Uyeler WHERE Yaş > 30
```

Buna göre bir OleDbCommand nesnesi aşağıdaki gibi oluşturulabilir.

```
string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=veri.
mdb";
OleDbConnection baglanti = new OleDbConnection(kaynak);
baglanti.Open();
string sorgu = "select Adı,Soyadı FROM Uyeler";
OleDbCommand komut = new OleDbCommand(sorgu,baglanti);
```

OleDbCommand nesnesinin CommandType özelliği ile komutun ne şekilde oluşturulacağı bildirilir. CommandType numaralwandırması ile bu özellik Text, StoredProcedure ya da TableDirect olacak şekilde ayarlanabilir. Varsayılan olarak bu özellik Text'tir. Bu özelliğin Text olması, komutun bir SQL cümleciği olduğunu göstermektedir.

2. İkinci yöntem OleDbCommand nesnesinin CommandType özelliğini TableDirect olacak şekilde ayarlamaktır. Bu şekilde hazırlanan OleDbCommand nesnesi ile komutu çalıştırdıktan sonra bir tablonun tamamı elde edilir. Bu şekildeki bir OleDbCommand nesnesi aşağıdaki gibi oluşturulabilir.

```
string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=veri.
mdb";
OleDbConnection baglanti = new OleDbConnection(kaynak);
baglanti.Open();
OleDbCommand komut = new OleDbCommand("Uyeler",baglanti);
komut.CommandType = CommandType.TableDirect;
```

CommandType numaralandırması System.Data isim alanında bulunduğunu hatırlatmak gerekir. (Uyeler, veritabanındaki tablonun adıdır.)

3. Üçüncü yöntem olarak veritabanlarındaki Stored Procedure dediğimiz yordamları çalıştırmak ve bu yordamlara parametre geçmek için kullanılan yöntemi inceleyeceğiz. Stored Procedure'lerin kullanımı ve hazırlanması bu kitabın konularından olmadığı için detayları bu kitapta anlatılmayacaktır. Veritabanındaki Stored Procedure yordamlarına parametre göndermek için OleDbCommand sınıfının Parameters

özelliğinin Add() metodu kullanılabilir. Aşağıda Stored Procedure kullanarak OleDbCommand nesnelerini nasıl oluşturabileceğimize bir örnek verilmiştir.

```
string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=veri.mdb";
OleDbConnection baglanti = new OleDbConnection(kaynak);
baglanti.Open();
OleDbCommand komut = new OleDbCommand("Yordam",baglanti);
komut.CommandType = CommandType.StoredProcedure;
komut.Parameters.Add("@Yaş","30");
```

Add() metodu ile Yordam isimli Stored Procedure yordamındaki Yaş parametresine 30 değeri gönderilmektedir.

OleDbCommand nesnelerinin oluşturulma yöntemlerinden en çok kullanılanı SQL cümleciğidir. Zaten SQL cümleciğini istediğimiz gibi düzenleyerek bir tablonun tamamını da elde edebiliriz.

Şimdi de OleDbCommand ile temsil edilen sorguların nasıl çalıştırılacağını inceleyelim. OleDbCommand üzerinden bir komutu çalıştırmak için 4 ayrı metot kullanılabilir. Bu metotlar aşağıdaki gibi sıralanmıştır.

1. `ExecuteNonQuery()` : Bu metodu genellikle bir veritabanındaki kayıtları değiştirmek, silmek ya da yeni bir kayıt eklemek istediğimizde kullanırız. Bu metodun geri dönüş değeri komutun çalıştırıldıktan sonra etkilediği kayıt sayısıdır. Bir tabloya yeni bir kayıt eklemek için aşağıdaki SQL cümleciği kullanılabilir.

```
INSERT INTO Uyeler (Sütünlar) VALUES (SütunDeğerleri)
```

Aşağıda, veritabanındaki bir tabloya yeni bir kaydın nasıl eklendiği gösterilmiştir.

```
using System;
using System.Data;
using System.Data.OleDb;

public class AccessOleDb
{
        public static void Main()
        {
```

```
            string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data
                                                Source=veri.mdb";

        OleDbConnection baglanti = new OleDbConnection(kaynak);

            baglanti.Open();

            string sorgu =@"INSERT INTO Uyeler
            (Adı,Soyadı,Yaş,KullanıcıAdı,Şifre,Puan)
            VALUES ('Mehmet','Demir',25,'demirm','SADEGT',256)";

        OleDbCommand komut = new OleDbCommand(sorgu,baglanti);

            komut.ExecuteNonQuery();

            baglanti.Close();
    }
}
```

Programı derleyip çalıştırdığınızda veritabanındaki Uyeler tablosunun aşağıdaki gibi değiştiğini göreceksiniz.

11	Cezmi	Kara	54	karac	op989898	325
13	Mehmet	Demir	25	demirm	SADEGT	256

Veritabanında bazı kriterlere uyan kayıtları değiştirmek için de bu metot kullanılabilir. Örneğin şifresi "ASHYG" olan bir üyenin şifresini "9875" olarak değiştirmek için aşağıdaki SQL cümleciği kullanılabilir.

```
UPDATE Uyeler Set Şifre='9875' where Şifre='ASHYG'
```

Aynı şekilde bir tablodan belirli kriterlere uyan kayıtları silmek için DELETE komutu kullanılır. Uyeler tablosunda şifresi "9875" olan üyenin kaydını silmek için aşağıdaki SQL cümleciği yazılabilir.

```
DELETE FROM Uyeler where Şifre='9875'
```

ExecuteNonQuery() metodunun geri dönüş değeri komutun çalıştırılması sonucu elde edilen kayıt sayısı, silinen kayıt sayısı ya da değiştirilen kayıt sayısı olabilir.

2. `ExecuteScalar()` : Bu metot ile veritabanından tek bir değer elde edilir. Örneğin bir tablodaki kayıt sayısını elde etmek için kullanılan COUNT komutunun kullanıldığı SQL cümleciklerinin çalıştırılması ExecuteScalar() metodu ile yapılabilir. Aşağıdaki programda Uyeler tablosundaki kayıt sayısının nasıl elde edildiği görülmektedir.

```
using System;
using System.Data;
using System.Data.OleDb;

public class AccessOleDb
{
        public static void Main()
        {
            string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data
                                                Source=veri.mdb";

        OleDbConnection baglanti = new OleDbConnection(kaynak);
            baglanti.Open();

            string sorgu ="SELECT COUNT(*) FROM Uyeler";

        OleDbCommand komut = new OleDbCommand(sorgu,baglanti);

            int kayıt = (int)komut.ExecuteScalar();

            Console.WriteLine("Toplam Kayıt Sayısı: " + kayıt);

            baglanti.Close();
        }
}
```

ExecuteScalar metodunun geri dönüş değeri object türünden olduğu için kayıt sayısını elde etmek için int türüne dönüştürme işlemi yapıldı. Programı çalıştırdığınızda Uyeler tablosundaki toplam kayıt sayısının ekrana yazıldığını göreceksiniz.

3. `ExecuteReader()` : Bu metodun SQL cümleciğinin SELECT komutu içerdiği durumlarda kullanıldığı sıklıkla görülür. SELECT komutu ile bir veritabanından bir sorgu yapılarak bir kayıt kümesi elde edilir. Elde edilen bu kayıt kümesi OleDbDataReader nesnesine aktarılır. OleDbDataReader ile bu kayıt kümesinden çeşitli yollardan veriler elde edilebilir. ExecuteReader() metoduna bir örnek vermeden önce OleDbDataReader sınıfının özelliklerini inceleyelim.

OleDbDataReader nesnesi ile bir kayıttan sadece ileriye doğru ilerleme yapılır. Yani bir kayıt kümesinden bir kaydı okuduktan sonra tekrar geriye dönüp eski kayıtları okumak mümkün değildir. OleDbDataReader nesnelerini new operatörü ile doğrudan oluşturamayız. Bunun yerine OleDbCommand nesnesinin ExecuteReader() metodunun geri dönüş değeri kullanılır.

OleDbDataReader sınıfında bildirilen iki indeksleyici ile sorgu sonucu elde edilen kayıt kümesindeki her sütuna ilişkin veriler elde edilebilir. Bu indeksleyicilerin kullanımı aşağıda verilmiştir.

```
DataReader[ 1] ;

DataReader[ "Adı"] ;
```

Bizim oluşturduğumuz Uyeler tablosundaki "Adı" sütunu 1. sütun olduğu için yukarıdaki her iki ifade de aynı değeri verecektir. Ayrıca OleDbDataReader sınıfındaki GetInt32(), GetByte(), GetString() metotları ile de özel değerler elde edilebilir. Eğer veritabanındaki sütunların veri türlerini biliyorsak bu metotları kullanmak yüksek performans sağlar, zira yukarıda bahsedilen indeksleyicilerin geri dönüş değeri object türünden olduğu için boxing işlemi ve unboxing işlemi gerektirir. Eğer sütunların veri türlerini bilmiyorsak bile indeksleyiciyi string yerine int türü ile kullanmak performans açısından daha avantajlıdır.

Aşağıdaki örnekte ExecuteReader() metodu kullanılarak OleDbDataReader nesnesi ile Uyeler tablosundaki uyelerin adı, soyadı ve yaşı ekrana yazdırılıyor.

```
using System;
using System.Data;
using System.Data.OleDb;

public class AccessOleDb
{
        public static void Main()
        {
                string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data
                                                Source=veri.mdb";
            OleDbConnection baglanti = new OleDbConnection(kaynak);

                baglanti.Open();

                string sorgu ="SELECT * FROM Uyeler";
            OleDbCommand komut = new OleDbCommand(sorgu,baglanti);

                OleDbDataReader ord = komut.ExecuteReader();
```



```
                Console.WriteLine("{ 0,-10} { 1,-10} { 2,-10} ",
                                        "Adı","Soyadı","Yaş");
                Console.WriteLine("--------------------------");

                while(ord.Read())
                {
                        Console.WriteLine("{ 0,-10} { 1,-10 } { 2,-10} ",
                                        ord[ 1] ,ord[ 2] ,ord[ 3] );
                }

                ord.Close();
                baglanti.Close();
        }
}
```

Programdaki Read() metodunun kullanımına dikkat edin. Read() metodunun geri dönüş değeri eğer OleDbDataReader ile okuma yapılabildiyse true değerindedir. Eğer okuma yapılmadıysa false değerindedir. Read() metodu her çağrıldığında OleDbDataReader nesnesindeki kayıt işaretçisi bir artırılır. İşaretçi son kaydı gösterdikten sonra Read() metodu ile artık okuma yapılamayacaktır. Programı derleyip çalıştırdığınızda aşağıdaki ekran görüntüsünü elde etmelisiniz.

```
Visual Studio .NET Command Prompt

H:\cskitap\kitap\Bolum14\kod>OleDbDataReader
Adi        Soyadi     Yas
----------------------------
Sefer      Algan      23
Gökcen     Yildirim   22
Inan       Erdem      23
Ihsan      Algan      20
A.Faruk    Nacaroglu  22
Ahmet      Hançer     35
Cemal      Öztürk     26
Ebru       Salmaner   28
Fatih      Azakli     26
Melih      Kalafat    22
Cezmi      Kara       54

H:\cskitap\kitap\Bolum14\kod>
```

OleDbDataReader ve OleDbConnection nesneleri ile işimiz bittiğinde Close() metodu ile bu nesneler için ayrılmış kaynakları iade etmeyi unutmayın.

4. `ExecuteXmlReader()` : Bu metot sadece SqlCommand ile uygulanabilir. SQL Server, verileri XML şeklinde elde etmek için çeşitli komutlar sağlamıştır. Bu komutlar SQL Server'a özel olduğu için OLEDB API'lerini destekleyen veritabanlarında bu komutların olmasını bekleyemeyiz. Bu metodun geri dönüş değeri ise ileride göreceğimiz XmlReader sınıfı türündendir.

OleDbDataReader ve OleDbCommand nesnelerinin burada değinmediğimiz birçok metodu ve özelliği vardır. Bütün bu özelliklere ve metotlara ulaşmanın en kolay yolu şüphesiz MSDN kütüphanesidir.

DataSet, DataAdapter, DataTable, DataRow, DataColumn

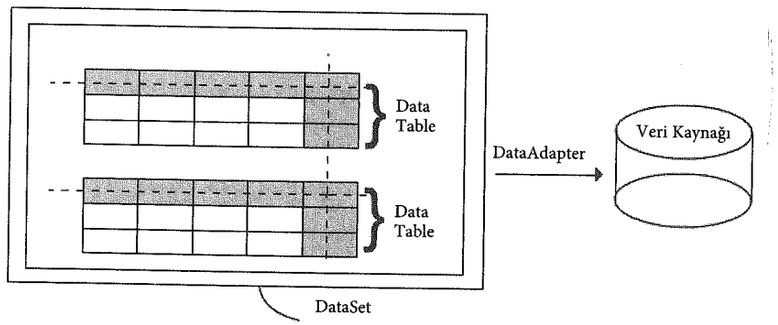
DataSet ve bir DataSet sınıfının içindeki yapıları temsil eden DataTable, DataRow ve DataColumn sınıfları ADO.NET'in veri erişim modeline getirdiği yenilikleri içerir.

DataSet, veri kaynağından bağımsız olarak veriler üzerinde işlem yapmamızı ve gerektiğinde DataAdapter yardımıyla DataSet üzerindeki verileri tekrar veri kaynağına aktarmamızı sağlar.

Bir DataSet ile, veri kaynağından bağlantı kesilse bile DataSet içindeki verilerle çalışabiliriz. Konunun başında bahsedilen offline çalışma DataSet ile olmaktadır. DataSet bir veri kaynağının adeta şablonu gibidir. Bir DataSet içinde bir ya da daha fazla tablo olabilir. Bu tablolar System.Data isim alanındaki DataTable sınıfı ile temsil edilir. Tabloların yapısında ise kayıtları ve sütunları temsil eden DataRow ve DataColumn sınıfları mevcuttur.

Dataset nesnelerine veri kaynağından bilgi alıp doldurmak için DataAdapter sınıfı kullanılır. DataAdapter sınıfı System.Common isim alanında bulunur. Biz bu bölümde OLDB veri kaynaklarına erişmek için DataAdapter sınıfından türemiş olan OleDbDataAdapter sınıfını kullanacağız. Ayrıca System.Data.SqlClients isim alanında SqlDataAdapter sınıfının bulunduğunu bilmeniz gerekir. DataSet ve DataAdapter nesnelerinin OLEDB ve SQL Server için sunduğu arayüzler aynıdır. Burada işlediğimiz OleDbDataAdapter ile aslında SqlDataAdapter sınıfını da incelemiş oluyoruz.

Bahsedilen bu sınıfların birbirleriyle olan ilişkisi aşağıdaki şekilde gösterilmiştir.



Yukarıdaki şekilde de görüldüğü üzere veri kaynağı ile DataSet nesneleri arasındaki bağlantı DataAdapter nesnesi ile olmaktadır.

Şimdi sırasıyla DataTable, DataColumn ve DataRow sınıflarını inceleyelim.

DataTable sınıfı bir veri kaynağındaki tablolara benzer. Yani bir DataTable nesnesinde sütunlar ve verileri içeren kayıt satırları bulunmaktadır. Bir DataSet nesnesi içinde birden fazla DataTable olabileceği için farklı DataTable nesneleri arasında çeşitli ilişkiler kurulabilir. DataTable nesnelerinin `Columns` özelliği ilgili tablodaki sütunları bir DataColumn koleksiyonu şeklinde verir. Aynı şekilde `Rows` özelliği de tablodaki satırları bir DataRow koleksiyonu şeklinde verir.

DataColumn sınıfı, bir tablodaki sütunları temsil eder. DataColumn ile temsil edilen bir sütunun veri türü C#'taki herhangi bir temel veri türü olabilir. DataColumn nesneleri ile bir sütunun çeşitli özellikleri değiştirilebilir. Aşağıda bir DataColumn nesnesinin oluşturulması ve bazı özelliklerinin ne anlama geldiği verilmiştir.

```
DataColumn dc = new DataColumn("UyeID",typeof(int));
//Sütunun boş değer kabul edip etmeyeceğini belirtir.
dc.AllowDBNull = false;
//Yeni bir kayıt eklendiğinde bu sütunun değerinin
//otomatik olarak artıp artmayacağını verir.
dc.AutoIncrement = true;
//Otomatik olarak artacak değerin başlangıç değeri
dc.AutoIncrementSeed = 10;
//Sütunun otomatik olarak artan değerinin
//her kayıt eklemede ne kadar artacağını belirtir.
dc.AutoIncrementStep = 1;
//Sütunun hangi isim ile temsil edildiğini belirtir.
dc.Caption = "UyeID";
//Sütunun veri tipini bildirir.
dc.DataType = typeof(string);
//Bir sütundaki verinin varsayılan değerini belirler.
dc.DefaultValue = (object)"*";
//Sütunun içerebileceği maximum karakter sayısını belirler.
dc.MaxLength = 10 ;
//Sütunun bir sütun koleksiyonunda kaçıncı sırada
//olduğunu verir. Sadece okunabilir bir özelliktir.
Console.WriteLine(dc.Ordinal);
//Sütunun tekil değerler içerip içermeyeceği belirtilir.
dc.Unique = true;
//Sütunun hangi tabloya ait olduğunu verir.
dc.Table;
```

DataRow nesnesi ise bir tablodaki satırları temsil eder. DataRow sınıfının birçok metodu vardır. Bu metotların tamamı burada anlatılmayacaktır. Burada daha çok bir DataRow nesnesini indeksleyici ile kullanıp kayıtlardaki verileri nasıl elde edebileceğimizi inceleyeceğiz.

Bir DataSet nesnesinin içini veri kaynağındaki bilgilerle doldurmak için DataAdapter sınıfının Fill() metodu kullanılır. Bu metot parametre olarak bir DataSet nesnesi almaktadır. Aşağıdaki örnekte şimdiye kadar anlatılan sınıfları kullanarak veritabanındaki tablo ile ilgili bilgileri nasıl elde ettiğimizi inceleyebilirsiniz.

```
using System;
using System.Data.Common;
using System.Data;
using System.Data.OleDb;

public class AccessOleDb
{
        public static void Main()
        {
                string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data
                                                Source=veri.mdb";
            OleDbConnection baglanti = new OleDbConnection(kaynak);

                baglanti.Open();
                string sorgu ="SELECT * FROM Uyeler";

                OleDbDataAdapter adp = new
                OleDbDataAdapter(sorgu,baglanti);

                DataSet ds = new DataSet();
                adp.Fill(ds, "Uyeler");

              foreach(DataColumn dc in ds.Tables["Uyeler"].Columns)
                 Console.WriteLine("Sütun adı: {0}, Veri tipi: {1}",
                        dc.Caption,dc.DataType.ToString());

                Console.WriteLine("\nUyeler Tablosu\n");

              foreach(DataColumn dc in ds.Tables["Uyeler"].Columns)
                        if (dc.Caption.Length > 10)
                                Console.Write("{0,-10}",
                                dc.Caption.Substring(0,5) + "...");
                        else
```



```
                                Console.Write("{0,-10}", dc.Caption);

                Console.WriteLine();

                for(int i=0; i<65; ++i)
                        Console.Write("-");

                Console.WriteLine();

                foreach(DataRow dr in ds.Tables["Uyeler"].Rows)
                {
                        object[] dizi = dr.ItemArray;

                        foreach(object i in dizi)
                                Console.Write("{0,-10}",i.ToString());

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

Programı derleyip çalıştırdığınızda aşağıdaki ekran görüntüsünü elde etmelisiniz.

```
Visual Studio .NET Command Prompt

H:\cskitap\kitap\Bolum14\kod>DataSet
Sütun adi: KullaniciID, Veri tipi: System.Int32
Sütun adi: Adi, Veri tipi: System.String
Sütun adi: Soyadi, Veri tipi: System.String
Sütun adi: Yas, Veri tipi: System.Int32
Sütun adi: KullaniciAdi, Veri tipi: System.String
Sütun adi: Sifre, Veri tipi: System.String
Sütun adi: Puan, Veri tipi: System.Int32

Uyeler Tablosu

Kulla...  Adi       Soyadi    Yas       Kulla...  Sifre     Puan
-------------------------------------------------------------------
1         Sefer     Algan     23        algans    SJKAMSJN  251
2         Gökcen    Yildirim  22        y_gokcen  kkkoI569  120
3         Inan      Erdem     23        erdemi    9089yhYL  99
4         Ihsan     Algan     20        madenci   56569874  265
5         A.Faruk   Nacaroglu 22        yakamoz   yaKAYTA   25
6         Ahmet     Hançer    35        hancer    OKUHGTF   0
7         Cemal     Öztürk    26        mcemalo   2365987   14
8         Ebru      Salmaner  28        ebruli    ndbdbOI   130
9         Fatih     Azakli    26        afatih    58987458  198
10        Melih     Kalafat   22        kalafatme IKJU7898  25
11        Gezmi     Kara      54        karac     op989898  325

H:\cskitap\kitap\Bolum14\kod>
```

Bu temel işlemlerden sonra bir DataSet üzerinde yapılan değişikliklerin veritabanı üzerinde nasıl yenileneceğini inceleyelim.

Bir DataSet nesnesindeki bir tabloya yeni bir kayıt eklemek için DataTable sınıfının NewRow() metodu kullanılır. DataSet nesnesindeki tabloları elde etmek için ise DataSet nesnesinin Tables özelliğini kullanabiliriz. Tables özelliğini bir indeksleyici ile kullanarak DataSet nesnesine eklenen tabloları tek tek elde edebiliriz. Yeni oluşturulan bu kaydın DataSet nesnesine aktarılması için DataSet nesnesindeki Rows koleksiyonunun Add metodu kullanılabilir. Son olarak DataSet nesnesinde yapılan bu değişiklerin veri kaynağında da güncellenmesi için DataAdapter nesnesinin Update() metodunu kullanmalıyız. Aşağıda Uyeler tablosuna yeni bir kayıt eklenmektedir.

DataAdapter nesnesinin Update() metodu çağrıldığında DataSet'te bulunan verilerin veritabanına ne şekilde kaydedileceğini, DataAdapter nesnesinin InsertCommand özelliği belirlemektedir. InsertCommand nesnesinin CommandText özelliği ile yeni kayıt eklemek için gerekli SQL cümleciği yazılmaktadır. Connection özelliğine ise bağlantı nesnesi atanmaktadır. Parameters özelliğinin Add() metodu ile veritabanındaki her bir sütunun veri türü ve hangi parametreye denk geldiği belirtilmektedir.

```
using System;
using System.Data.Common;
using System.Data;
using System.Data.OleDb;

public class AccessOleDb
{
        public static void Main()
        {
                string kaynak="Provider=Microsoft.Jet.OLEDB.4.0;Data
                                                Source=veri.mdb";
            OleDbConnection baglanti = new OleDbConnection(kaynak);

                baglanti.Open();

                string sorgu ="SELECT * FROM Uyeler";

                OleDbDataAdapter adp = new
                OleDbDataAdapter(sorgu,baglanti);


                DataSet ds = new DataSet();
                adp.Fill(ds, "Uyeler");

                DataRow r = ds.Tables["Uyeler"].NewRow();

                r["Adı"] = "Seda";
```



```
                r["Soyadı"] = "Karan";
                r["Yaş"] = 26;
                r["KullanıcıAdı"] = "karans";
                r["Şifre"] = "SOIJHYP";
                r["Puan"] = 320;

                adp.InsertCommand = new OleDbCommand();
                adp.InsertCommand.Connection = baglanti;
                adp.InsertCommand.CommandText ="Insert Into Uyeler
                 (Adı,Soyadı,Yaş,KullanıcıAdı,Şifre,Puan) values
                                                (@Adı,@Soyadı,@
Yaş,
                                        @KullanıcAdı,@Şifre,@Puan)";

                 adp.InsertCommand.Parameters.Add("@Adı",
                                        OleDbType.VarChar ,50,"Adı");
                adp.InsertCommand.Parameters.Add("@Soyadı",
                                    OleDbType.VarChar,50,"Soyadı");
                adp.InsertCommand.Parameters.Add("@Yaş",
                                        OleDbType.Integer ,50,"Yaş");
        adp.InsertCommand.Parameters.Add("@KullanıcıAdı",
                                OleDbType.VarChar,50,"KullanıcıAdı");
        adp.InsertCommand.Parameters.Add("@Şifre",
                                OleDbType.VarChar,50,"Şifre");
                        adp.InsertCommand.Parameters.Add("@Puan",
                                OleDbType.Integer,50," Puan");

                ds.Tables["Uyeler"].Rows.Add(r);

                adp.Update(ds,"Uyeler");

                baglanti.Close();
        }
}
```

Programı derleyip çalıştırdığınızda Uyeler tablosuna aşağıdaki gibi yeni bir kayıt eklenmiş olması gerekir.

8	Ebru	Salmaner	28	ebruli	ndbdbOI	130
9	Fatih	Azaklı	26	afatih	58987458	198
10	Melih	Kalafat	22	kalafatme	IKJU7898	25
11	Cezmi	Kara	54	karac	op989898	325
16	Seda	Karan	26	karans	SOIJHYP	320

DataSet nesnesinden bir kayıt silmek istendiğinde DataRow nesnesinin Delete() metodu kullanılabilir. Silinen kayıtların DataAdapter ile veritabanında güncellenmesi için DataAdapter nesnesinin DeleteCommand nesnesi tanımlanmalıdır. Aynı işlemler DataSet'teki bir kaydın değiştirildiği durum için de geçerlidir. Bunun için ise UpdateCommand nesnesi tanımlamalıyız. Her iki nesne de DataAdapter nesnesinin örnekte de görüldüğü gibi birer özelliğidir.

DataSet nesnelerinin veritabanında güncellenmesi için yapmamız gereken en önemli iş, DataSet'te yapılan her değişikliğin DataAdapter ile nasıl bir ilişkisi olduğunu belirlemektir. Örneğin DataSet'e eklenen bir kayıt için DataAdapter nesnesinin InsertCommand nesnesinin CommandText özelliğini uygun bir SQL cümleciği ile belirtmemiz gibi. Diğer bir önemli nokta da DataSet'teki veri türlerinin tipi ile gerçek veritabanındaki sütunların tipi arasındaki uyumluluktur. Bu uyumluluğu sağlamak için Parameters'ın Add() metodunda ikinci parametre olarak OleDbType numaralandırıcısı ile uygun bir veri türü seçilmektedir.

ADO.NET 2.0 ve DbProviderFactory (Veri Sağlayıcı Bağımsız Erişim)

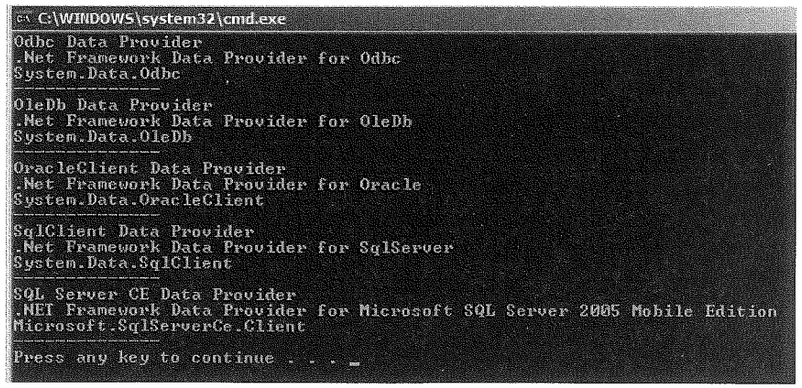
ADO.NET 2.0 ile birlikte veritabanı sağlayıcılarından bağımsız veri erişimi yapabilmekteyiz. Bu ADO.NET 2.0'a eklenen soyut nesneler ile yapılmaktadır. Daha önce de bahsettiğimiz gibi ADO.NET içinde farklı veritabanı sistemleri ile çalışan farklı nesneler bulunmaktadır. Örneğin **SqlConnection**, **OledbConnection**, **OracleConnection** gibi. Her bir nesne topluluğu sadece kendi veritabanı sisteminde verimli çalışmaktadır. Oledb standardını destekleyen bütün veritabanları ile Oledb ön eki ile başlayan (OleDbConnection, OleDbCommand...) nesneler kullanılmaktadır ancak bu durumda bir performans kaybı meydana gelmektedir. Veritabanı sistemlerine erişirken hem veritabanı bağımsızlığını sağlamak hem de bağlanma sırasında kendi sağlayıcılarını kullanarak performanstan ödün vermeyen yeni bir sistem geliştirilmiştir. Bu sistem literatürde **abstract factory** (*soyut fabrika*) denilen bir tasarım deseni ile gerçekleştirilmiştir.

Buna göre Connection, Command, Adapter gibi veritabanı bağımlı nesneler soyut sınıflar üzerinden işlem görecektir. SqlConnection, OleDbCommand gibi veri sağlayıcı bağımlı nesneler DbCommand, DbConnection gibi soyut nesnelerden türeyecektir. İstenilen sağlayıcı ile gerçek nesneleri üretecek bir **fabrika** (*factory*) modeli de oluşturularak veritabanı erişimi yapacak programcının veritabanı sağlayıcı hakkında fazla bilgi sahibi olmaması sağlanmaktadır. Yani bütün veritabanı bağımlı nesneler (connection, adapter, command, datareader, parameter) direkt olarak değil de bir fabrika tarafından oluşturulacaktır.

.NET Framework 2.0'da tanımlı olan soyut fabrikaları aşağıdaki gibi ekrana yazdırabiliriz. Dilediğimiz fabrikayı kullanarak istediğimiz veritabanı nesnelerini üretip kullanabiliriz.

```
using System.Data;
using System.Data.Common;
using System;

namespace providerfactories
{
    class Program
    {
        static void Main(string[] args)
        {
            DataTable dtProviders = DbProviderFactories.
GetFactoryClasses();
            foreach (DataRow drProvider in dtProviders.Rows)
            {
            for (int i = 0; i < dtProviders.Columns.Count -1; i++)
                {
                    Console.WriteLine(drProvider[i].ToString());
                }
            Console.WriteLine("--------------");
            }
        }
    }
}
```

Yukarıdaki programı derleyip çalıştırdığınızda aşağıdaki gibi bir ekran görüntüsü almalısınız.



Görüldüğü gibi .NET 2.0 içerisinde 5 adet **DbProviderFactory** nesnesi bulunmaktadır. İstenilen veri sağlayıcısı 3.parametre ile belirtilen **provider invariant name** (mesela *System.Data.SqlClient*) vasıtası ile aşağıdaki gibi alınabilir.

```
DbProviderFactory fact = DbProviderFactories.GetFactory
("System.Data.SqlClient");
```

Bu işlem yapıldıktan sonra fact nesnesi üzerinen **Connection**, **Command**, **Adapter** gibi veritabanı bağımlı nesneler elde edilecektir. Bu aşamadan sonra işlemlerin veritabanı bağımsız olabilmesi için bütün nesnelerin fact üzerinden **CreateXXX** ile başlayan metotlarla yapılması gerekir. Örneğin bir SQL Server üzerindeki tablodan veriler aşağıdaki gibi bir **DataReader** nesnesine aktarılabilir.

```
using System.Data;
using System.Data.Common;
using System;

namespace providerfactories
{
    class Program
    {
        static void Main(string[] args)
        {
            DbProviderFactory fact = DbProviderFactories.
GetFactory("System.Data.SqlClient");

            DbConnection con = fact.CreateConnection();
            con.ConnectionString = "Data Source=localhost\\
YUKON;user id=sa;pwd=1;initial catalog=Northwind;";
            con.Open();

            DbCommand cmd = fact.CreateCommand();
            cmd.Connection = con;
            cmd.CommandText = "select * from Customers";

            DbDataReader reader = cmd.ExecuteReader();

            while(reader.Read())
            {
                Console.WriteLine(reader[1].ToString() + " " +
reader[2].ToString());
            }
        }
    }
}
```


Factory nesnesi üzerindeki örneklerimizi çoğaltmak elbetteki mümkündür. Bu bölümdeki amacımız sadece .NET 2.0 ile birlikte gelen önemli bir yenilikten bahsetmekti. Metotlar ve sınıflar üzerinde oynayarak diğer karmaşık uygulamalarınızı şimdiden geliştirebileceğinizi düşünüyorum.

Bu kısımda ADO.NET kütüphanesi ile temel veritabanı işlemlerinin nasıl gerçekleştirildiğini inceledik. ADO.NET'in yetenekleri anlattıklarımızla sınırlı değildir. Örneğin bir DataSet nesnesine birden fazla tablo ekleyip, veritabanından bağlantımızı kesip, tablolar arasında istediğimiz gibi ilişki kurabiliriz. Ve az önce anlattığımız Update() metodu ile veritabanını DataSet nesnesindeki verilerle güncelleyebiliriz. Tabi güncellemeyi yapacak DataAdapter nesnesinin ilgili komut ayarlarının yapılmış olması gerekir. Bu kısımda anlatılan temel işlemlerin dışındaki kavramlar ilişkisel veritabanları ile çalışma becerisi gerektirmektedir. İlişkisel veritabanlarının hazırlanması veya programlama ile yönetilmesi C# dilinin anlatıldığı bu kitabın bir konusu değildir. ADO.NET ile ilgili yazılmış geniş içerikli kaynaklar veya MSDN kütüphanesi kalan konuları öğrenmeniz için gerekli araçlardır.

XML İşlemleri ve System.Xml

.NET platformunda XML'in önemi büyüktür. Çünkü hem Framework'ün konfigürasyon dosyalarının çoğu XML tabanlıdır hem de ADO.NET gibi büyük bir kütüphane XML ile iç içe çalışmaktadır. XML yeni neslin iletişim dilidir. Günümüzde farklı platformlardaki sistemler birbirleri ile haberleşirken XML dilini kullanıyorlar. Daha doğrusu XML bu haberleşme için bir araç olmuştur. Bunda elbette ki XML dilinin herhangi bir platforma özgü olmaması ve W3C standartları tarafından desteklenmesinin büyük payı vardır. Biz bu başlık altında XML ve diğer XML bağlantılı standartlara değinmeyeceğiz, bu kısımdaki amacımız var olan XML belgelerini C# ile nasıl düzenleyebileceğimiz ve XML belgelerindeki bilgileri nasıl elde edebileceğimizi görmektir.

XML bir programlama dili değildir. XML belgelerini, biçimlendirilmiş metin tabanlı veri depolama işine yarayan belgeler şeklinde nitelendirebiliriz. Veritabanları, elde olan bilgileri çeşitli biçimlerde saklamak için kullanılan gelişmiş sistemlerdir. Bazı veritabanları sunucu – istemci mimarisini desteklerken bazı veritabanları dosya sisteminde normal bir dosyaymış gibi davranır. Dosya sisteminde bulunan veritabanları hızlı olmakla beraber günümüz şartlarında kullanışlı değildir. Çünkü dosya sisteminden aynı anda binlerce kişinin aynı dosyayı okumaya çalışması sistemin çökmesine sebep olabilmektedir. Bunun için verileri daha düzenli ve daha sistematik bir şekilde işleyen veritabanı sunucuları kullanılır. Microsoft'un SQL Server sunucusu ve Oracle veritabanı yönetim sistemleri bunlara verilebilecek en önemli iki örnektir.

Veritabanı sunucuları daha piyasada yokken metin tabanlı bilgi saklama yöntemleri kullanılırdı. Bir önceki başlık altında oluşturduğumuz MS Access veritabanındaki Uyeler tablosuna benzer bir tablo metin tabanlı bir tablo olarak aşağıdaki gibi yazılıyordu:

```
Veri.txt
Sefer, Algan, POLJK, algans, 25, 698
Ali, Köse, olLmK, köse, 20, 152
Veli, Çınar, 52dsJK, cinar, 15, 15
.......
```

Yukarıdaki her bir satır veritabanındaki bir kaydı göstermektedir. Her bir sütun ise virgül işareti ile birbirinden ayrılmış durumdadır. Bu dosyadan kayıt ve sütunlarla ilgili bilgileri elde edebilmek için dosyayı açıp çeşitli ayıklama işlemleri yapmamız gerekir. Fakat XML ile metin tabanlı veri saklama sistemine büyük kolaylık gelmiştir. Normal veritabanlarındaki gibi verilere çeşitli isimler verilir ve gerektiğinde bu verinin ismi kullanılarak asıl veri dosyadan çekilir. Tabi XML dosyaları ile ilgili yazma ve okuma gibi temel işlemleri yapabilmek için sistemimizde XML motoru dediğimiz kütüphanelerin olması gerekir. Eğer yoksa XML belgeleri ile ilgili işlem yapmak bir hayli zor olacaktır. XML ayıklayıcı bir defa yazıldıktan sonra XML dosyalarını normal bir veritabanı gibi kullanabiliriz. XML, W3C standartlar komitesi tarafından desteklendiği için hemen bütün platformlar XML desteği vermektedir. Bir platformun XML desteği vermesinden kastımız, XML belgelerini işleyebilmek için gerekli olan motorun sağlanmasıdır. XML'in standart olmasından dolayı XML tabanlı birçok uygulama alanı gelişmiştir. Örneğin farklı sistemler arasında haberleşmeyi sağlayan Web Servisleri bu uygulama alanlarından en önemlisidir. Web Servisleri sayesinde internet protokolü dediğimiz HTTP üzerinden, XML yardımıyla bir sunucudaki metodu çalıştırıp metodun geri dönüş değerini, aradaki çeşitli katmanları kullanarak XML formatında elde edebiliriz. Bu kitapta web servisleri ve XML standartları anlatılmayacaktır. Dolayısıyla sizin giriş düzeyinde XML söz dizimine hakim olmanız beklenmektedir. Eğer XML ile ilgili daha detaylı çalışmak istiyorsanız Zafer Demirkol'un Pusula Yayıncılık'tan çıkan XML kitabını edinmenizi tavsiye ederim.

XML Belgeleri

Şimdi isterseniz basit bir XML dosyası oluşturup XML belgesindeki çeşitli elemanları hatırlayalım.

```
<?xml version="1.0" encoding = "ISO-8859-9"?>
<yemekler>
      <menu adı="Sultan Sofrası">
            <corba>Mercimek</corba>
            <ana_yemek>Döner</ana_yemek>
            <tatlı>Kadayıf</tatlı>
      </menu>
```



```
      <menu adı="Çilingir Sofrası">
            <corba>Domates Çobası</corba>
            <ana_yemek>Balık</ana_yemek>
            <tatlı>Lokma</tatlı>
      </menu>
</yemekler>
```

En sade hali ile bir XML belgesi yukarıdaki gibi oluşturulabilir. Belgeden de görüldüğü üzere her veri, kapanan ve açılan etiketler arasına yazılmıştır. Belgenin ilk satırında, çeşitli bildirimler yapılmaktadır. Bu bildirimler versiyon bilgisi ve karakter kodlamasıdır. Bu bildirimler bu kadarla sınırlı değildir. Her XML belgesinin mutlaka bir kök elemanı olmalıdır. Bu örnekte kök eleman

```
<yemekler>
.....
</yemekler>
```

etiketi ile belirtilen elemandır. Kök elemanı olmayan bir XML belgesi geçerli değildir. Diğer bütün elemanlar kök elemanının içinde bulunur. Tıpkı bir ağaç yapısı gibi, yemekler elemanında iki adet menu elemanı bulunmakta. menu elemanı ise farklı yemek çeşitlerinden oluşan 3 ayrı elemandan oluşmaktadır.

XML belgelerindeki veriler açılan ve kapanan etiketler arasına yazılır. Örneğin

```
<ana_yemek>Balık</ana_yemek>
```

etiketinde "Balık" bilgisinin bir ana yemek olduğu söylenmektedir. Ayrıca her etiket için farklı nitelikler de tanımlanabilir. Örneğin

```
<menu adı="Sultan Sofrası">
```

ifadesinde menu elemanının "adı" isminde bir niteliği vardır. Bu niteliğin değeri ise "Sultan Sofrası" şeklindedir.

Ana hatlarıyla bir XML belgesini inceledik. Şimdiki amacımız C# ile bir XML belgesini okuyup verileri elde etmektir. Birazdan inceleyeceğimiz sınıfları gördükten sonra XML belgeleri ile çalışmanın ne kadar kolay olduğunu göreceksiniz.

XML sözdizimi ve kurallar hakkında daha fazla bilgi edinmek istiyorsanız diğer XML kaynaklarınından faydalanmanız gereklidir, zira bu kısımda XML dili ilgili daha fazla ayrıntılı bilgi verilmeyecektir.

Hazırladığımız bu ilk XML belgesinin ağaç şeklinde nasıl bir yapı sunduğunu daha iyi görmek için bu belgenin Internet Explorer ile nasıl bir çıktı verdiğine bakalım.

Çıktıdan da görüldüğü üzere her bir eleman iç içe kümelenerek ağaç şeklinde bir yapı oluşturulmuştur.

System.Xml İsim Alanı

XML belgeleri üzerinde işlem yapmak için geliştirilmiş sınıflar System.XML isim alanında bulunur. Bu isim alanında birçok sınıf vardır. Bu sınıflardan sadece önemli olanlarını inceleyeceğiz. Kullanma ihtimalinizin yüksek olduğu sınıflar aşağıda verilmiştir.

`XmlReader`: XML belgelerinden verileri okumamız için tasarlanmış bir sınıftır. Tamponlama mekanizması ile çalışmamaktadır. Yani veriler direkt dosya sisteminden alınır. (Sadece ileri yönlü bir okuma yapılabilir.)

`XmlWriter`: Bir akıma ya da dosya sistemine XML verilerini kaydetmek için kullanılır. (Sadece ileri yönlü bir okuma yapılabilir.)

`XmlTextReader`: XML belgelerinden sadece ileri yönde okuma yapabilir ve XML verilerine bir akım (stream) gibi erişmemizi sağlar.

`XmlTextWriter`: XML akımlarını dosya sisteminde oluşturmak için kullanılır.

`XmlNode`: XML belgesindeki tek bir düğümü temsil etmek için kullanılır.

`XmlNodeReader`: Bu sınıf okumak için bir düğüm bilgisini kaynak olarak kullanır.

`XmlDocument`: Bir XML belgesinin belleğe bir ağaç yapısı şeklinde yüklenmesini ve bu ağaç üzerinde çeşitli işlemler yapılabilmesini sağlar.

System.Xml isim alanındaki diğer türlere de konu içinde ayrıca değinilecektir.

Şimdi XML belgelerinden okuma yapan ve XML belgelerine yazma yapan XmlTextWriter ve XmlTextReader sınıflarını inceleyelim. Öncelikle her iki sınıfın da özet sınıf olduğunu belirtelim. Bu sınıflar sırasıyla XmlWriter ve XmlReader sınıflarından türemiştir. XmlTextWriter ve XmlTextReader sınıfları XML belgesi ile bir akım gibi çalıştığı için daha yüksek performans sağlanır. Bu iki grup sınıfın yaptıkları işler hemen hemen aynıdır. Bu yüzden biz burada XmlTextReader ve XmlTextWriter sınıflarını inceleyeceğiz.

XmlTextReader, bir XML belgesinden okuma yapmamızı sağlar. Bir XmlTextReader nesnesi oluşturmanın birçok yolu vardır. Yani XmlTextReader sınıfının birçok yapıcı metodu vardır. Bu yapıcı metotlardan en sık kullanılan üçü aşağıda verilmiştir.

`XmlTextReader(string dosya_yolu)`: Bu yapıcı metot ile parametrede belirtilen XML belgesine ilişkin bir nesne oluşturulur. Örnek:

```
string dosya = "yemekler.xml";
XmlTextReader = new XmlTextReader(dosya);
```

`XmlTextReader(Stream akım)`: Oluşturulan nesne, parametre ile belirlenen Stream türünden bir nesneye ilişkindir. Stream nesnesi bir FileStream olabileceği gibi NetworkStream de olabilir. Örnek:

```
string dosya = "yemekler.xml";
FileStream fs = new FileStream(dosya, FileMode.Open)
XmlTextReader = new XmlTextReader(fs);
```

`XmlTextReader(XmlTextReader text_reader)`: Bu yapıcı metot ile XmlTextReader nesnesine ilişkin yeni bir nesne oluşturulur.

XmlTextReader sınıfının Read() metodu XML belgesinden okuma yapar. Read() metodu her çağrıldığında XML belgesindeki bir sonraki düğüm okunur. Dosyanın sonuna gelindiğinde Read() metodu okuma yapamayacağı için false değerine dönecektir. Read() metodu her okuma yaptığı iteresyonda yeni bir düğüm üretir. XmlTextReader sınıfındaki çeşitli özellikler bu düğüme ilişkindir. Şimdi yapıyı daha iyi anlamak için aşağıdaki gibi bir XML dosyası oluşturun.

yemekler.xml

```
<?xml version="1.0" encoding="ISO-8859-9" ?>
<yemekler>
        <menu adı="Sultan Sofrası">
                <corba>Mercimek</corba>
                <yemek>
                        <ana_yemek>Döner</ana_yemek>
                        <ara_yemek>Börek</ara_yemek>
                </yemek>
                <tatlı>Kadayıf</tatlı>

        </menu>
        <menu adı="Çilingir Sofrası">
                <corba>Domates</corba>
                <yemek>
                        <ana_yemek>Alabalık</ana_yemek>
```

```
                <ara_yemek>Kızartma</ara_yemek>
            </yemek>
            <tatlı>Lokma</tatlı>
        </menu>
</yemekler>
```

Bu Xml belgesindeki her bir satır bir düğümü ifade etmektedir. Ancak her düğümün farklı özellikleri vardır. Örneğin bazı düğümler sonlandırma düğümleri iken bazıları başlangıç düğümleridir. Bazı düğümlere ait nitelik bilgileri varken bazılarında yoktur. Ayırca bazı düğümlerin değeri de yoktur. Örneğin menu elemanı içinde sadece diğer elemanlar vardır. Ayrı bir veri türü olarak herhangi bir veri bulundurmamaktadır. Düğümlerin bütün bu özelliklerini XmlTextReader sınıfını çeşitli özellikleri ile kontrol edebilmekteyiz. Şimdi yukarıdaki Xml belgesini okuyup düğümlerle ilgili çeşitli bilgiler elde eden aşağıdaki programı yazın. Programı inceledikten sonra özellikler ve metotlar hakkında biraz daha konuşacağız.

```
using System;
using System.Xml;

class XmlBelgeleri
{
        static void Main()
        {
            XmlTextReader xtr = new XmlTextReader("yemekler.xml");

              while(xtr.Read())
              {
                    if(xtr.NodeType == XmlNodeType.Element)
                    {
                          Console.Write("{ 0,-10} ",xtr.Name);
                          if(xtr.HasAttributes)
                          {
                           Console.Write("\t{ 0} niteliği var, ",
                                      xtr.AttributeCount);

                              while(xtr.MoveToNextAttribute())
                                      Console.Write("{ 0} =
                                 { 1} ",xtr.Name,xtr.Value);
                              Console.WriteLine();
                          }
                          else
                          Console.WriteLine("\tNitelik yok...");
                    }
              }
        }
}
```


Önce programın çıktısına bakalım. (Programı derlemeden önce yemekler.txt'nin kaynak kod dosyasıyla aynı klasörde olmasına dikkat edin.)

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum14\kod>XmlTextReader
yemekler        Nitelik yok...
menu            1 niteligi var, adi = Sultan Sofrasi
corba           Nitelik yok...
yemek           Nitelik yok...
ana_yemek       Nitelik yok...
ara_yemek       Nitelik yok...
tatli           Nitelik yok...
menu            1 niteligi var, adi = Çilingir Sofrasi
corba           Nitelik yok...
yemek           Nitelik yok...
ana_yemek       Nitelik yok...
ara_yemek       Nitelik yok...
tatli           Nitelik yok...

H:\cskitap\kitap\Bolum14\kod>
```

Programın çıktısından da görüleceği üzere yemekler.xml dosyasındaki bütün elemanların adı ve sahip oldukları nitelikler yazdırılmıştır. Peki nasıl yaptık bu işlemleri? Öncelikle XmlTextReader sınıfının NodeType özelliği ile ilgili düğümün eleman (element) olup olmadığını kontrol ediyoruz. Düğümler eleman (element) olmak zorunda değildir, örneğin bir düğüm yorum satırı ya da komut bildirimi olabilir. Ya da daha spesifik olarak son eleman (end element) olabilir. XmlTextReader sınıfının NodeType özelliği System.Xml isim alanında bulunan XmlNodeType numaralandırıcısı türündendir. Bu namaralandırıcının sembollerinden biri de Element'tir. XmlNodeType numaralandırıcısının diğer sembolleri ise aşağıdaki gibidir.

Attribute	CDATA	Comment
Document	DocumentFragment	DocumentType
Element	EndElement	EndEntity
Entity	EntityReference	None
Notation	ProcessingInstruction	SignificantWhitespace
Text	Whitespace	XmlDecleration

XmlTextReader sınıfının Name özelliği, ilgili düğümün etiket ismini; Value özelliği ise ilgili düğümün değerini vermektedir. Programda bir düğümün Element olduğuna karar verildikten sonra HasAttributes özelliği ile elemanın nitelik içerip içermediği tespit edilmektedir. Eğer nitelik içeriyorsa AttributeCount özelliği ile toplam nitelik sayısı yazdırılıyor. Daha sonra XmlTextReader sınıfının MoveToNextAttribute() metodu ile bu nitelikler arasında iteratif bir şekilde ilerleniyor. Her yeni nitelikle karşılaşıldığında o niteliğin adı Name özelliği ile, değeri ise Value özelliği ile ekrana yazdırılıyor.

XmlTextReader sınıfının diğer iki önemli metodu MoveToAttribute() ve MoveToElement()'dir. Bu metotlar elemanın ya da niteliğin sıra indeksine göre XmlTextReader imlecini ilgili elemana konumlandırır. Bu özellik ve metotların yanısıra XmlTextReader sınıfının birçok özelliği ve metodu vardır. Bunların tamamına MSDN kütüphanesinden erişebilirsiniz.

Programlama yolu ile XML dosyalarını oluşturmak için XmlTextWriter sınıfını kullanabiliriz. XmlTextWriter sınıfı her bir düğüm çeşidini yazdırmak için Write ile başlayan birçok metoda sahiptir. XmlTextWriter sınıfının kullanımı oldukça basittir. Aşağıdaki programı inceleyerek bunu görebilirsiniz. Bu programda yeni.xml isimli bir belge oluşturuluyor ve bu belgeye çeşitli düğümler ekleniyor. Her bir düğümün hangi metotlarla yazıldığına dikkat edin.

```
using System;
using System.Collections;
using System.Xml;

class XmlBelgeleri
{
        static void Main()
        {

                Hashtable sozluk = new Hashtable();
                sozluk.Add("Araba","Car");
                sozluk.Add("Masa","Table");
                sozluk.Add("Okul","School");
                sozluk.Add("Bilgisayar","Computer");

                XmlTextWriter xw = new XmlTextWriter("yeni.xml",
System.Text.Encoding.GetEncoding("windows-1254"));

                xw.Formatting = Formatting.Indented;
                xw.WriteStartDocument();
                xw.WriteComment("Türkçe-İngilizce Sözlüğü");

                xw.WriteStartElement("Sozluk");

                int i=1;
                foreach(DictionaryEntry d in sozluk)
                {
                        xw.WriteStartElement("kelime");
                        xw.WriteAttributeString("No",i.ToString());
```



```
                        xw.WriteStartElement("Türkçe");
                        xw.WriteString(d.Key.ToString());
                        xw.WriteEndElement();

                        xw.WriteStartElement("İngilizce");
                        xw.WriteString(d.Value.ToString());
                        xw.WriteEndElement();

                        xw.WriteEndElement();

                        i++;
                }

                xw.WriteEndElement();
                xw.WriteEndDocument();
                xw.Close();
        }
}
```

Programı daha yakından incelemeye başlamadan önce, programın çalıştırıldıktan sonra oluşturduğu XML belgesinin içine bakalım.

yeni.xml

```
<?xml version="1.0" encoding="windows-1254"?>
<!—Türkçe İngilizce Sözlüğü-->
<Sozluk>
        <kelime No="1">
                <Türkçe>Araba</Türkçe>
                <İngilizce>Car</İngilizce>
        </kelime>
        <kelime No="2">
                <Türkçe>Okul</Türkçe>
                <İngilizce>School</İngilizce>
        </kelime>
        <kelime No="3">
                <Türkçe>Masa</Türkçe>
                <İngilizce>Table</İngilizce>
        </kelime>
        <kelime No="4">
                <Türkçe>Bilgisayar</Türkçe>
                <İngilizce>Computer</İngilizce>
        </kelime>
</Sozluk>
```

yeni.xml dosyasından da görüldüğü üzere düzenli ve okunabilir bir XML dosyası oluşturuldu. Şimdi programı ve XmlTextWriter sınıfını daha yakından inceleyelim. Öncelikle XML belgesini oluşturacak bilgiler bir Hashtable koleksiyonu şeklinde oluşturuldu. Koleksiyonlar konusunda görmüş olduğumuz yöntem ile bu Hashtable nesnesi okunup XML belgesine aktarılacak.

XmlTextWriter sınıfının 3 tane yapıcı metodu vardır. Bütün yapıcı metotlar oluşturulacak XML dosyasına ilişkin bilgileri parametre olarak alır. Bu metotlar aşağıdaki gibidir.

`XmlTextWriter(string dosya, System.Text.Encoding kodlama)`: Bu yapıcı metot ile oluşturulacak XML dosyasının yol bilgisi ve XML belgesinin karakter kodlaması System.Text.Encoding sınıfı türünden belirtilir.

`XmlTextWriter(Stream akım, System.Text.Encoding kodlama)`: Bu yapıcı metot ile oluşturulacak XML dosyasına ilişkin akım bilgisi alınmaktadır. Bu akım bir NetworkStream olabileceği gibi bir FileStream nesnesi de olabilir. İkinci parametrenin işlevi birinci metodunki ile aynıdır.

`XmlTextWriter(TextWriter tw)`: Bu yapıcı metot ile oluşturulacak XML belgesinin System.IO isim alanında bulunan TextWriter üzerinden yapılacağı bildiriliyor.

Bu programda biz birinci yapıcı metodu kullandık. Encoding parametresi olarak

```
System.Text.Encoding.GetEncoding("windows-1254");
```

yazıldığına dikkat edin. Encoding sınıfı System.Text isim alanında bulunmaktadır. Encoding nesnesi oluşturmak için statik GetEncoding() metodu bir string parametresi ile kullanılmıştır. Parametre ile verilen "windows-1254" yazısı karakter kodlamasının Windows uyumlu ve Türkçe olacağı belirtilmektedir.

XML belgesinin bizim rahat okuyabileceğimiz bir formatta oluşturulması için

```
xw.Formatting = Formatting.Indented;
```

deyimi yazıldı. Formatting özelliği bir numaralandırıcıdan ibarettir. Indented sembolü ise her elemanın bizim gözle rahat görebileceğimiz biçimde kümeleneceğini belirtmektedir. Eğer bu satırı yazmamış olsaydık XML belgemiz tek bir satır halinde oluşturulurdu, dolayısıyla XML belgesindeki elemanların ne şekilde kümelendiğini açıkça göremezdik.

WriteStartDocument() metodu ile XML belgesine versiyon bilgisi yazılıyor. Daha sonra WriteComment() metodu ile XML belgesine bir yorum satırı eklenmektedir. XML belgelerinde yorum satırları aşağıdaki gibidir.

```
<!-Türkçe İngilizce Sözlüğü-->
```


Bu kısmın başında denildiği gibi her XML belgesi mutlaka bir kök eleman içermelidir. Bu yüzden WriteStartElement() metodu ile yeni bir başlangıç elemanı yazdırılıyor. Bu metot ile XML belgesine

```
<Sozluk>
```

yazdırmış oluyoruz. Sozluk etiketinin kapatılması için XML belgesine diğer bilgileri yazdırmamız gerekir. Bu bilgileri yazdıktan sonra WriteEndElement() metodu ile

```
</Sozluk>
```

yazdırıyoruz. Bir foreach döngüsü ile Hashtable nesnesindeki bütün anahtar-değer ikilileri bir kelime etiketi oluşturacak şekilde XML belgesine yazdırılıyor. Her bir elemana ilişkin yazdırılma işlemi bittikten sonra WriteEndElement() metodunun çağrıldığını unutmayın. WriteAttributeString() metodu ile bir elemana aşağıdaki gibi nitelik veriliyor.

```
xw.WriteAttributeString("No",i.ToString());
```

"No" niteliğin ismidir, ikinci parametre ise niteliğin alacağı değerdir. Niteliğin alacağı bu değer döngünün her iterasyonunda artan bir tamsayıdır. Bu parametre string türünden olduğu için ToString() metodu kullanılmıştır.

Diğer bir metot olan WriteString() ise bir eleman düğümü için değer yazmaktadır. Hashtable nesnesindeki Türkçe ve İngilizce kelimeler bu metot ile yazdırılmaktadır.

XmlTextWriter sınıfının birçok metodu daha vardır. Bu metotlarda XML belgelerine diğer düğüm elemanlarını yazdırmak için kullanılır. Bu metotların kullanım şekli yukarıda bahsedilen metotlar ile aynıdır. Bu yüzden eğer XML bilginiz varsa burada anlatılmayan metotları ve özellikleri kolaylıkla öğrenip programlarınızda kullanabilirsiniz. Bu bilgileri elbette ki MSDN kütüphanesinden öğrenebilirsiniz. Çünkü MSDN kütüphanesi her bir metot ya da özellikle ilgili ayrıntılı bilgi sunmaktadır. Bu bölümün amacı .NET kütüphanesini bütün ayrıntılarıyla anlatmak değil sadece size bu kütüphaneyi kullanmaya nereden başlayacağınıza dair bir fikir vermektir.

DOM (Document Object Model)

XML belgeleri ile çalışmanın tek yolu XmlTextReader ve XmlTextWriter sınıfları değildir. W3C'nin DOM denilen modelini destekleyen ve System.Xml isim alanında bulunan XmlDocument sınıfı ile de çalışmamız mümkündür. XmlDocument ile bir XML belgesi tamamen belleğe yüklenir ve bir DataSet gibi üzerinde işlemler yapılıp tekrar XML belgesine güncelleme yapılır. XmlDocument ile XML belgesi belleğe bir ağaç yapısı şeklinde yüklendiği için çeşitli sınıflar yardımıyla her bir düğüm elde edilip üzerinde çeşitli işlemler yapılabilir.

XmlDocument ile çalışırken sıkça kullanacağımız iki sınıf daha vardır. Bunlar XmlElement ve XmlAttribute sınıflarıdır. Bu sınıflar sırasıyla bir eleman ve nitelik temsil etmek için kullanılır. Bir XmlDocument nesnesi oluşturmak için yol bilgisi ya da akım bilgisi gerekmektedir. XmlDocument nesnesi bellekteki verileri temsil etmektedir. Bu yüzden XmlDocument nesnesine XML belgesini yüklememiz için XmlTextReader sınıfına ihtiyacımız var. XmlTextReader sınıfı ile okunan XML belgesini, XmlDocument nesnesine aktarmak için XmlDocument sınıfının Load() metodu kullanılır. XmlDocument nesnesinde yapılan değişiklikleri tekrar XML belgesine kaydetmek için ise Save() metodu kullanılır. Şimdi bir önceki başlık altında oluşturduğumuz yeni.xml belgesini XmlDocument nesnesine ne şekilde aktarabileceğimize bakalım.

```
XmlTextReader xr = new XmlTextReader("yeni.xml");

XmlDocument xd = new XmlDocument();

xd.Load(xr);
```

XmlDocument sınıfının burada tamamını anlatamayacağımız birçok metodu, özelliği hatta olayı vardır. Aşağıda sık kullanabileceğimiz bazı özellikler verilmiştir.

`Attributes`: İlgili düğüme ilişkin bir nitelik koleksiyonu döndürür. Bu koleksiyonun türü XmlAttributeCollection'dır.

`ParentNode`: Bir düğüme ilişkin XmlNode sınıfı ile temsil edilmiş ana düğümü verir.

`HasChildNodes`: Eğer ilgili düğüm başka bir yavru düğüm (**child node**) içeriyorsa true değeri döndürür.

`FirstChild`: İlgili düğüme ilişkin ilk yavru düğümü XmlNode sınıfı türünden verir.

`LastChild`: İlgili düğüme ilişkin son yavru düğümü XmlNode sınıfı türünden verir.

`ChildNodes`: İlgili düğüme ilişkin yavru düğümleri bir XmlNodeList koleksiyonu şeklinde verir.

`NextSibling`: Üzerinde çalışılan düğümden hemen sonra gelen düğümü verir.

`PreviousSibling`: Üzerinde çalışılan düğümden bir önce gelen düğümü verir.

`InnerText`: Bir düğümün içinde yavru düğümleri ve varsa düğümlerin değerlerini verir.

`InnerXml`: Bir düğüme ilişkin bilgileri ve yavru düğümleri XML etiketleri ile beraber string türünden verir.

`OuterXml`: Bir düğüme ilişkin bilgileri XML etiketleri ile beraber string türünden verir. Bu özellikte yavru düğümler alınmaz.

`Name`: Bir düğümün ismini verir.

`Value`: Bir düğüme ilişkin değeri verir.

`NodeType`: Düğümün hangi tipte olduğunu XmlNodeType numaralandırması ile verir.

XmlDocument sınıfının birçok metodu daha bulunmaktadır. Bu metotlar içinde yine MSDN kütüphanesine başvurmanız gerekir.

Aşağıdaki örnekte XmlDocument sınıfının kullanımına bir örnek verilmiştir.

```
using System;
using System.IO;
using System.Xml;
class XmlBelgeleri
{
        static void Main()
        {
                XmlTextReader xr = new XmlTextReader("yeni.xml");

                XmlDocument doc = new XmlDocument();
                doc.Load(xr);

                XmlNodeList xnl = xnl = doc.ChildNodes;

            Console.WriteLine("Toplam yavru düğüm: " + xnl.Count);
                foreach(XmlNode xn in xnl)
                        Console.WriteLine("Yavru düğüm adı:{0},
                                        türü:{1}",xn.Name,xn.NodeType);

                Console.WriteLine("\n");

                XmlElement xe = doc.DocumentElement;

                if(xe.HasChildNodes)
                {
```

```
                Console.WriteLine("İlk yavru
                            düğüm:{ 0} ",xe.FirstChild.Name);
            }

            XmlNodeList xnl2 = xe.FirstChild.ChildNodes;

            foreach(XmlNode xn in xnl2)
                Console.WriteLine("Yavru düğüm adı:{ 0} ,
                            türü:{ 1} ",xn.Name,xn.NodeType);

        }
}
```

Programın ekran çıktısı aşağıdaki gibidir.

```
D:\WINNT\System32\cmd.exe

H:\cskitap\kitap\Bolum14\kod>XmlDocument
Toplam yavru düğüm: 3
Yavru düğüm adi:xml, türü:XmlDeclaration
Yavru düğüm adi:#comment, türü:Comment
Yavru düğüm adi:Sozluk, türü:Element

Ilk yavru düğüm:kelime
Yavru düğüm adi:Türkçe, türü:Element
Yavru düğüm adi:Ingilizce, türü:Element

H:\cskitap\kitap\Bolum14\kod>
```

Bu örnekle birlikte XmlDocument sınıfını giriş niteliğinde de olsa incelemiş olduk. Bundan sonra yazacağınız programlar sizin XML belgelerine ne şekilde ihtiyaç duyduğunuzla alakalıdır. Burada vurgulamamız gereken önemli bir nokta var: Sınıf kütüphanesinin çok geniş olması gözünüzü korkutmasın. Çünkü kütüphaneler sadece lazım oldukça kullanılır. Hiçbir zaman .NET kütüphanesindeki bütün yapıları öğrenmeye kalkmayın. Çünkü bu hem zor hem de gereksizdir. Geliştireceğiniz programın ihtiyaçlarını tespit ettikten sonra gerekli sınıfları öğrenmek sizin için çok kolay olacaktır. Yeter ki burada da değindiğimiz temel kavramları iyice öğrenmiş olun.

.NET kütüphanesinde XML belgeleri ile ilgili işlem yapan sınıflar bu kadarla sınırlı değildir. System.Xml.Xsl isim alanında XSL dönüşümleri yapan sınıflar vardır. Aynı zamanda System.Xml.Xpath isim alanında XML ağacında çeşitli sorgulamalar yapabimemize imkan sağlayan çeşitli sınıflar mevcuttur. XSL ve XPATH kavramları XML standartları içerisindedir. Bu standartlarla ilgili ayrıntılı ve detaylı bilgiyi XML üzerine yazılmış çeşitli kitaplardan edinebilirsiniz.

16

Visual Studio.NET ve Form Tabanlı Uygulamalar

- **Visual Studio.NET**
- **Visual Studio.NET Projeleri**
  - **Yeni Proje Eklemek**
- **Oluşturma (Build) ve Hata Ayıklama (Debug)**
- **Windows Uygulamaları Geliştirme**
  - **Form ve Kontrollerin Kullanımı**
- **Visual Studio.NET ile Form Tabanlı Çalışmak**
  - **Birden Fazla Form ile Çalışmak**
  - **Diğer Kontroller**

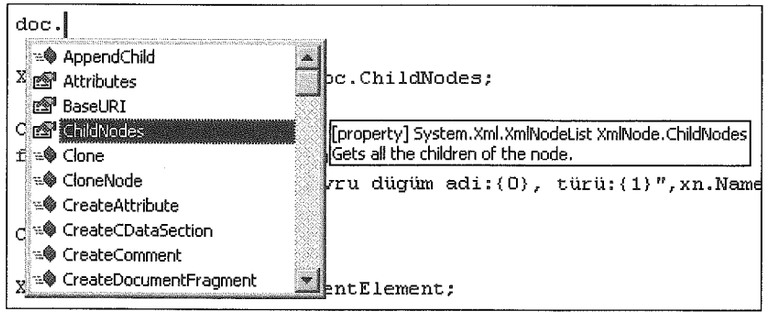
Bu kitabın yazılmasındaki amaç C# dilini anlatmaktır. Bu yüzden bütün bölümlerde programlarımızı basit bir metin editöründe yazdık ve komut satırı derleyicisini kullandık. Ancak gerçek hayatta uygulamaların bu kadar basit olmadığını ve kodların binlerce satıra eriştiğini düşünürsek kaynak kodumuzu yönetebileceğimiz ve program yazmak için optimize edilmiş çeşitli modern editörlere ihtiyacımız vardır. C# ile .NET ortamında programlama yaparken belki de en sık kullanacağınız araç Visual Studio.NET olacaktır. Bu bölümde Visual Studio.NET aracının kullanımına değinilecektir. Ayrıca Windows pencerelerine yönelik uygulama geliştirmenin nasıl olduğunu temel hatlarıyla inceleyeceğiz.

Visual Studio.NET

Visual Studio.NET, gelişmiş bir yazılım geliştirme ortamıdır. Öyle ki Visual Studio.NET ortamına alıştığınızda artık bir metin editöründe program yazamayacak duruma bile gelebilirsiniz. Bunun sebebi Visual Studio.NET ortamının programcıya büyük imkanlar sağlamasıdır. Peki nedir bu imkanlar?

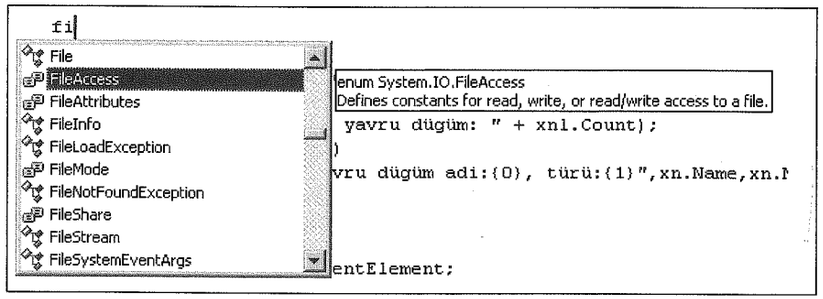
Her şeyden önce Visual Studio.NET çok gelişmiş bir metin editörüne sahiptir. Bu metin editörünün birçok modülle birlikte çalışma yeteneği vardır. Örneğin Windows programları geliştirirken kontrollerin özellikleri değiştirildiği zaman editördeki kaynak kod otomatik olarak değiştirilir. Editör, C# diline çok hakimdir, zira kod yazarken parantezlerin ve blokların birbirine karışmaması için çeşitli uyarılar vermektedir.

Editörün en önemli ve en sevilen özelliği ise akıllı bir editör olmasıdır. Bu akıllılığı, bir sınıfın elemanlarını size gösterebilmesinden kaynaklanır. Bir çoğunuz bunu zaten biliyorsunuzdur. Ama Visual Studio .NET gibi bir ortamda çalışmamış birisi için aşağıdaki ekran görüntüsü yardımcı olacaktır.

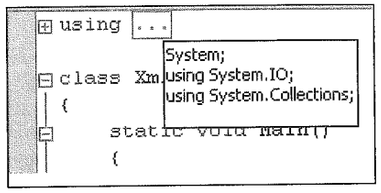


Yukarıdaki ekran görüntüsünde XmlDocument türünden bir nesneye ait metotlar ve özelliklerin '.' işareti girildikten sonra ne şekilde göründüğü gösterilmektedir. Ayrıca her bir metot ve özellik seçildiğinde eleman hakkında kısa bilgi verilmektedir.

Visual Studio.NET'in bu özelliği .NET kütüphanesini kullanmayı kolaylaştırmaktadır. Editörün diğer bir önemli özelliği ise **Ctrl+Space** tuş kombinasyonu ile bir faaliyet alanındaki nesnelerin isimlerini göstermesidir. Editörün herhangi bir boş alanında Ctrl+Space tuşuna bastığınızda ilgili yerde kullanımı geçerli olan bütün değişkenler ve türler bir liste halinde aşağıdaki gibi gösterilir. Kullanmak istediğiniz nesneyi ya da türü seçtikten sonra Enter ya da Space tuşuna basarsanız seçtiğiniz değişken ya da türün ismi otomatik olarak kaynak koda dahil edilir.



Editöre "fi" yazısı yazıldığı anda Ctrl+Space tuşuna basıldığı için "fi" yazısı ile başlayan elemanlar listelendi. (Her bir veri türü yukarıdaki listeden de görüldüğü üzere farklı sembollerle gösterilmektedir.)



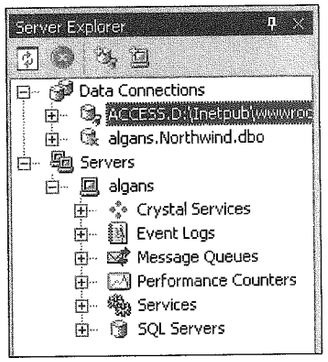
Editörle ilgili diğer bir güzel özellik ise kod bloklamaya imkan vermesidir. Önişlemci komutlarında görmüş olduğumuz #region ve #endregion komutları kodların bloklanmasına yarar. Bu komutları kullanarak istediğimiz kod bloğunu kümelenebilecek şekilde işaretleyebiliriz. Fakat varsayılan olarak metot bildirimleri, sınıf bildirimleri kümelenebilir. Kümelenmiş bir kod bloğunu aşağıdaki örnekte görmektesiniz.

Farenin imleci kümelenmiş blok üzerine geldiğinde kümelenmiş kod küçük puntolarla gösterilmektedir.

Visual Studio.NET geliştirme ortamı ile bütünleşik çalışan bir hata ayıklayıcı program bulunmaktadır. Debugger denilen bu hata ayıklayıcı ile çalışma zamanında bir değişkenin değerini adım adım izleyip çeşitli saptamalarda bulunabiliriz. Hata ayıklayıcının nasıl kullanıldığı ilerleyen kısımlarda anlatılacaktır.

Visual Studio.NET ortamında derleyiciyi komut satırından çalıştırmamıza gerek yoktur. Daha sonra açıklayacağımız projeyi oluşturma sırasında derlemenin ne şekilde olduğu anlatılacaktır. Ayrıca Visual Studio.NET, Debug ve Release gibi çeşitli derleme imkanları sunmaktadır. Build ve Release şeklinde derlemenin ne demek olduğu da ileride anlatılacaktır.

Visual Studio.NET geliştirme ortamındaki belki de en ilginç özellik, Visual Studio.NET ile bütünleşik olan MSDN yardım dokümanlarının indeksinin yazdığımız koda göre otomatik bir şekilde değişmesidir. Ayrıca F1 tuşuna basarak da Visual Studio.NET içindeki Explorer ile yardım dosyalarını görüntüleyebilirsiniz.



Visual Studio.NET ortamından çıkmadan sisteminizde bulunan çeşitli sunuculara ve servislere erişmeniz de mümkündür. Örneğin bir SQL Server'daki veritabanında bulunan kayıtları incelemek için Visual Studio.NET ortamının dışına çıkmanıza gerek yoktur. Server Explorer pencerisinden sisteminizde bulunan sunuculara ve diğer Windows servislerine kolaylıkla ulaşabilirsiniz. Aşağıda benim bilgisayarımda Server Explorer penceresinin görüntüsü bulunmaktadır.

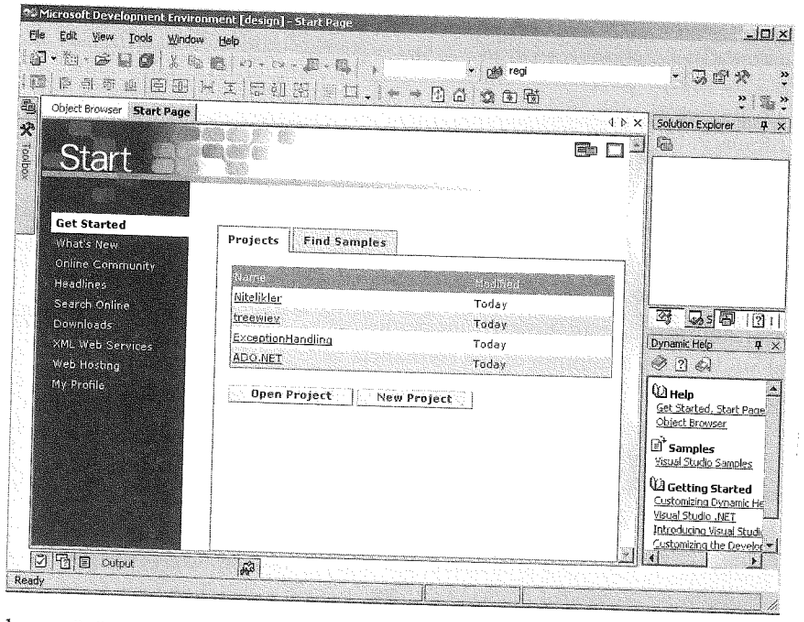
Yukarıdaki pencerenin köşesinde bulunan işaretini tıkladığınızda pencere animatik bir şekilde kaybolacaktır. Tekrar ilgili pencereye tıklarsanız pencere yeniden gelecektir. Bu işlem daha çok kod yazma editörüne daha çok alan ayırmak için yapılır.

Visual Studio.NET IDE (Integrated Development Environment)'sinin burada anlatamayacağımız fakat programcının işini kolaylaştıran birçok özelliği daha vardır. Bunları Visual Studio.NET ortamını kullandıkça öğreneceksiniz.

Visual Studio.NET Projeleri

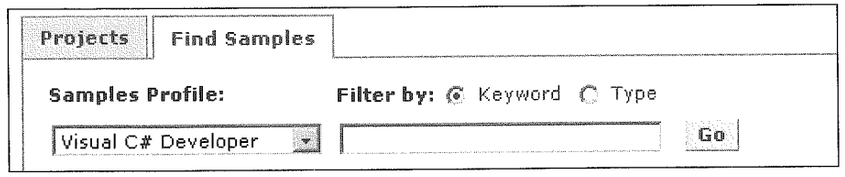
Visual Studio.NET programcılara birtakım proje şablonları sunmaktadır. Bu şablonlar özellikle grafik arayüzlü projelerde büyük fayda sağlamaktadır. Örneğin grafik arayüzlü bir Windows uygulaması geliştiriyorsanız çeşitli Windows kontrollerini görsel olarak yerleştirmek için yapacağımız işlem sürekle-bırak (drag and drop)'tan başka bir şey değildir. Proje şablonları her bir projeye başlamadan önce çok sık yazılan kodları otomatik olarak yazdırır. Örneğin şu ana kadar yaptığımız konsol uygulamalarını düşünün. Her programımızda mutlaka bir sınıf ve Main() metodu vardı. Ayrıca kaynak kodun en başında da using bildirimi vardı. Eğer Visual Studio.NET ile bir konsol projesi açarsak bu başlangıç kodları otomatik olarak bizim için oluşturulacaktır. Her bir proje şablonu için değişik başlangıç kodları mevcuttur. Şimdi Visual Studio.NET ile oluşturabileceğimiz proje şablonlarını inceleyelim.

Start menüsünden Programs -> Microsoft Visual Studio.NET -> Visual Studio.NET yolunu izleyerek geliştirme ortamımızı başlatalım. Visual Studio.NET ilk açıldığında aşağıdaki ekran görüntüsü ile karşılaşırız.



Bu ekran görüntüsündeki orta sayfa bir html sayfası yani bir web sayfasıdır. Bu sayfada en son açmış olduğunuz projeler bulunmaktadır. Ayrıca yandaki menüde .NET

ile ilgili çeşitli online kaynakların bulunduğu bağlantılar mevcuttur. Web sayfasındaki "Find Samples" sekmesini tıkladığınızda aşağıdaki ekran görüntüsünü elde ederiz.

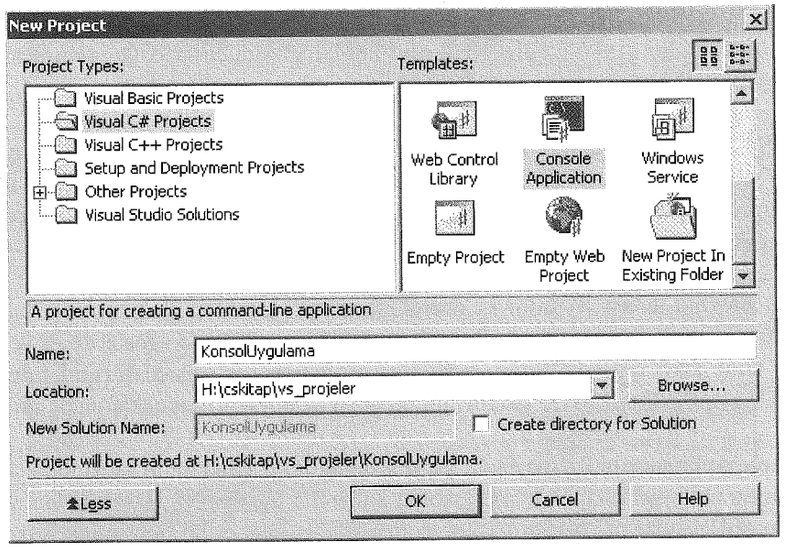


Bu ekrandan bir anahtar kelime girerek dünyadaki diğer programcıların yapmış olduğu örnek projelere ya da MSDN içerisindeki örnek projelere erişebilirsiniz. Tabi internetteki kaynaklara erişebilmeniz için internet bağlantınızın olması gerekmektedir.

Visual Studio.NET'i açtığınızda gelen ilk sayfadaki "New Project" düğmesine basarak ya da File-> New -> Project yolunu izleyerek yeni bir proje oluşturalım. Karşımıza gelecek ekranda çeşitli dillere ilişkin proje tiplerini seçme imkanımız olacaktır. Biz C# ile ilgilendiğimize göre aşağıdaki ekran görüntüsünden Project Type (Proje Tipi) olarak "Visual C# Projects" Templates (Şablonlar) ekranından da dilediğimiz proje şablonunu açacağız.

Yeni açılan ekrandaki proje şablonlarını tek tek ne olduğunu açıklayacağız. Ancak şunu bilmenizde fayda var ki bazı proje tipleri bu kitabın kapsamı dışındadır. "ASP.NET Web Service" proje şablonu, "Windows Service" proje şablonu bunlara birer örnektir.

Yeni bir proje açtığımızda karşımıza gelecek ekran aşağıdaki gibidir.



Yukarıdaki görüntüde bulunan "Name" alanı projenin ismini belirtmektedir. "Location" alanı ise projenin hangi klasörde oluşturulacağını belirtmektedir. "OK" düğmesine tıkladığınızda "Location" ile belirtilen yolda "Name" ile belirtilen yeni bir klasör oluşturulur ve proje ile ilgili dosyalar bu klasörde toplanır. "OK" tuşuna basmadan önce "Templates" kısmında gördüğünüz proje tipleri hakkında biraz bilgi verelim.

Bir önceki sayfada bulunan ekran görüntüsünde bütün proje şablonları görünmemektedir.

Windows Application: Klasik windows uygulamaları için özelleştirilmiş bir şablondur. Bu proje tipini seçtiğinizde Windows pencereleri ile ilgili isim alanlarının otomatik olarak kaynak kodunuza eklendiğini görürsünüz. Bu bölümün ikinci kısmında bu proje şablonu ile ilgili genel bilgiler verilecektir.

Class Library: Daha önce yapmış olduğumuz bağımsız sınıf tasarımı ile ilgili bir proje tipidir. Bu proje tipi ile başka programlarda kullanılmak üzere sınıf kütüphanesi geliştirilir. Bu proje şablonunda program derlendiği zaman oluşacak assembly dosyası dll uzantılı olacaktır.

Windows Control Library: Bu proje şablonunda grafik arayüzü bulunan sınıf tasarımları yapılır. Genellikle varolan Windows kontrollerinden türetilerek yapılırlar. Bu proje şablonunu bir önceki versiyonlarda bulunan ActiveX kontrollerine benzetebiliriz. Bu proje şablonunda geliştirilen kontroller diğer .NET uyumlu dillerde kullanılabilmektedir.

ASP.NET Web Application: Klasik web sayfalarını oluşturmak için özelleştirilmiş bir proje şablonudur. ASP.NET ile .NET sınıf kütüphanesinden faydalanma imkanımız olduğu için web programlama .NET ile oldukça gelişmiştir diyebiliriz.

ASP.NET Web Service: C# ile XML Web Servisleri oluşturmak için özelleştirilmiş bir proje şablonudur. XML Web Servsileri ile uzak ya da yerel bir sistemdeki bir metot http üzerinden XML formatında çağrılabildiği için yeni iş uygulamalarında sıkça kullanılacak bir proje şablonudur. Bu kitapta Web Servislerine değinilmeyecektir.

Web Control Library: Windows Control Library proje şablonuna benzer bir şablondur. Bu sefer oluşturulan kontroller internet tarayıcısı üzerinde gösterilecek şekilde .NET tarafından ayarlanır. Programcının yapacağı işler çok istisnai durumlar dışında genelde aynıdır. Zaten .NET platformunun gücü de buradan kaynaklanmaktadır.

Console Application: Şu ana kadar yapmış olduğumuz gibi komut satırından çalışacak programlar geliştirmek için bu şablon kullanılır. csc derleyicisi varsayılan olarak bir konsol uygulaması olacak şekilde derlediği için kitabın bu bölümüne kadar csc

derleyicisini kullandık. Gerçek uygulamalarda bu proje şablonu çok nadir kullanılır. En çok kullanıldığı yerlerden biri sunucu-istemci mimarisine dayanan küçük çaplı projelerde sunucu bir konsol uygulaması olabilir. Bu proje tipinin diğer bir kullanımı da C# ya da diğer .NET dillerinin temellerini iyi bir şekilde öğrenimi amaçlandığında söz konusudur. Bu kitabın birinci amacı size C# dilini öğretmek olduğu için şimdiye kadar bu uygulama şablonu kullanıldı.

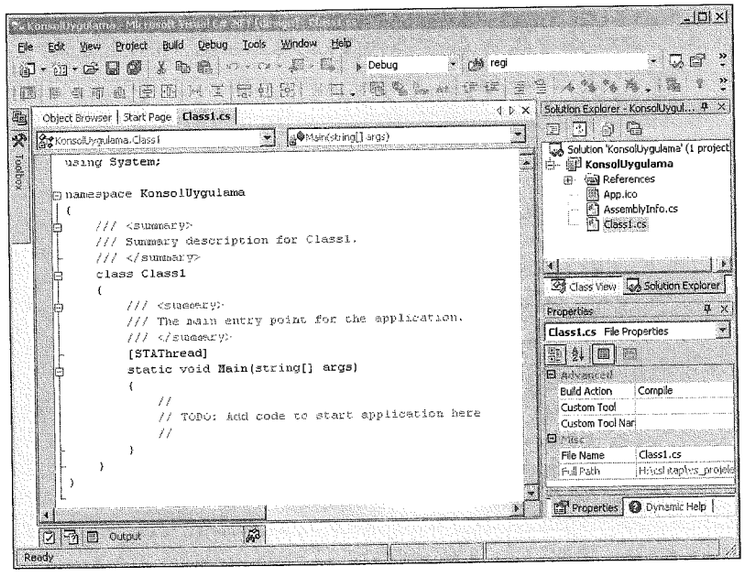
Windows Service: Windows 2000 ve NT gibi sistemlerde arka planda çalışacak servis dediğimiz modüllerin geliştirilmesi için özelleştirilmiş bir proje şablonudur.

Empty Project: Boş bir proje oluşturmak için bu şablon kullanılır. Boş bir proje oluşturduktan sonra gerekli bütün kaynak kodları içeren dosyaları programcının kendisi eklemelidir.

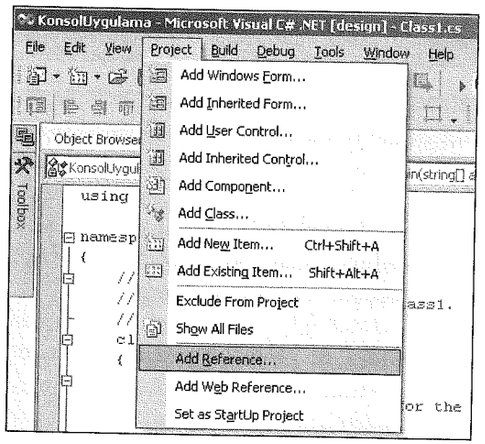
Empty Web Project: Yine boş bir proje oluşturulur ancak bu şekilde oluşturulan proje web sunucusuna göre kendini ayarlar. Dolayısıyla bu proje tipine eklenecek kodlar bir web sayfası şeklinde derlenebilecek şekilde ayarlanır.

New Project In Existing Folder: Bu proje tipi genellikle başka ortamlarda (editörlerde) yazılmış kaynak kodları Visual Studio.NET ortamına entegre etmek için kullanılır. Adından da anlaşılacağı üzere varolan bir dizinde yeni bir proje oluşturulur.

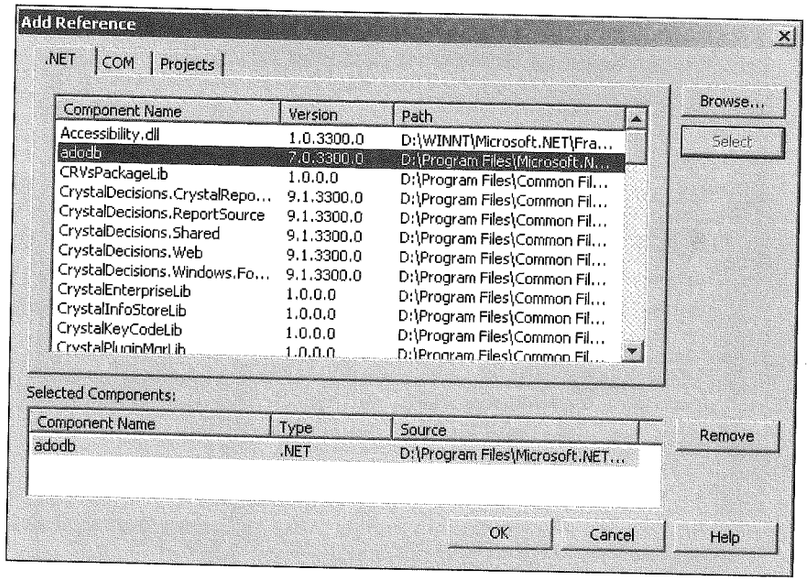
Şablon olarak "Console Application"ı seçip uygun isimleri verdikten sonra "OK" düğmesine tıklayın; aşağıdaki ekran görüntüsüyle karşılaşacaksınız.



Ekran görüntüsünden de görüldüğü üzere proje oluşturulup yeni bir Class1.cs isimli bir kaynak dosya oluşturulmuştur. "Solution Explorer" penceresinde projemize ilişkin kaynak kodlar ve çeşitli bilgiler bulunmaktadır. Bu bilgiler arasında **Reference** başlığı altında harici assembly dosyaları bulunmaktadır. Bu referanslar projeye eklediğimiz assembly dosyalarına göre değişmektedir. Daha önceden oluşturmuş olduğumuz bir sınıf kütüphanesini kullanmak için gibi **Project** menüsünden **Add Reference** menüsünü tıklayıp istediğimiz sınıf kütüphanesine referans verebiliriz.

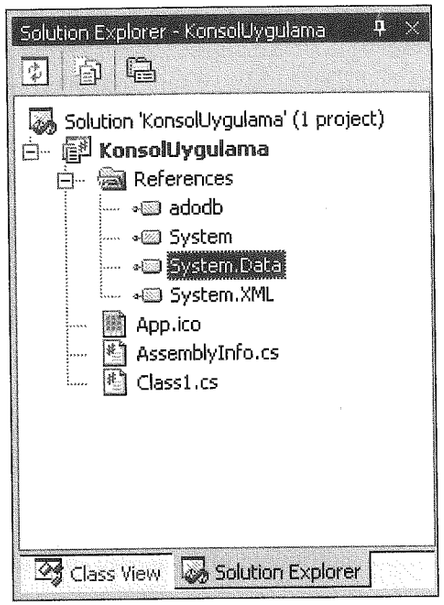


Add Reference menüsüne tıkladığımızda aşağıdaki ekran görüntüsü gelecektir.



Bu pencereden .NET sekmesini seçtikten sonra sistemimize daha önceden yüklenmiş olan ve Visual Studio.NET'in varlığından haberdar olan .NET uyumlu sınıflar görülmektedir. İster bu sınıflardan birini seçip "Select" düğmesine tıklayıp "Selected Components" kısmına ekleriz, istersek de "Browse" düğmesini kullanıp kendi oluşturduğumuz sınıf kütüphanesini "Selected Components" bölümüne ekleriz. Eklemek istediğimiz Dll kütüphanesini seçtikten sonra "OK" düğmesine tıklayıp geliştirme ortamına tekrar dönelim. "Solution Explorer" penceresi eklenen bu yeni referansa göre aşağıdaki gibi yeniden oluşturulacaktır.

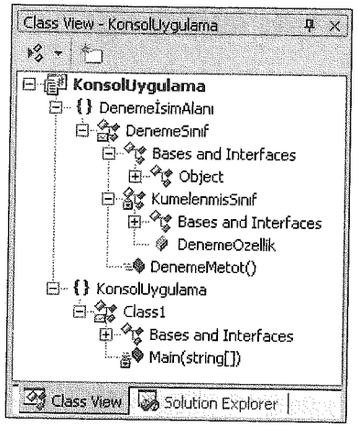
"Solution Explorer" penceresinden de görüleceği üzere programa eklenen referanslar eklenme sırasına göre listelenmiştir. Derleme işlemi sırasında bu referanslara ilişkin assembly dosyaları kullanılır. Diğer bir önemli pencere de "Solution Explorer" sekmesinin yanında bulunan "Class View" sekmesini tıklayarak açacağımız penceredir. Bu pencerede kaynak kod içinde bildirilmiş olan sınıflar hiyerarşik bir yapıda gösterilir. Kaynak kodumuzu aşağıdaki şekilde değiştirip "Class View" penceresinin ne şekilde değiştiğini inceleyelim.



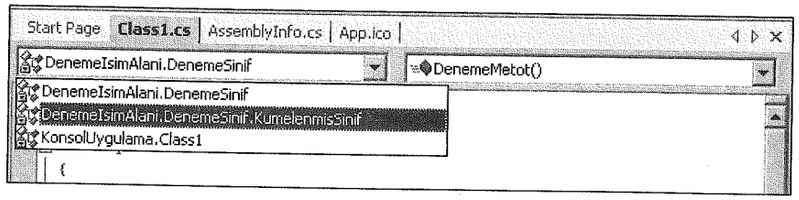

```
using System;

namespace DenemeİsimAlanı
{
        class DenemeSınıf
        {
                class KumelenmisSınıf
                {
                        public int DenemeOzellik;
                }
                public int DenemeMetot()
                {
                }
        }
}

namespace KonsolUygulama
{
```

```
        class Class1
        {

                static void Main(string[] args)
                {

                }
        }
}
```

"Class View" pencresinde kaynak kodda bildirimi yapılan bütün isim alanları, isim alanlarında bildirilen sınıflar ve sınıflarda bildirilen metot ve özellikler ya da diğer elemanlar hiyerarşik bir biçimde gösterilir. "Class View" penceresinden herhangi bir elemana tıkladığımızda kaynak kodda ilgili elemanın bildirimin yapıldığı yere farenin imleci konumlandırılır. Binlerce satırlık kaynak kodu olan projelerde bu pencere inanılmaz derecede faydalıdır.



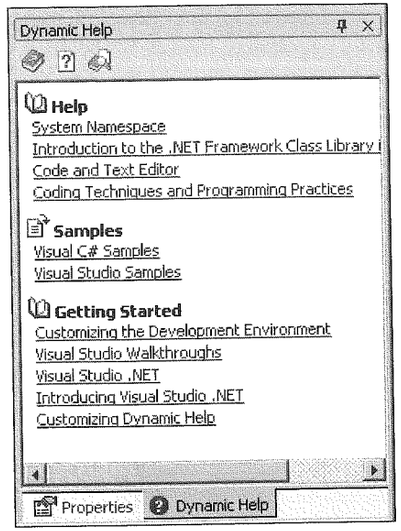
Sınıflar ve metotlar arasında dolaşabilmemizi sağlayan diğer bir görsel yapı ise bir sonraki sayfadaki görüntüdür. İlgili sınıf ya da diğer yapılar seçildikten sonra sınıf listesinin hemen yanındaki listeden sınıfın elemanı seçilirse farenin imlece ilgili elemanın bildiriminin yapıldığı yere konumlandırılacaktır.



Proje ortamındaki diğer önemli bir pencere ise "Dynamic Help" penceresidir. Bu pencerede kaynak kod üzerinde herhangi bir anahtar sözcüğün ya da sınıfın üzerine gelindiğinde ilgili birim ile ilgili yardım dosyalarına gitmek için bağlantılar gelir. Örneğin farenin imlecini

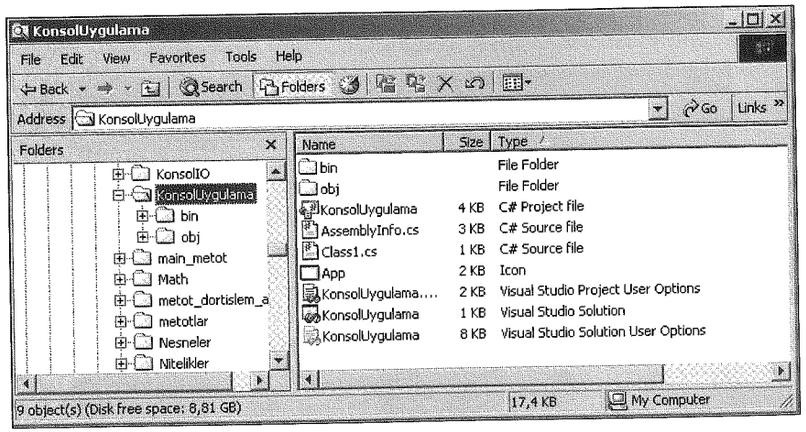
```
using System;
```

deyimi üzerine getirdiğinizde "Dynamic Help" penceresi aşağıdaki gibi değişmektedir.



Yardım penceresinde yardım almak istediğiniz anahtar sözcük ile beraber standart olan birtakım başlıklar da vardır. Bu başlıklardan birine tıklayarak yardım dokümanının açılmasını sağlayabilirsiniz. Yardım dokümanını ister "Dynamic Help" penceresinin içinde, ister Visual Studio.NET ortamındaki tarayıcı sayfada, isterseniz standart yardım dosyası okuyucusu ile de görebilirsiniz. Bu seçenekleri görmek için yardım almak istediğiniz başlığın üzerine farenin sağ tuşu ile tıklamınız yeterlidir.

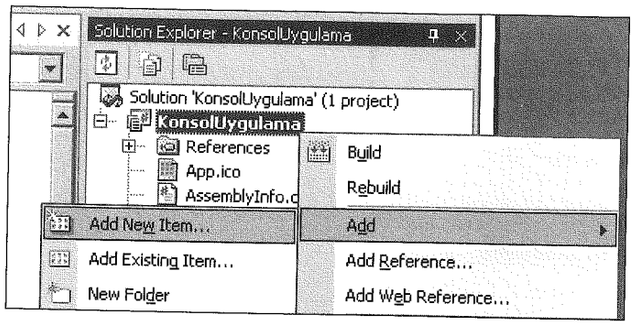
Şimdi "Console Application" projesinin seçmiş olduğumuz klasörde oluşturduğu dosyalara bakalım. Basit bir konsol uygulaması oluşturmamıza rağmen aşağıdaki ekrandan da gördüğünüz üzere tek bir kaynak kod dosyası yerine birçok dosya oluşturulmuştur.



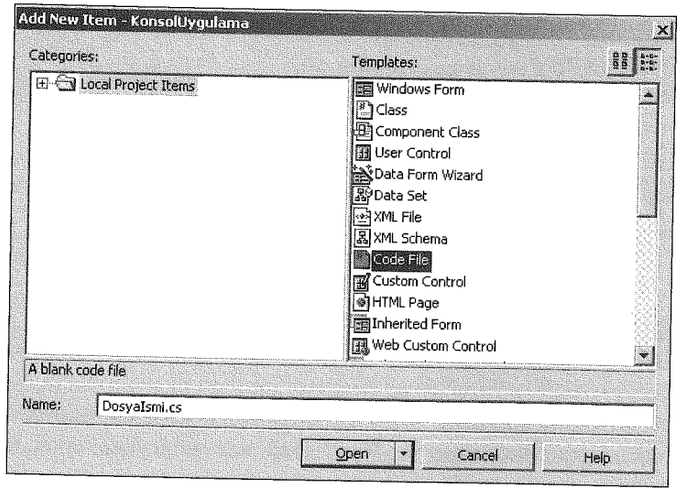
Projemizin dizininde yukarıdaki ekran görüntüsünden de görüleceği üzere iki ayrı dizin oluşturulmuştur. Bu dizinlerden "bin" isimli olanı projeyi derlerken oluşturulan assembly dosyalarını tutmak için oluşturulmuştur. "obj" isimli dosya ise ara kod denilen object kodları ve çeşitli geçici dosyaları saklamak için kullanılır. "Class1.cs" isimli dosya Main() metodunun bulunduğu kaynak dosyamızı belirtmektedir. Diğer dosyalar ise proje ile ilgili çeşitli bilgileri XML formatında saklayan birimlerdir. Yeni bir proje açıldığında Visual Studio.NET varsayılan olarak bir "Solution" (Çözüm) dosyası oluş-

turmaktadır. "Solution" birden fazla projeyi tek çatı altında toplamak için kullanılan bir yapıdır. Örneğin bir "Solution" içinde hem konsol uygulaması hem de bir Windows uygulaması oluşturabiliriz. Birden fazla proje ile çalışmanın nasıl yapıldığına birazdan değineceğiz. Ama öncelikle tek bir projeye yeni bir dosyanın nasıl eklendiğine bakalım.

Oluşturduğumuz konsol uygulamasına yeni bir kaynak kod dosyası eklemek için "Solution Explorer" penceresini kullanacağız. "Solution Explorer" pencresinden proje ismine sağ tıklayıp "Add New Item" menüsünü aşağıdaki gibi seçin.



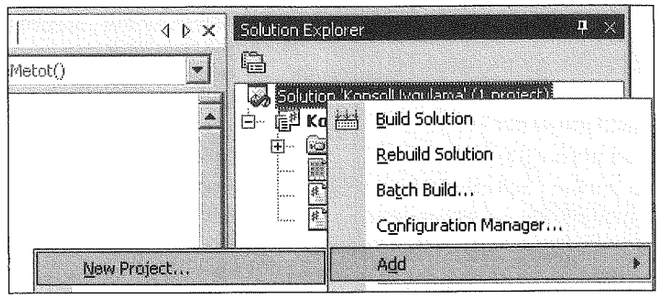
"Add New Item" seçildikten sonra aşağıdaki ekran görüntüsü gelecektir. Bu ekran görüntüsünden projeye çeşitli yeni birimler eklemeniz mümkündür. Örneğin bir "Class" eklediğinizde yeni bir kaynak kod projeye eklenip sınıfla ilgili ilk bildirimler yapılacaktır. "Cod File" eklediğinizde ise, cs uzantılı boş bir dosya oluşturulacaktır.



Proje ismine sağ tıkladığınızda çıkan menüden "Solution Explorer" penceresindeki herhangi bir dosya silenebileceği gibi yeni bir klasör de oluşturulabilir. Ayrıca "Add Existing Item" seçeneği ile de varolan bir dosya projeye eklenebilir.

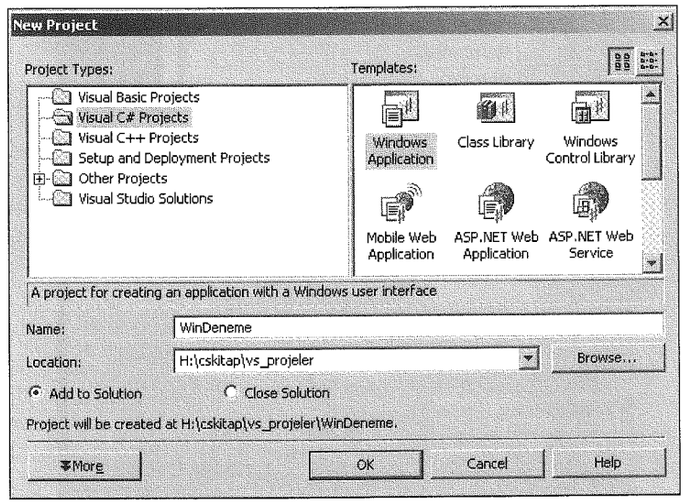
Yeni Proje Eklemek

Daha önce denildiği gibi Visual Studio.NET içinde birden fazla proje ile çalışmamız mümkündür. Birden fazla proje ile çalışmak genellikle çok katmanlı iş uygulamalarında görülmektedir. İşin ilginç tarafı projelerin dilinin önemli olmamasıdır. Bir "Solution" altında C# ile geliştirilen bir Windows uygulaması ile C++ ile geliştirilen bir proje aynı anda bulunabilir. .NET "Solution"ın tamamı derlenmeye çalışıldığında her bir proje için ilgili derleyiciyi çalıştırılır. Bir projeye yeni bir proje eklemenin iki yolu vardır. Birincisi "Solution Explorer" penceresinden aşağıdaki gibi "Solutin 'Konsol Uygulaması'" satırına sağ tıklayıp gelen menüden "Add" sekmesini oradan da "New Project" sekmesini seçmektir.



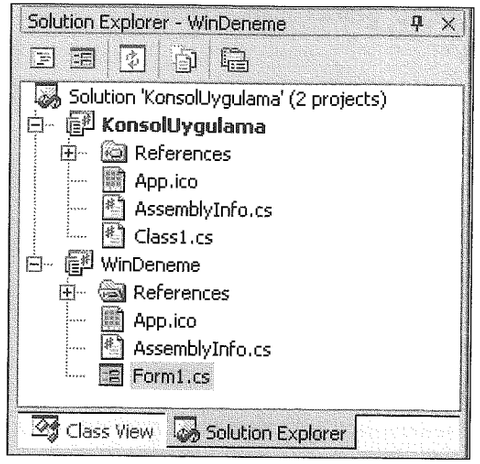
"New Project" sekmesine tıklandığında daha önce de gördüğümüz yeni proje tipi ve proje şablonu ekleme ekranı gelecektir. İlgili seçenekleri işaretledikten sonra "OK" düğmesine tıkladığınızda seçtiğiniz proje direkt "Solution"a eklenmiş olacaktır.

Yeni bir proje eklemek için kullanabileceğiniz diğer bir yöntem ise "File" menüsünden "New" oradan da "Project" sekmesini seçip aşağıdaki ekran görüntüsünü getirmektir.

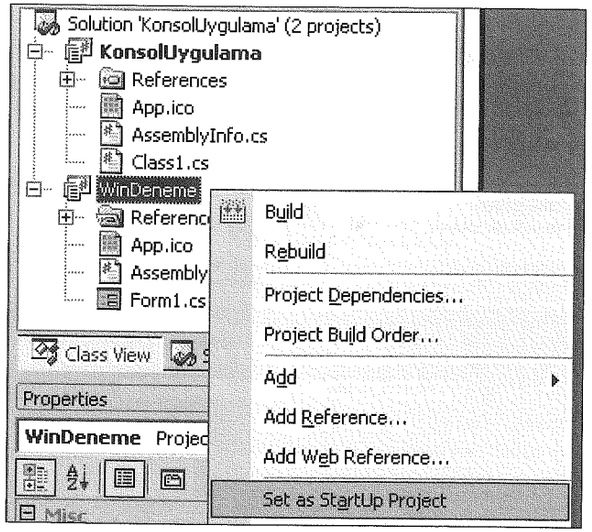


Daha önce görmüş olduğumuz "New Project" ekranından farklı olarak altta "Add to Solution" ve "Close Solution" diye iki seçeneğin bulunduğunu görüyorsunuz. Eğer "Close Solution" seçeneğini seçmiş olsaydınız hali hazırda açık olan projeniz kapatılıp yeni bir "Solution" içinde yeni bir Windows uygulaması oluşturulmuş olurdu. Bu yüzden "Solution"a yeni bir proje eklemek için "Add to Solution" seçeneğini seçmiş olmamız gerekir.

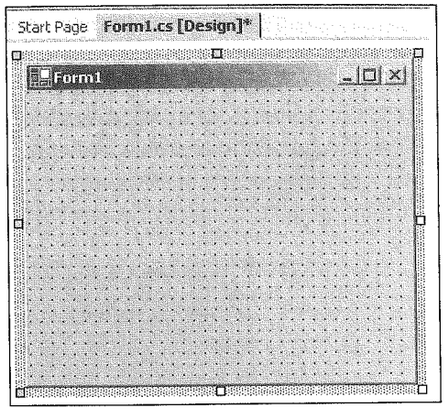
Proje eklendikten sonra "Solution Explorer" penceresi yandaki gibi değişecektir.



Birden fazla proje ile çalışırken dikkat etmemiz gereken nokta başlangıç projesinin ayarlanmasıdır. "Solution" daha sonra göreceğimiz "Build" ile derlendiği zaman "Solution"daki bütün projeler derlenir ancak yine daha sonra göreceğimiz "Start Without Debugging" seçeneğini seçtiğimizde çalışma zamanında sadece bir projeye ait uygulama devreye girer. Peki, bu uygulama hangisidir? Varsayılan olarak bu ilk eklediğiniz projedir. Ancak "Solution Explorer" penceresindeki herhangi bir pencereye sağ tıklayıp "Set as Startup Project" seçeneğini tıkladığınızda başlangıç projesi değiştirilmiş olur. Örneğin yeni oluşturduğumuz WinDeneme projesini aşağıdaki gibi başlangıç projesi yapabiliriz.

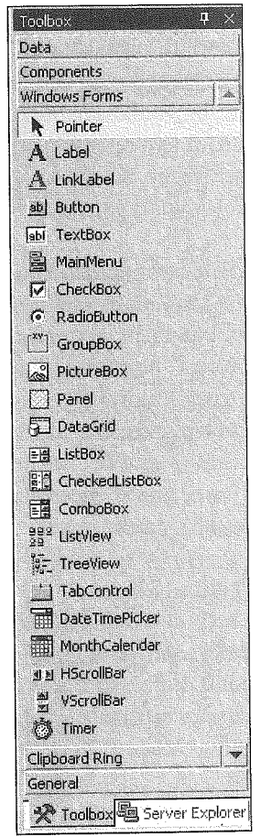


Şimdi isterseniz yeni eklemiş olduğumuz Windows uygulamasına ilişkin bazı özellikleri inceleyelim. Her şeyden önce bir konsol uygulamasından farklı olarak aşağıdaki gibi bir pencere tasarım ekranı oluşturulmuştur.

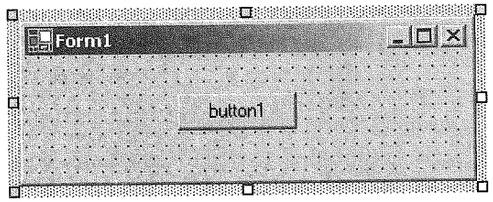


Oluşturulan bu pencere uygulama çalıştırıldığında görünecek ekrandan başka bir şey değildir. Oluşturulan bu form üzerine çeşitli kontroller konularak kompleks yapıda Windows uygulamaları geliştirebilmemiz mümkündür. Form üzerine konulacak kontrollerin listesi yanda gösterilen ToolBox penceresinin Windows Forms sekmesinde mevcuttur.

Eğer ToolBox penceresini göremiyorsanız "View" menüsünden "ToolBox" sekmesini seçerek pencereyi görünür hale getirebilirsiniz. İsterseniz Ctrl+Alt+X tuş kombinasyonlarını kullanarak da ToolBox penceresini görünür hale getirebilirsiniz.

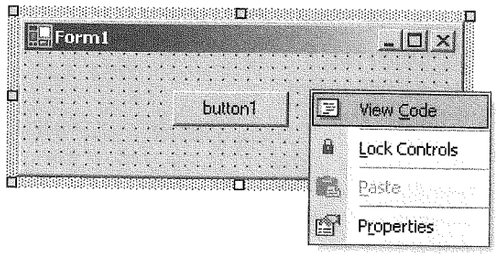


ToolBox menüsünde bulunan elemanlar kontrol olarak adlandırılmaktadır. Buradaki kontrollerin her biri .NET sınıf kütüphanesinde birer sınıf ile temsil edilirler. Bir sonraki başlık altında bu kısmı daha ayrıntılı işleyeceğiz ancak şimdilik ToolBox penceresinden bir "Button" kontrolüne çift tıklayarak kontrolün formun üzerine yerleşmesini aşağıdaki gibi sağlayın.



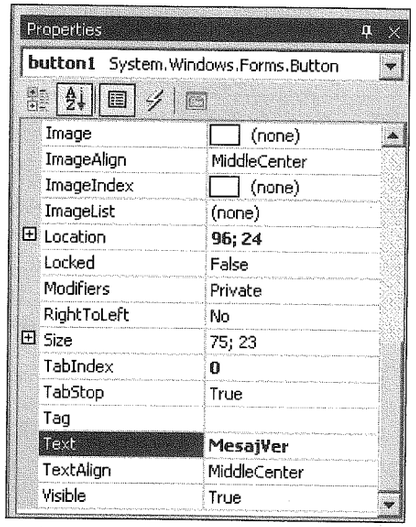
Formun kenarlarında bulunan tutamaçlarla oynayarak formun boyutlarını dilediğiniz gibi ayarlayabilirsiniz. Tabi ayarladığınız bu özellikler kaynak koda otomatik olarak yansımaktadır. Aynı şekilde button1 kontrolünü de fare ile taşıyarak formun istediğiniz yerine kontrolü yerleştirebilirsiniz.

"button1" kontrolünün üzerinde varsayılan olarak "button1" yazdığına dikkat edin. Bu özellik tamamen "Windows Application" proje şablonunun sağladığı bir özelliktir. Kaynak koda baktığınızda button1 isimli bir nesnenin oluşturulduğunu ve bu nesnenin metin özelliğinin "button1" yazısı olacak şekilde ayarlandığını görürsünüz. Peki, kaynak koda nasıl bakacağız? Form üzerinde herhangi bir noktaya aşağıdaki gibi sağ tıklayıp "View Code" sekmesini seçerseniz oluşturulan şablonla ilgili kaynak kodlar görülecektir. Form tasarımı dışındaki bütün işlerimizi bu kaynak kod üzerinde yapacağız.



Şimdilik kaynak kodun içeriği bizi pek ilgilendirmiyor. Bir sonraki başlık altında zaten bu kodlar ayrıntılı olarak anlatılacaktır. Bizim için önemli olan Visual Studio.NET geliştirme ortamının bizim için sağladığı kolaylıkları görmektir.

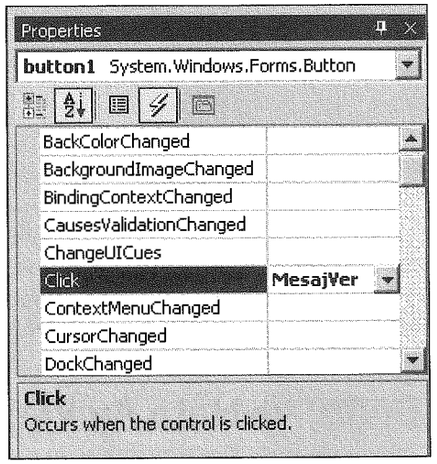
Windows kontrollerinin özelliklerini çalışma zamanında kod ile değiştirebileceğimiz gibi tasarım aşamasında aşağıdaki "Properties" penceresinden de değiştirebiliriz. Örneğin forma yerleştirdiğimiz button1 kontrolünün üzerinde yazan "button1" yazısını "MesajVer" şeklinde değiştirmek için "Properties" penceresinde yandaki gibi değişiklik yapılır.



"button1" kontrolü ile ilgili özelliklerin "Properties" ekranında görülmesi için form üzerinde "button1" kontrolünün seçili olması gerekmektedir. "button1" kontrolü daha sonra göreceğimiz gibi .NET sınıf kütüphanesinde Buton isimli bir sınıfla temsil edilir. Buton sınıfının Text özelliği kontrolün üzerindeki yazıyı temsil etmektedir. Bu yüzden çalışma zamanında bir Buton kontrolüne ilişkin Text özelliği değiştirildiğinde Buton nesnesi üzerinde görülen yazı değişecektir.

Ayrıca "Properties" penceresinde koyu bir şekilde görülen özelliklerin de şablon tarafından otomatik olarak değiştirildiğini bilmeniz gerekir.

Kontrollerle ilgili olayları görmek için "Properties" penceresindeki ⚡ işaretine tıklayın. Açılacak listede kontrole ilişkin bütün olaylar listelenecektir. Bir olay oluştuğunda hangi metodun çağrılacağı ise olay isminin yanındaki boşluğa yazılır. Örneğin "button1" nesnesine tıklandığında MesajVer() isimli metodun çağrılması için olay listesinde aşağıdaki gibi bir değişiklik yapılır. Bir Buton nesnesinin tıklanma olayı "Click" olayı ile temsil edilir. Bu konular bir sonraki kısımda anlatılacağı için şimdilik sizin dikkat etmeniz gereken sadece tasarım aşamasının ve Visual Studio.NET geliştirme aracının sağladığı imkanları öğrenmektir.

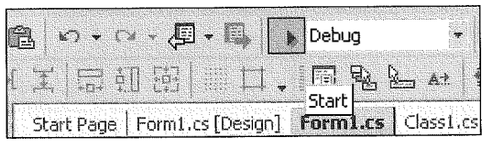


Şimdi sıra geldi MesajVer() isimli metodun içini yazmaya. Kaynak kodda MesajVer() metoduna gelmek için olay listesinden Click olayına çift tıklayın ve metodun içini aşağıdaki gibi değiştirin.

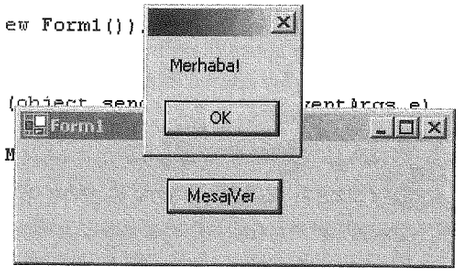
```
private void MesajVer(object sender, System.EventArgs e)
{
        MessageBox.Show("Merhaba!");
}
```

MessageBox sınıfının statik Show() metodu parametre olarak aldığı yazıyı bir mesaj kutusu şeklinde ekranda gösterir. Şimdi isterseniz programı çalıştırıp ne oluşturduğumuza bakalım.

Programı çalıştırmak için aşağıdaki resimde de görülen ok şeklindeki düğmeye tıklayın ya da F5 tuşuna basın.



Tahmin edeceğiniz üzere ilk başta aşağıdaki gibi bir form ekranda görülecek. Form üzerindeki Botun kontrolüne basıldığında ekrana bir mesaj kutusu içinde "Merhaba!" yazıldığı görülecek.



Diğer bir başlığa geçmeden "ToolBox" penceresindeki diğer sekmeleri inceleyelim.

Data: Veri kaynaklarına erişmek ve veri kaynaklarındaki bilgileri elde etmek için kullanılan çeşitli kontrolleri barındırır.

Components: Çeşitli Windows servisleri ile ilgili kontroller bu sekme altında bulunur.

Oluşturma (Build) ve Hata Ayıklama (Debug)

Biz geliştiriciler olarak programlarımızı yazarken ihtiyacını en sık duyduğumuz kavramlar test ve hata ayıklamadır. Test ve hata ayıklama (debug) için .NET geniş imkanlar sağlamaktadır. .NET ortamındaki debugger (hata ayıklayıcı) programı sayesinde test aşamasında programın içinde adım adım ilerleyip nesnelerin her adımdaki durumlarını inceleyebiliriz. Hata ayıklama işlemini anlatmaya başlamadan önce Visual Studio.NET ortamında bir projenin nasıl derlenebileceğini inceleyelim. Visual Studio.NET projelere ait bütün kaynakların birleştirilip(build) derlenmesi için iki seçenek sunmaktadır. Bunlar **Debug** ve **Release** seçenekleridir. Debug seçeneğini seçerek bir programı derlediğimizde Debugger programının çeşitli ihtiyaçlarını karşılamak için çalıştırılabilir dosyaya birçok eklenti yapılmaktadır. Örneğin çalışma zamanında programda adım adım ilerlemek için Debugger programının program içindeki nesnelerden ve değişkenlerden haberdar olmak zorundadır. Debug seçeneği ile derlenen çalıştırılabilir dosyaların boyutunun büyük olmasının sebebi de budur. Ayrıca Debug seçeneği ile derlenen çalıştırılabilir dosyalarda herhangi bir kod optimizasyonu yapılmamaktadır. Bildiğiniz gibi JIT derleyicileri IL kaynak kodunun içinde çeşitli optimizasyon işlemleri yapar. Optimizasyon kavramına bir örnek vermek gerekirse;

```
int a = 10;
Consele.Write(a);
```

deyimleri JIT derleyicisi tarafından aşağıdaki koda çevrilebilir.

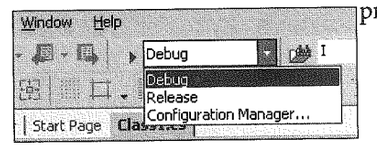

```
Consele.Write(10);
```

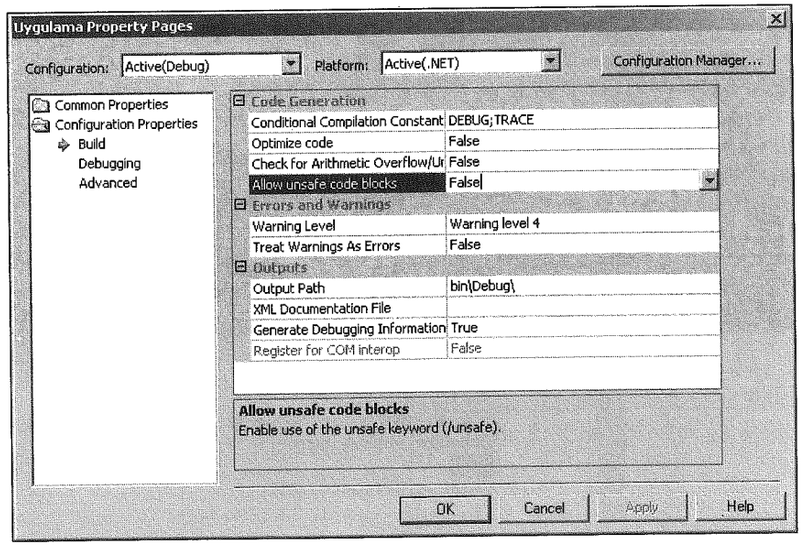
JIT derleyicisinin bu şekilde bir optimizasyon yaptığı garanti değildir. Çünkü Microsoft firması, bu kitabın yazıldığı tarihlerde JIT derleyicisinin yaptığı optimizasyonlar hakkında herhangi bir yazılı belge yayınlamamıştır. Diğer bir derleme şekli olan **Release** seçeneği ise programın geliştirme aşaması bittiğinde müşteriye sunulucak programı oluşturmak için kullanılır. Release seçeneği ile derlenen kaynak kodlarda herhangi bir Debeggur bağımlı bir sembol ya da işaret bulunmaz. Ayrıca JIT derleyicisi çalıştırılabilir dosya üzerinde kod optimizasyonu yapmaktadır.

Bir solution ortamındaki projelerin tamamının ya da üzerinde çalışılan projenin derlenmesi Build menüsündeki seçeneklerden yapılır. Programın Debugger ile çalıştırılmaması için CTRL+F5 tuş kombinasyonunun kullanılması ya da Debug menüsünden "Start Without Debugging" seçeneğinin seçilmesi gerekir. Programın Debugger ile çalıştırılması için ise F5 tuşuna ya da Debug menüsünden Start seçeneğine tıklayabilirsiniz.

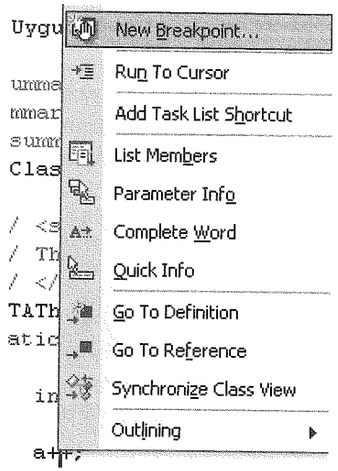
Debug ve Release seçenekleri değişik şekillerde seçilebilir. Birincisi aşağıda görüldüğü gibi derleme işlemi yapmadan önce Debug ve Release seçeneklerinden birini seçmektir.

Ayrıca "Solution Explorer" penceresinden projeye sağ tıklayıp proje ile ilgili çeşitli özellikleri değiştirebileceğimiz aşağıdaki ekran görüntüsünü elde edip Debug ya da Release seçeneğini değiştirebiliriz.





Yukarıdaki "Property Pages" penceresini Project menüsündeki Properties seçeneğini tıklayarak da açabiliriz. Bu pencereden projenin derlenmesi ile ilgili ayarlar ve genel özellikler değiştirilebilir. Penceredeki ağaç şeklindeki menüyü kullanarak proje ile ilgili diğer özellikleri de değiştirebilmek mümkündür. Debug ve Release seçeneklerini bu pencerede en üstte bulunan "Configuration Manager" düğmesine tıklayarak açılacak pencereden yapabiliriz.



Derleme işlemlerini yaptıktan sonra asıl önemli konu olan hata ayıklama işleminin nasıl yapıldığı üzerinde duralım. Hata ayıklama işlemleri genellikle "Breakpoint" dediğimiz satırların varlığı ile gerçekleştirilir. Programın akışı "Breakpoint" olarak işaretlenen kod satırlarına geldiğinde program durdurulur ve Debugger programı devreye girer. Kaynak koddaki herhangi bir satıra "Breakpoint" koymak için ilgili satıra geldiğinizde fare ile sağ tıklayıp yanda verilen resimdeki gibi "New Breakpoint" seçeneğini seçmeniz gerekir.

Bir satıra "Breakpoint" yerleştirmek için kullanılabilecek diğer bir yol ise ilgili satırdayken F9 tuşuna basmaktır. "Breakpoint" eklenen satırlar aşağıdaki gibi işaretlenir.

```
        static void Main(string[] args)
        {
            int a = 10;

            a++;

            Console.WriteLine(a);
        }
```

Şimdi aşağıdaki gibi bir program yazıp şekilde belirtildiği gibi bir "Breakpoint" yerleştirdikten sonra programı Debugger ile çalıştırıp sonuçları inceleyelim.

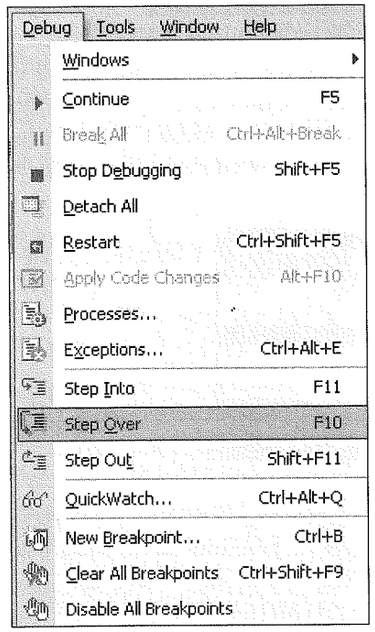
```
using System;

class Class1
{
    static void Main(string[] args)
    {
        int a = 10,b=0;
        a++;
        b = a - 5;
        Console.WriteLine(b);
        b++;
        Console.WriteLine(a);
    }
}
```


Programı Debugger ile derlemek için F5 tuşuna basmayı unutmayın. Eğer Ctrl+F5 tuş kombinasyonunu kullanırsanız "Breakpoint" işaretli noktalarda Debugger programı devreye girmeyecektir.

Program çalıştırıldığında "Breakpoint" ile işaretlenmiş noktaya kadar her şey normal gidecektir. Ancak "Breakpoint" noktalarına gelindiğinde program durdurulacak ve aşağıdaki gibi bir ekran görüntüsü ile karşılaşılacaktır.

```
            int a = 10,b=0;
            a++;
            b = a - 5;
            Console.WriteLine(b);
            b++;
            Console.WriteLine(a);
        }
    }
```

Bu noktada Visual Studio.NET ortamındaki Debug menüsünün içeriğinin aşağıdaki gibi değiştiğini göreceksiniz. Bu menüde Debugger programı ile ilgili çeşitli komutlar bulunmaktadır. Örneğin program içinde adım adım ilerlemek için "Step Over" seçeneğini seçebilirsiniz.

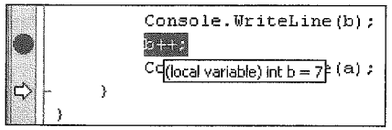


"Step Over" seçeneğini iki defa seçtikten sonra kaynak koddaki sarı ile belirlenen okun iki satır ilerlediğini aşağıdaki gibi görmeniz gerekir.

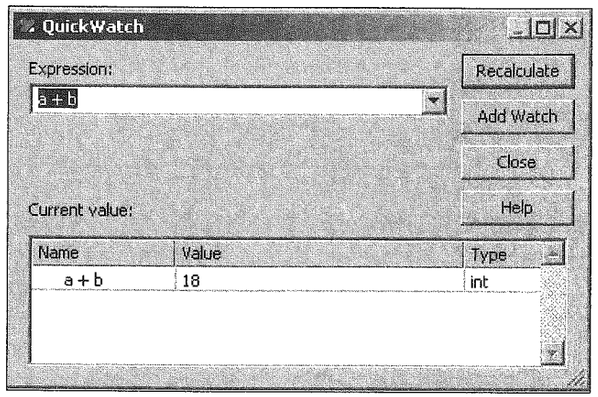
```
            b++;
            Console.WriteLine(a);
        }
    }
```

Yandaki Debug menüsünde bulunan "Continue" seçeneğini tıklarsak programın akışı diğer bir "Breakpoint" noktasına kadar devam eder. Menüdeki "Step Into" seçeneği ile kaynak kodda belirli bir metodun içine de geçmeniz mümkündür. Debugger programı aktif durumdayken programın akışı içerisinde herhangi bir değişkenin değerini farenin imlecini ilgili değişkenin üzerine getirerek görebiliriz.

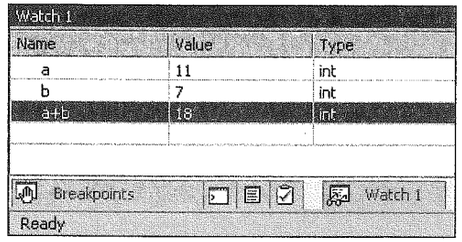
Örneğin aşağıda b değişkenin değeri gösterilmektedir.



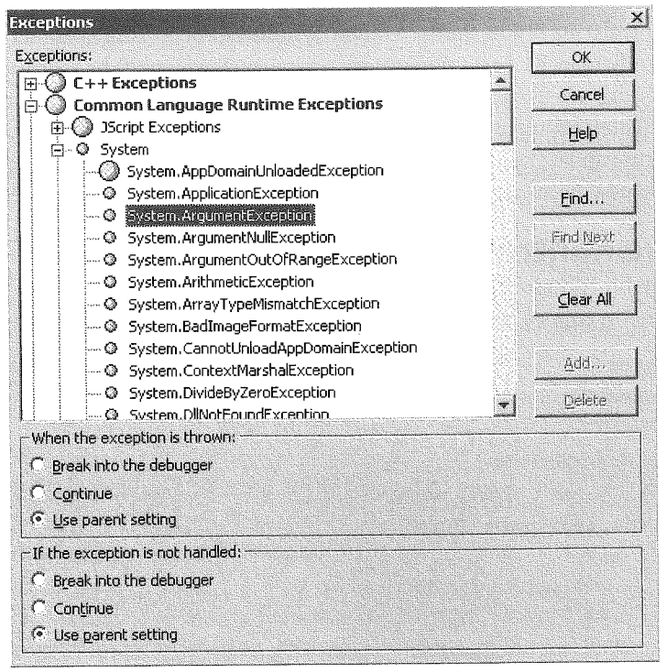
Debug menüsündeki "QuickWatch" seçeneği ile bir değişkenin değeri ile ilgili daha ayrıntılı bilgi elde edebilmemiz mümkündür. Üstelik aşağıda da gördüğünüz gibi "Quick Watch" penceresinden o anki değişkenleri kullanarak bir ifadeyi hesaplayabiliriz. Aşağıda programın akışı "Breakpoint" noktasına geldiği anda a ve b değişkenlerinin toplam değeri "Quick Watch" penceresindeki ifade ile hesaplanmıştır.



"QuickWatch" penceresinde hesaplanan ifadenin anlık olarak takip edilebilmesi ve ifade hakkında daha detaylı bilgi elde etmek için ifadeyi "QuickWatch" penceresindeki "Add Watch" düğmesini kullanarak Visual Studio.NET ortamındaki "Watch" penceresine ekleyebilirsiniz. Bu ifadeyi "Watch" penceresine ekledikten sonra "Watch" penceresinin görüntüsü aşağıdaki gibi olmaktadır.



Hata ayıklama ile ilgili diğer bir önemli nokta ise Debugger programının istisnai durumlarda ne şekilde davranacağını belirlemektir. Debugger programının bu durumlardaki davranışlarını belirlemek için Debug menüsünden Exception seçeneğini seçip aşağıdaki ekran görüntüsünün çıkmasını sağlayın. Aynı işlemi Ctrl+Alt+E tuş kombinasyonları ile de yapmak mümkündür.



İlgilendiğiniz istisnai durumu seçtikten sonra istisnai durum nesnesinin fırlatılması ve yakalanması durumunda Debugger programının ne yapması gerektiğini en altta bulunan seçeneklerden yapabilirsiniz. Eğer Continue seçeneği seçilirse ilgili istisnai durum oluştuğunda ve istisnai durum yakalandığında programın akışı devam eder. "Use parent setting" seçeneği ise birbirinden türemiş olan Exception sınıflarında türemiş sınıfın türediği sınıfa ait özellikleri alması için kullanılır.

Windows Uygulamaları Geliştirme

Windows uygulamaları, özellikle son kullanıcıya hitap eden grafik arayüzlü programlardır. Her ne kadar son dönemlerde web tabanlı programlama modeli yaygınca kullanılsa da Windows işletim sisteminde yalın bir şekilde ya da diğer servislerle ilişkili bir şekilde çalışacak grafik arayüzlü programlara da ihtiyaç duyulmaktadır. .NET sınıf kütüphanesi sayesinde Windows formlarına yönelik program geliştirmek gerçekten çok kolaylaşmıştır. Form dediğimiz nesneler, Windows tabanlı programlarda gördüğümüz arayüzlerden başka bir şey değildir. Windows formlarına yönelik program geliştirmek Visual Studio.NET ile çok daha esnek hale getirilmiştir. Bu durumu herhangi bir metin editöründe bu tür programları geliştiremeyeceğimiz anlamına gelmez. Windows formlarına ve kontrollerine yönelik çeşitli temel bilgileri daha yakından kavrayabilmek için ilk etapta yine bir metin editörünü kullanıp komut satırı derleyicisinde derleme yapacağız.

Windows formlarına ilişkin sınıflar **System.Windows.Forms** isim alanında bulunmaktadır. Bu sınıflardan ilk başta öğrenmemiz gereken Form sınıfıdır. Form sınıfı program çalıştırıldığında ekranda boş bir pencerenin gösterilmesi için gereken özellikleri belirler. Geliştireceğimiz uygulamalarda bu Form sınıfından yeni sınıflar türetip daha gelişmiş form yapıları tasarlayabiliriz. Form sınıfının birçok özelliği ve birçok metodu ve aynı zamanda birçok olayı vardır. Bu elemanlardan önemli olanlarına değineceğiz.

Form ve Kontrollerin Kullanımı

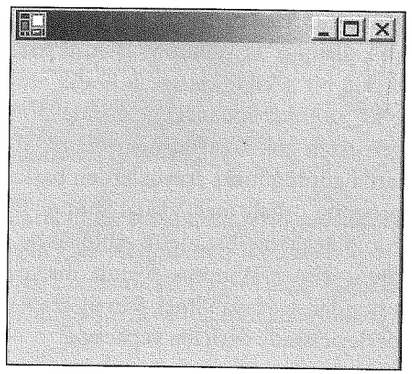
Şimdi basit bir Windows penceresinin nasıl oluşturulacağına bakalım. Aşağıdaki programı metin editörünüzde yazın.

```
using System;
using System.Windows.Forms;

namespace WindowsForm
{
        class IlkForm : Form
        {
                static void Main()
                {
                        Application.Run(new IlkForm());
                }
        }
}
```

Programı yazdıktan sonra komut satırı derleyicisinde aşağıdaki gibi derleyin.

```
csc /t:winexe /r:System.Windows.Forms IlkForm.cs
```

/t:winexe argümanı ile derlenecek programın Windows form tabanlı bir uygulama olduğu bildiriliyor. Dolayısıyla oluşturulan exe uzantılı dosya çalıştırıldığında konsol ekranı gelmeden direkt uygulama penceresi aşağıdaki gibi açılacaktır.



Form yandaki şekilde en basit haliyle gösterilmiştir. Herhangi bir başlığı ya da üzerinde herhangi bir kontrolü bulunmamaktadır. Tek özelliği "Minimize" ve "Maximize" düğmelerinin olmasıdır. Tabi bunlar görünen özellikler. Henüz daha Form sınıfının kabiliyetlerini görmedik. Şimdi yukarıda yazmış olduğumuz IlkForm.cs isimli dosyadaki kodlara yakından göz atalım.

```
class IlkForm : Form
```


Satırında Form sınıfından yeni bir sınıf türetiyoruz. Dolayısıyla Form sınıfının bütün özellikleri bizim bildirdiğimiz IlkForm sınıfında mevcut olacak. IlkForm sınıfının bildirimine yeni bir özellik eklemediğimiz için IlkForm ile Form sınıfları aynı özelliklerdedir. Uygulamayı bir form üzerinden başlatmak için System.Windows.Forms isim alanında bulunan **Application** sınıfı kullanılmaktadır. Bu sınıf yardımıyla uygulama bazında çeşitli işlemler yapılabilir. Örneğin bir uygulamanın başlatılması ya da sonlandırılması gibi. Yeni bir uygulama başlatmak için statik **Run()** metodu kullanılır. Run() metodunun birkaç aşırı yüklenmiş versiyonu vardır. Bizi ilgilendiren versiyonu parametresi Form sınıfı türünden olanıdır. IlkForm sınıfı, Form sınıfından türediği için Form türünden referanslara IlkForm nesne referansları atanabilir. Dolayısıyla Run() metoduna parametre olarak IlkForm sınıfı referanslarını verebiliriz. Main() metodundaki

```
Application.Run(new IlkForm());
```

deyimi ile bu işlem yapılmaktadır. Bu deyimi aşağıdaki gibi de yazmamız doğrudur.

```
IlkForm YeniForm = new IlkForm();
Application.Run(YeniForm);
```

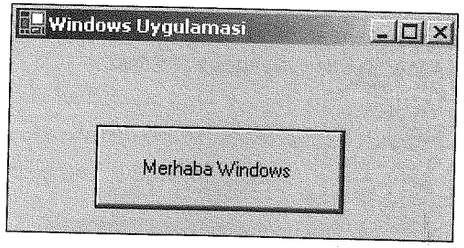
Şimdi bu programımızı biraz daha geliştirerek form üzerine bir buton ekleyelim. Form üzerine bir buton ekleyebilmemizi sağlayan Form sınıfının Controls isimli özelliğidir. Controls özelliği System.Windows.Forms isim alanında bulunan Control sınıfı türünden bir koleksiyondur. Controls özelliğinin Add() metodu ile Form nesnesine yeni kontroller eklenmektedir. Şimdi bu yöntemi kullanarak Form üzerine birkaç özelliği bulunan bir buton ekleyelim.

```
using System;
using System.Windows.Forms;

namespace WindowsForm
{
        class IlkForm : Form
        {
                private Button IlkButon;
                public IlkForm()
                {
                        IlkButon = new Button();

                        IlkButon.Text = "Merhaba Windows";
                        IlkButon.Location = new System.Drawing.
Point(50,50);
                        IlkButon.Cursor = Cursors.No;
                        IlkButon.Size = new System.Drawing.
```

```
Size(150,50);

                this.Text = "Windows Uygulaması";

                this.Controls.Add(IlkButon);
            }

            static void Main()
            {
                Application.Run(new IlkForm());
            }
        }
}
```

Programdan da görüleceği üzere IlkForm sınıfında private üye elemanı olacak şekilde **Buton** türünden bir eleman bildirilmiştir. IlkForm sınıfının varsayılan yapıcı metodunda form üzerine eklenecek kontrollerle ilgili işlemler yapılmaktadır. IlkButon referansı oluşturulduktan sonra Buton sınıfının çeşitli özellikleri değiştiriliyor. Buton sınıfının Text özelliği butonun üzerindeki yazıyı, Location özelliği butonun form üzerindeki konumunu, Cursor özelliği fare buton üzerine geldiğinde farenin imlecinin şeklini, Size özelliği ise buton kontrolünün boyutlarını belirtmektedir. Location özelliğinin Point türünden olması ve Size özelliğinin Size sınıfı türünden olmasından dolayı System.Drawing isim alanındaki sınıflar kullanılmıştır. Aynı şekilde Form sınıfının Text özelliği de formun başlığını değiştirmektedir. Son olarak daha önce dediğimiz gibi Form sınıfının koleksiyon olan Controls özelliğinin Add() metodu ile IlkForm penceresinde görünmek üzere IlkButon butonu yerleştiriliyor. Eğer bunu yapmamış olsaydık IlkButon nesnesi form üzerinde görülmezdi.

Şimdi bu kodu bir önceki programı derlediğimiz gibi derleyin ve programı çalıştırın. Programı çalıştırdığınızda yandaki gibi bir görüntü elde etmeniz gerekir.



Farenin imlecini butonun üzerine getirdiğinizde imlecin değiştiğini göreceksiniz. Butona tıkladığınızda ise herhangi bir olayın meydana gelmediğini fark etmiş olabilirsiniz. Elbette, şu an butona tıklandığında yapılması gereken işleri bildirmedik. Bundan sonraki amacımız da bu zaten. Butona tıklandığında ekrana mesaj kutusu çıkarmayı sağlayacağız.

Kitabın geçmiş bölümlerinde temsilcileri ve olayları anlatırken bu yapıların görsel arayüz içeren programlarda sıkça kullanıldığını söylemiştik. Evet, gerçekten de öyle. Bir butona tıklandığında belirli bir metodu çalıştırmanın en güzel yöntemi olayları kullanmaktır. Hatırlarsanız olaylar bir veri türüydü. Olayları (events) anlatırken olayı tetikleyen ve olayın oluşması sonucunda çalıştırılacak metodu biz ayarlamıştık. An-

cak Windows kontrollerine yönelik olaylar zaten sınıf bildirimleri içinde tanımlanmıştır. Bize düşen bu olayların oluşması sonucunda çağrılacak metotları bir temsilci yardımıyla belirlemektir. Windows kontrollerinin her biri çok sayıda olay içermektedir. Örneğin Form sınıfının 50'nin üzerinde olayı vardır. Form sınıfının olaylarına örnek verecek olursak; formun üzerine çift tıkladığımızda DoubleClick olayı, form belleğe yüklendiğinde Load olayı, form kapatıldığında Close olayı meydana gelir. Her bir olaydan sonra çalıştırılacak metot **EventHandler** isimli temsilci ile belirtilir.

Aşağıdaki programda buton nesnesinin Click olayından sonra çalıştırılacak metot EventHandler temsilcisi ile belirlenmektedir. EventHandler temsilcisinin temsil ettiği metotların prototipi hatırlarsanız aşağıdaki gibiydi.

```
void Metot(object o, EventArgs e)
```

EventArgs, System isim alanında bulunan bir temsilcidir.

Olayların kullanımını görmek için aşağıdaki programı yazın ve derleyin. Bu programın bir öncekinden farkı IlkButon nesnesinin Click olayının belirlenmesi ve Click olayı meydana geldiğinde çağrılacak metodun bildirilmesidir.

```
using System;
using System.Windows.Forms;

namespace WindowsForm
{
        class IlkForm : Form
        {
                private Button IlkButon;

                public IlkForm()
                {
                        IlkButon = new Button();
                        IlkButon.Text = "Merhaba Windows";
                        IlkButon.Location = new
                                        System.Drawing.Point(50,50);
                        IlkButon.Cursor = Cursors.No;
                        IlkButon.Size = new System.Drawing.
Size(150,50);

                        IlkButon.Click += new
EventHandler(ButonTıklandı);
```

```
            this.Text = "Windows Uygulaması";

            this.Controls.Add(IlkButon);
        }

        static void Main()
        {
            Application.Run(new IlkForm());
        }

        void ButonTıklandı(object o, EventArgs a)
        {
            MessageBox.Show("Butona tıklandı!");
        }
    }
}
```

Önce programı çalıştırıp sonucunu görelim ardından da ne yaptığımıza bakacağız. Programı çalıştırıp butona tıkladığınızda yandaki gibi bir ekran görüntüsü ile karşılaşırsınız.

Bu programdaki en kritik nokta,

```
IlkButon.Click += new EventHandler(ButonTıklandı);
```

deyimidir. Bu deyim ile IlkButon nesnesinin Click olayı gerçekleştiğinde ButonTıklandı() metodunun çağrılacağı belirtilmektedir. Bunun için System isim alanında bulunan ve önceden tanımlanan EventHandler temsilcisi kullanılmaktadır.

Gördüğünüz gibi grafik arayüzlü programlar geliştirmenin hiçbir zorluğu bulunmamaktadır. Bilmeniz gereken tek şey C# dilini etkin kullanmak ve .NET sınıf kütüphanesindeki kontrollere ikişkin sınıfların elemanlarını MSDN dokümanlarından öğrenmektir. Bu kitabın temel amacı C# dilini öğretmek olduğu için Windows uygulamalarında kullanılan bütün kontroller bu kitapta anlatılmayacaktır.

Visual Studio.NET ile Form Tabanlı Çalışmak

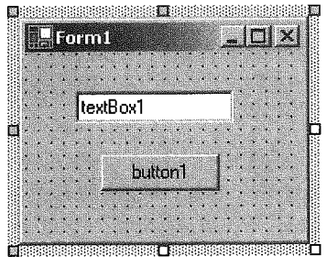
Şimdiye kadar anlattığımız yöntemlerle size Windows programlama konusunda temel bilgileri vermek amaçlanmıştı. Çoğu zaman form tabanlı uygulama geliştirmek için Visual Studio.NET'in Windows Application şablonunu kullanacağınızdan eminim. Bu proje şablonu ile Windows kontrollerine yönelik görsel tasarım yapılabilmektedir. Örneğin bir butonu form üzerine yerleştirmek için yapmanız gereken tek iş ToolBox penceresinden Buton seçeneğine çift tıklamak ya da sürekle bırak yöntemiyle formun üzerine sürüklemektir. Tasarım penceresinden yapacağınız her değişiklik otomatik olarak kaynak koda yansıyacaktır.

Şimdi Windows Application proje şablonunu daha yakından incelemek için basit bir proje oluşturalım. Visual Studio.NET proje şablonlarını anlatırken Windows Application şablonunun ne şekilde oluşturulacağını anlatmıştık. Bu yöntemle yeni bir Windows Application projesi oluşturup formu aşağıdaki gibi tasarlayın.

Formun üzerine ilgili kontrolleri yerleştirmek için ToolBox penceresini kullanmayı unutmayın. Form üzerine aşağıdaki gibi bir Button ve bir TextBox kontrolü yerleştirin.

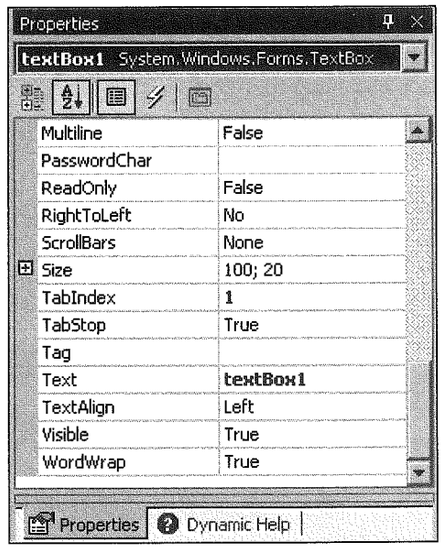
TextBox kontrolü veri girişi için özelleştirilmiş bir kontroldür.

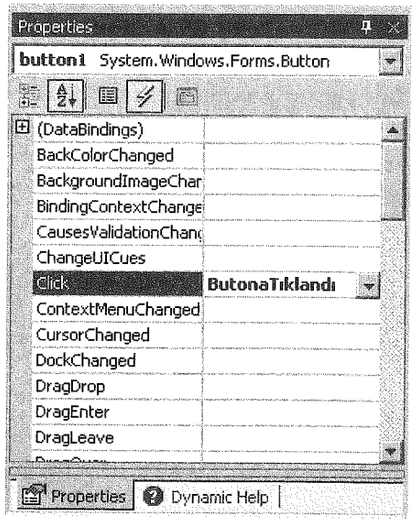
Hiçbir kod yazmadan şu ana kadar yaptığımız her şeyi gerçekleştirmiş olduk. Kontrollere ilişkin özellikleri daha önce de soylenildiği gibi Visual Studio.NET içerisindeki Properties penceresinden değiştirmek mümkündür. Bir kontrolün form üzerindeki yerini ayarlamak için kontrolü fare ile taşımak yeterlidir. Aynı şekilde kontrollerin büyüklüğünü ayarlamak için kontrolü seçip kontrolün köşelerindeki tutamaçlarla oynamak gerekir.



Tabi Properties penceresindeki Location ve Size özelliklerini değiştirerek de istediğimiz işlemleri gerçekleştirebiliriz. Properties penceresinde form üzerinde hangi kontrol seçili ise o kontrole ilişkin özellikler gösterilir. Örneğin form üzerinde TextBox1 kontrolü seçildiğinde properties penceresinin aşağıdaki gibi değiştiğini görürsünüz.

Diğer kontrollere ilişkin özellikleri görmek için Properties penceresindeki listeden ilgili kontrolü ya da form üzerinde özelliklerini görmek istediğiniz kontrolü seçmeniz gerekmektedir. Özelliklerin değiştirilmesi gibi kontrollere ait olayların da eklenmesi kolaydır. Properties penceresindeki ⚡ işaretine tıkladığınızda ilgili kontrole ait olaylar listelenir. Bir olayın meydana gelmesi için çağrılacak metot, olayın yanındaki kutucuğa aşağıdaki gibi yazılmalıdır. Olay sonrası çağrılacak metot yazılıp metoda çift tıklandığında metodun bildirimi ve EventHandler temsilcisinin otomatik olarak kaynak koda eklendiğini göreceksiniz.





Button1 nesnesinin Click olayına ilişkin metodu oluşturduktan sonra proje şablonunun oluşturduğu kaynak kodları incelemenin zamanı geldi. Bu kaynak kodun tamamını "Windows Application" proje şablonu oluşturmuştur.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace ilkWinApp
{
        /// <summary>
        /// Summary description for Form1.
        /// </summary>
        public class Form1 : System.Windows.Forms.Form
        {
                private System.Windows.Forms.TextBox textBox1;
                private System.Windows.Forms.Button button1;
                /// <summary>
                /// Required designer variable.
                /// </summary>
                private System.ComponentModel.Container components =
                                                        null;

                public Form1()
```



```
{
        //
        // Required for Windows Form Designer support
        //
        InitializeComponent();

        //
        // TODO: Add any constructor code after // Initi
                                        alizeComponent call
        //
}

/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
        if( disposing )
        {
                if (components != null)
                {
                        components.Dispose();
                }
        }
        base.Dispose( disposing );
}

#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not
                                                modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
        this.textBox1 = new System.Windows.Forms.
                                        TextBox();
        this.button1 = new System.Windows.Forms.Button();
        this.SuspendLayout();
        //
        // textBox1
        //
        this.textBox1.Location = new
        System.Drawing.Point(32, 24);
```

```
            this.textBox1.Name = "textBox1";
            this.textBox1.TabIndex = 1;
            this.textBox1.Text = "textBox1";
            //
            // button1
            //
            this.button1.Location = new
            System.Drawing.Point(48, 64);
            this.button1.Name = "button1";
            this.button1.TabIndex = 2;
            this.button1.Text = "button1";
            this.button1.Click += new
            System.EventHandler(this.ButonaTıklandı);
            //
            // Form1
            //
            this.AutoScaleBaseSize = new System.Drawing.Size
                                                    (5, 13);
            this.ClientSize = new System.Drawing.Size
                                                (176, 117);
            this.Controls.AddRange(new
            System.Windows.Forms.Control[] {
                                    this.button1,

                                    this.textBox1} );

            this.Name = "Form1";
            this.Text = "Form1";
            this.ResumeLayout(false);

        }
        #endregion

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [ STAThread]
        static void Main()
        {
            Application.Run(new Form1());
        }

        private void ButonaTıklandı(object sender,
        System.EventArgs e)
```



```
        {
            MessageBox.Show("Merhaba!");
        }
    }
}
```

Şimdi oluşturulan bu kaynak kodu biraz yakından inceleyelim. Kaynak kodu incelediğinizde Form sınıfından türetilen Form1 sınıfında textBox1 ve Button1 adında iki yeni eleman bildirildiğini görürsünüz. Bu elemanlar tasarım penceresindeki Button ve TextBox kontrollerini temsil etmektedir. Proje şablonu, her yeni kontrol eklendiğinde kontrol nesnesine bir isim vermektedir. Varsayılan olarak bu isim button1, button2, button3 şeklinde devam eder. İstersek Properties penceresinden bu varsayılan isimleri değiştirebiliriz. Kontrollerle ilgili yapılan işlemler bizim yaptığımız gibi türetilen form sınıfının yapıcı metodunda yapılmamıştır. Kontrollerle ilgili ilk işlemler Form1 sınıfının bir üye elemanı olan InitializeComponent() isimli bir metot ile yapılmaktadır. Doğal olarak bu metot Form1() sınıfının yapıcı metodunda çağrılmaktadır. Form1 nesnesi oluşturulduğunda InitializeComponent() metodu çağrılmaktadır. Form1 sınıfının yapıcı metoduna yeni deyimler eklemek şablonun yapısını bozmaz ancak görsel tasarımın yapılabilmesi için InitializeComponent() metodunun mutlaka bulunması gerekir.

`InitializeComponent()` metodunda Form sınıfına ait `SuspendLayout()` ve `ResumeLayout()` metotları kullanılmıştır. SuspendLayout() metodu dikkat ederseniz kontrollerin ilk özellikleri değiştirilmeye başlanmadan hemen önce kullanılmıştır. Bu metot ile kontrollerin özelliklerinin tamamı değiştirildiğinde ekrana çizilmesi sağlanır. Bir kontrolün her özelliği değiştirildiğinde ekrana çizdirilmesi yerine bütün özellikleri değiştirip bir defa çizdirilmesi sistem kaynaklarının etkili kullanımı açısından önemlidir. ResumeLayout() metodu görüldüğü anda eğer daha önce SuspendLayout() metodu kullanmışsa bu metottan itibaren değiştirilen özellikleri içeren yeni kontrol ekrana çizdirilir.

`InitializeComponent()` metodunda kontrollerin değiştirilen özelliklerinin ne amaçla kullanıldığına bakalım. Location özelliği daha önce de denildiği gibi kontrolün form üzerindeki konumunu belirtmektedir. Bu özellik Point türünden olduğu için yeni bir Point nesnesi oluşturulmaktadır. Kontrollerin Text özelliği her kontrolde farklı anlama gelebilmektedir. Button için Text özelliği, düğmenin üzerinde yazılacak olan yazıdır. TextBox kontrolü için de benzer bir anlam ifade eder. TextBox kontrolü ile girilen yazıyı Text özelliği temsil eder. Form sınıfının Text özelliği ise Formun başlığını temsil etmektedir. Yani çalışma zamanında Form1 sınıfının Text özelliği değiştirildiğinde formun başlığı otomatik olarak değiştirilecektir. Kontrollerin Name özelliği, kontrole bir isim vermektedir. Varsayılan bu özellik boş bir string nesnesidir. Name özelliğini kullanarak çalışma zamanında kontrolle ilgili işlemler yapılabilir. TabIndex özelliği ile kullanıcı Tab tuşuna bastığında kontrollerin aktif olma sırası belirtilmektedir. Proje şablonu ilk eklenen kontrolün TabIndex özelliğine 1 değerini vermektedir.

Ve her yeni kontrol eklendiğinde bu değer bir artırılır. İstersek kendimiz bu TabIndex özelliğini değiştirip kontrollerin Tab tuşu ile aktif olma sırasını değiştirebiliriz.

Proje şablonunun eklediği olaylar ise aşağıda görüldüğü gibidir.

```
this.button1.Click += new System.EventHandler
                                        (this.ButonaTıklandı);
```

Bu deyimi eklediği gibi ButonaTıklandı() metodunun bildirimi de yapılmıştır. EventHandler çoklu temsilci olduğu için bir olay meydana geldiğinde birden fazla metodu çalıştırabiliriz. Örneğin button1 kontrolünün Click olayı gerçekleştiğinde Metot1() ve Metot2() metotlarının çalıştırılması için kaynak kodda aşağıdaki gibi değişiklik yapmak gerekir.

```
...

this.button1.Click += new System.EventHandler(this.Metot1);
this.button1.Click += new System.EventHandler(this.Metot2);
...
...
private void Metot1(object sender, System.EventArgs e)
{
}
private void Metot2(object sender, System.EventArgs e)
{
}
```

Kontrollerle ilgili işlemler bittikten sonra Form1'in aşağıdaki gibi bazı özellikler değiştirilmiş.

```
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(176, 117);
```

ClientSize özelliği formun tasarım sırasındaki boyutunu belirler. Program çalıştırıldığında formun boyutu Size sınıfı ile belirlenen boyutta gözükecektir. ClientSize özelliğine bir formun başlık çubuğu ve kaydırma çubuğu dahil değildir. Bu özelliğin daha net anlaşılması için bir örnek verelim. Yeni bir Windows Application uygulaması açın ve form üzerine bir button kontrolü yerleştirin. Ardından formun Resize olayına aşağıdaki kodu yazıp programı çalıştırın.

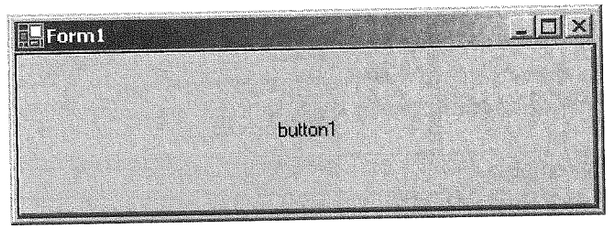
```
private void Form1_Resize(object sender, System.EventArgs e)
{
        button1.Size = this.ClientSize ;
}
```


Bu metot ile Formun boyutu her değiştiğinde button1 kontrolünün boyutu form elemanının ClientSize özelliğine atanıyor. Dolayısıyla form ne kadar büyütülürse büyütülsün button1 kontrolü formun bütün alanını kaplayacaktır. Form belleğe yüklendiğinde de button'un ekranı kaplaması için formun Load olayını aşağıdaki gibi değiştirin.

```
this.Load += new System.EventHandler(this.Form1_Resize);
```

Kontollere ilişkin birden fazla farklı olaydan sonra ortak bir metot çağrılabilir.

Eğer anlatılan işlemleri doğru bir şekilde yaptıysanız projeyi Ctrl + F5 tuş kombinasyonunu kullanarak derlediğinizde aşağıdaki gibi bir ekran görüntüsü elde etmelisiniz. Aşağıdaki ekran formun boyutunun değiştirildiği durumdaki görüntüsüdür.



InitializeComponent() metodundaki diğer bir önemli deyim ise

```
this.Controls.AddRange(new System.Windows.Forms.Control[] {
                                        this.button1,
                                        this.textBox1});
```

deyimidir. Daha önce kontrolleri formun Controls koleksiyonuna eklemek için Form sınıfının Controls elemanına ait Add() metodunu kullanmıştık. Ancak proje şablonu Add() yerine AddRange() metodunu kullanmış. AddRange() metodu parametre olarak bir kontrol dizisi almaktadır. İsterseniz Add() metodu ile bütün kontrolleri tek tek Controls koleksiyonuna ekleyebilirsiniz.

Son olarak Dispose() metodu tekrar yüklenerek kontroller için ayrılmış sistem kaynaklarının iade edilmesi için imkan yaratılıyor.

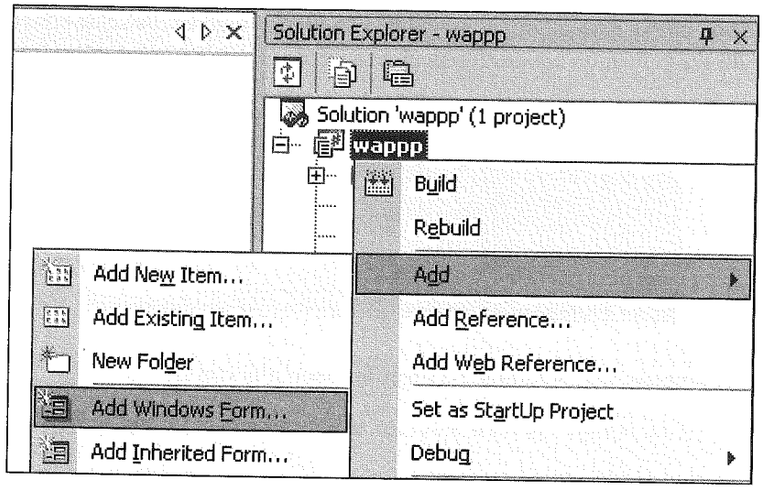
Main() metodunun STAThread niteliği bu kitapta hiç incelemediğimiz thread kavramı ile alakalıdır. Bu niteliği silerseniz programınızın çalışmasında bir sorun olmaz.

Proje şablonunun kaynak kodlara eklediği yorum satırlarını da dilediğiniz gibi silebilirsiniz.

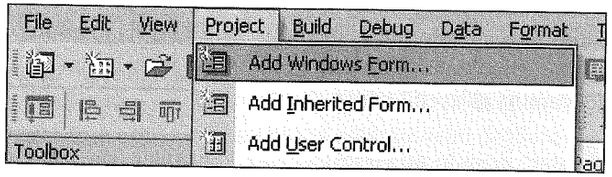
Birden Fazla Form ile Çalışmak

Gerçek uygulamaların hemen hemen tamamında bir form yerine birden fazla form ile çalışılır. Örneğin şifre korumalı bir programda ilk gelen form şifre sorgulama ekranıdır. Eğer şifre doğru girilirse bu ekran kapatılır ve yeni bir ekran açılır. Birden fazla form ile çalışmak sanıldığı kadar zor değildir. Bunun için tek yapmamız gereken Visual Studio.NET ortamındaki projemize yeni form tasarım ekranı eklemek ve kaynak kodda bazı değişiklikler yapmaktır.

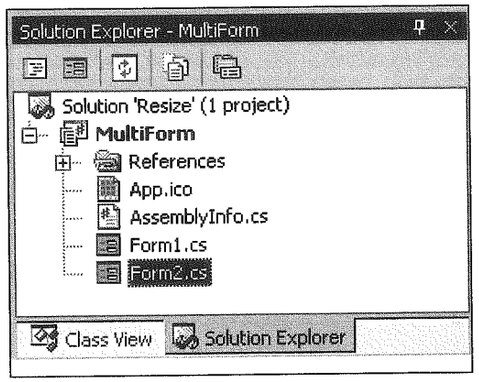
Yeni bir "Windows Application" uygulaması başlatın ve aşağıdaki gibi projeye yeni bir Form ekleyin. Projeye yeni bir form eklemek için "Solution Explorer" penceresini ya da "Project" menüsündeki "Add Windows Form" seçeneğini kullanabilirsiniz.



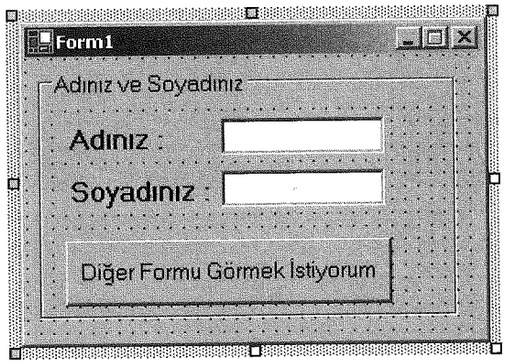
Projeye yeni bir form eklemek için diğer yöntem aşağıdaki gibi "Project" menüsünü kullanmaktır.



Yeni form eklendiğinde "Solution Explorer" penceresi aşağıdaki gibi değişir. Her bir forma ilişkin ayrı bir kod dosyası vardır. Herhangi bir forma ait kod dosyasını görmek için forma sağ tıklayıp "View Code" seçeneğini seçmeniz yeterlidir. Tekrar tasarım ekranına dönmek için ise Form1 ya da Form2'ye tıklamanız gerekir.



Yeni eklenen formun kaynak kodunu incelediğinizde Main() metodunun eklenmemiş olduğunu göreceksiniz. Bu da programın ilk çalışmaya başlaması ile Form1 penceresinin görüntüleneceği anlamına gelmektedir. Projenin oluşturulması ile açılan ilk form üzerinde aşağıdaki gibi bir tasarım yapın. Formun üzerine ToolBox penceresinden bir Button, bir GroupBox (Kontrolleri gruplamak için kullanılır), iki TextBox ile iki Label (Label kontrolü form üzerinde etiket görevi yapmaktadır) kontrolü ekleyin.



Kontrollerin çeşitli özellikleri değiştirilerek yukarıdaki tasarım ekranı oluşturulmuştur. Button kontrolünün Click olayına ilişkin metodu oluşturup aşağıdaki gibi yazın.

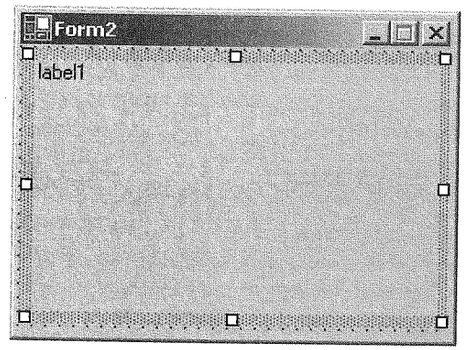
```
private void button1_Click(object sender, System.EventArgs e)
{
        if(Ad.Text == String.Empty || Soyad.Text == String.Empty)
                MessageBox.Show("Adınızı ve Soyadınızı giriniz.");
        else
        {
                Form2 frm2 = new Form2();
```

```
            frm2.Show();
            frm2.label1.Text = "Merhaba, " + Ad.Text + " " +
                                                Soyad.Text;
            frm2.label1.Text += "\n\nTarih ve Saat: " +
                                                DateTime.Now;
        }
}
```

Yukarıdaki metotta yeni bir Form2 nesnesi oluşturuluyor. Oluşturulan frm2 formuna ait label1 kontrolü Form1 üzerinden kontrol ediliyor. frm2 formunun label1 kontrolüne başka sınıflardan erişebilmek için label1 elemanı aşağıdaki gibi public olarak bildirilmelidir.

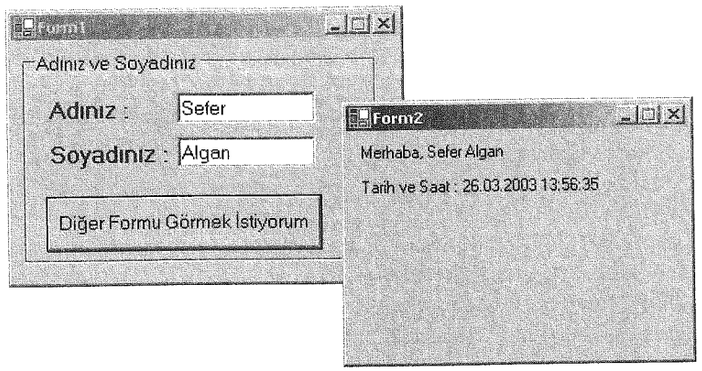
```
public class Form2 : System.Windows.Forms.Form
{
            public System.Windows.Forms.Label label1;
            ...
 ...
}
```

Projeye sonradan eklediğimiz Form2'yi aşağıdaki gibi tasarladıktan sonra projeyi Ctrl+F5 ile derleyip sonuçlarına bakalım.



Bir formdan başka bir formdaki kontrole erişmek için erişilecek kontrolün bildirimi public erişim belirleyicisi ile yapılmalıdır.

Projeyi derledikten ve ana form açıldıktan sonra Ad ve Soyad yazılıp butona tıklandığında ikinci formun gözüktüğü üstelik açılan form üzerindeki label1 kontrolünün yazısı form1 üzerinden değiştirildiği görülecektir. Programın çalışma anından örnek bir görüntü aşağıda verilmiştir.



Diğer Kontroller

Bu başlık altında Windows uygulamalarında sıkça kullanılan Button, Label ve TextBox dışındaki bazı kontroller gösterilecektir. Burada gösterilmeyen kontrollerle ilgili detaylı bilgiyi MSDN yardım dokümanlarından bulabilirsiniz. Bu başlık altında anlatılacak kontrollerin bütün özelliklerini, metotlarını ve olaylarını da yine MSDN yardım dokümanlarından bulabilirsiniz.

Mesaj Kutuları

Mesaj kutuları program çalışırken hata, uyarı ya da bilgi vermek amacıyla kullanılan pencerelerdir. Mesaj kutuları System.Windows.Forms isim alanında bulunan MessageBox sınıfı ile temsil edilir. Bir mesaj kutusunu ekranda görüntülemek için MessageBox sınıfının statik metodu olan Show() kullanılır. Show() metodunun en genel parametrik yapısı aşağıdaki gibidir.

```
DialogResult Show(string mesaj,
        string başlık,
        MessageBoxButtons a,
        MessageBoxIcon i)
```

MessageBoxButtons bir numaralandırmadır. Bu numaralandırma ile gösterilecek mesaj kutusundaki buton çeşitleri bildirilir. Bu numaralandırma 6 sembol içermektedir. Bunlar AbortRetryIgnore, OK, OKCancel, RetryCancel, YesNo, YesNoCancel sembolleridir. Sembollerin isimleri mesaj kutusunda gösterilecek butonların isimleri ile paralellik gösterir.

Son parametre olan MessageBoxIcon da bir numaralandırmadır. Bu numaralandırma mesaj kutusunda gösterilecek ikonu belirtmektedir. Her sembole ilişkin ikon farklıdır. Bu semboller Asterisk, Error, Exclamation, Hand, Information, None, Question, Stop ve Warning'dir. Bu sembollerin her birini deneyerek ne şekilde bir ikonla gösterildiğini görebilirsiniz.

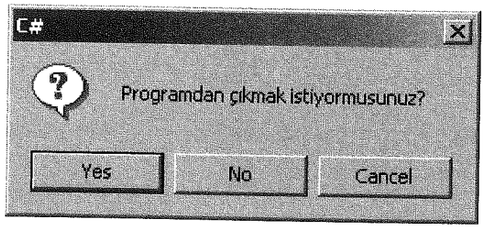
Son olarak Show() metodunun geri dönüş değerine bakalım. DialogResult türü bir mesaj kutusunda hangi butona basıldığını tutacak şekilde özelleştirilmiştir. DialogResult sınıfının diğer sabitleri Abort, Cancel, Ignore, No, Yes, None ve Retry'dır. Örneğin bir mesaj kutusu gösterilip basılan tuşun OK olup olmadığı aşağıdaki gibi test edilebilir.

```
if(MessageBox.Show("Hatalı","C#")== DialogResult.OK)
        textBox1.Text = "OK düğmesine bastınız";
```

Yukarıda Show() metodunu kullanırken 2. ve 3. parametrelerin yazılmadığına dikkat edin. Bu şekildeki bir kullanımda varsayılan olarak sadece OK butonu gösterilecektir. Bunun yanında herhangi bir ikon mesaj kutusunda belirmeyecektir. Yukarıdaki Show() metodunun sonucunda yandaki gibi bir mesaj kutusu gösterilecektir.

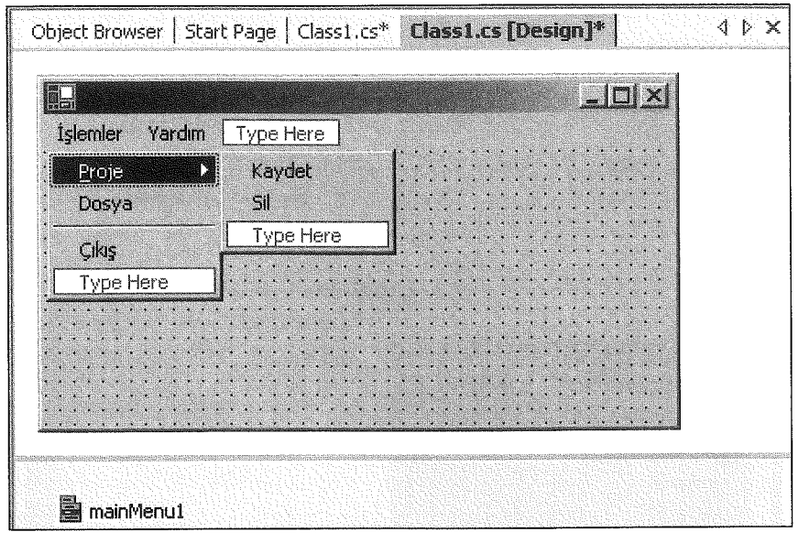
Diğer kontrollere geçmeden önce 4 parametresi de aşağıdaki gibi girilmiş bir mesaj kutusunu görelim.

```
string baslik = "C#";
string mesaj = "Programdan çıkmak istiyormusunuz?";
MessageBox.Show(mesaj,baslik,MessageBoxButtons.YesNoCancel,
        MessageBoxIcon.Question);
```



Menüler

Hemen bütün form tabanlı uygulamaların vazgeçilmez yapısı menülerdir. Menüler belirli işleri yapmak için hiyerarşik bir şekilde düzenlenebilen yapılardır. İki tip menü oluşturma imkanımız vardır. Bunlardan birincisi ana menü dediğimiz menülerdir. Ana menüler formun en tepesinde bulunan bir çoğumuzun aşina olduğu yapılardır. Bir forma ait ana menü oluşturmak için MainMenu kontrolü kullanılır. MainMenu kontrolünü ToolBox penceresinden seçip aşağıdaki gibi kolaylıkla forma ekleyebiliriz.



Her bir menü elemanının ayrı bir Click olayı vardır. İlgili menü elemanına tıklandığında yapılması gerekenleri yazmak için menü elemanına çift tıklayıp Click olayına ilişkin metodun gövdesine gidebilirsiniz.

Menü elemanlarına kısayol tuşu vermek için & karakteri kullanılır. Örneğin yukarıdaki şekilde "Proje" isimli menü elemanı tasarım penceresinde yazılırken "&Proje" şeklinde yazılmıştır. Bu yüzden form aktifken Alt+P tuşlarına basılınca Proje isimli menü elemanı seçilecektir.

MainMenu tasarımı ile ilgili diğer bir nokta ise menü elemanları arasına ayırıcı çizgi (separator) koymaktır. Bir noktaya ayırıcı çigi koymak için menü elemanına sağ tıklayıp açılacak menüden "Insert Separator" seçeneğini tıklayın.

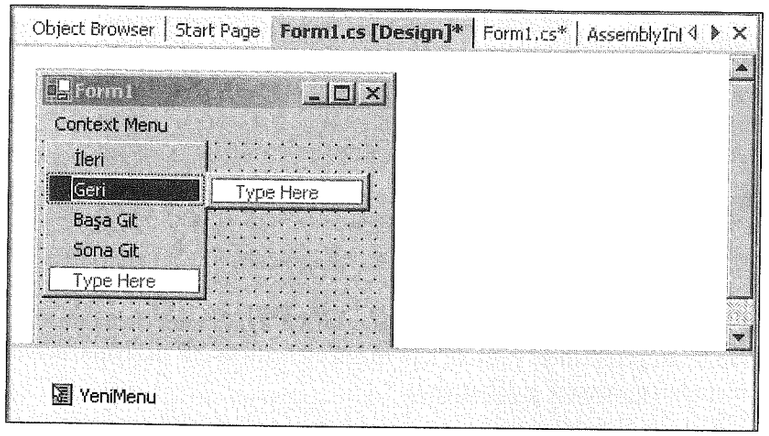
MainMenu kontrolünün ilgili elemanlarını değiştirmek ve özelliklerini görmek için elemanın üzerine farenin imleci ile tıklamak yeterlidir.

Forma ilişkin kaynak kodu incelediğinizde MainMenu nesnesinin MenuItem isimli yapılardan oluştuğunu görürsünüz.

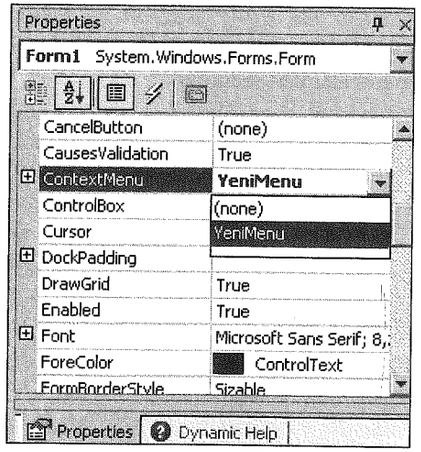
Diğer bir menü çeşidi ise form üzerindeki kontrollerle ilgilidir. Bu menü çeşidi ContextMenu kontrolü ile tasarlanır. ContextMenu kontrolünün tasarımı MainMenu ile aynıdır. ContextMenu kontrolünün tek farkı belli bir kontrole ilişkin olmasıdır. ContextMenu'nun görünmesi için farenin sağ tuşu ile ilgili kontrole sağ tıklamanız gerekmektedir. Ayrıca tasarlanan ContextMenu kontrolünün hangi elemana ilişkin olduğunuda belirtmek gerekir.

ToolBox menüsünden ContextMenu kontrolünü seçip form üzerine yerleştirin. Daha sonra bu ContextMenu kontrolünün hangi elemana ait olduğunu belirlemek için form üzerinden bir kontrol seçip Properties penceresinden ContextMenu özelliğini tasarlamış olduğumuz menü kontrolü olacak şekilde değiştirin.

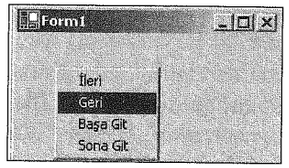
Şimdi adım adım bir ContextMenu kontrolü oluşturalım. ToolBox penceresinden ContextMenu kontrolünü bulun ve forma ekleyin. Formun yeni görüntüsü aşağıdaki gibi olacaktır.



ContextMenu kontrolünün adını (Name özelliğini) "YeniMenu" olarak değiştirin. ContextMenu kontrolüne MainMenu kontrolündeki gibi yeni menü elemanları ekleyin. YeniMenu'yü form elemanına ilişkilendirmek için form penceresi seçili iken Properties penceresinden aşağıdaki şekilde gördüğünüz gibi bir değişiklik yapın.



Projeyi bu haliyle derleyip çalıştırdığımızda herhangi bir menü göremeyiz. Ancak form penceresine sağ tıkladığımızda imlecin bulunduğu yerde aşağıdaki gibi bir menünün belirdiğini görürüz.



ContextMenu kontrolünü bir form ile ilişkilendirmek zorunda değilsiniz. Örneğin form yerine bir button kontrolü de seçebilirdik. Ancak daha çok formlarla birlikte kullanılır.

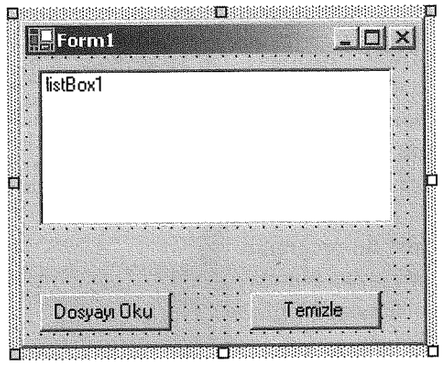
ListBox

ListBox kontrolü bir dizi yazıyı görüntülemek için kullanılır. Bu kontrolü daha önce gördüğümüz ArrayList koleksiyonunun görsel ifadesi şeklinde düşünebiliriz. ListBox kontrolünün kullanımına bir örnek verelim.

C: dizini altında aşağıdaki gibi bir txt uzantılı dosya oluşturun. Bu dosyanın her bir satırı okunup listbox kontrolüne yerleştirilecek. ListBox kontrolü de yine ToolBox penceresindedir.

```
//Okullar.txt
İstanbul Teknik Üniversitesi
Orta Doğu Teknik Üniversitesi
Yıldız Teknik Üniversitesi
Karadeniz Teknik Üniversitesi
```

Yeni bir proje oluşturup aşağıdaki gibi bir form tasarlayın.



"Dosyayı Oku" yazan butonun Click olayını aşağıdaki şekilde değiştirin.

```
private void button1_Click(object sender, System.EventArgs e)
{
        FileStream fs = File.Open(@"C:\Okullar.txt",FileMode.Open);

        StreamReader sr = new StreamReader(fs);
        string okul;
        int toplam = 0;

        while((okul = sr.ReadLine()) != null)
        {
                listBox1.Items.Add(okul);
```

```
            toplam++;
        }

        label1.Text = "Toplam okul sayısı = " + toplam;

        fs.Close();
        sr.Close();
}
```

ListBox kontrolünün Items koleksiyonu listbox kontrolündeki elemanları tutmaktadır. Items koleksiyonunun Add() metodu ile ListBox kontrolüne yeni eleman eklenmektedir. Add() metodunun parametresi object türünden olduğu için bütün temel veri türlerini koleksiyona ekleyebiliriz. Form üzerindeki bir Label kontrolü üzerine dosyadan okunan toplam satır sayısı da yazdırılmaktadır.

"Temizle" yazan botunun Click olayını ise aşağıdaki şekilde değiştirin.

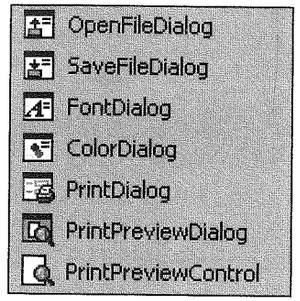
```
private void button2_Click(object sender, System.EventArgs e)
{
        listBox1.Items.Clear();
}
```

Items koleksiyonunun Clear() metodu keloksiyonu boşaltmaktadır. Items koleksiyonunun ve ListBox kontrolünün burada anlatamayacağımız birçok özelliği daha bulunmaktadır.

ListBox kontrolünü içeren bu programı çalıştırıp "Dosyayı Oku" yazan butona tıkladığınızda aşağıdaki ekran görüntüsünü elde etmeniz gerekir.

Diyalog Pencereleri

Diyalog pencereleri kullanıcıdan bilgi almak için kullanılır. Genellikle diyalog pencereleri kapanmadan ana uygulamaya geçilmez. Bu şekilde çalışan diyalog pencerelerine Modal diyalog penceresi denilmektedir. Kendimiz form sınıfından yeni sınıflar türeterek uygulamamıza özel diyalog pencereleri geliştirebiliriz. Bunun yanı sıra .NET kütüphanesinde önceden tanımlı 7 tane diyalag penceresi bulunmaktadır. Bu diyalog pencerelerinin tamamı ToolBox penceresinde alt alta yandaki gibi sıralanmıştır.



Herhangi bri diyalog penceresini kullanmak için ilgili kontrole çift tıklamanız yeterlidir. Bu diyalog kontrollerini sırayla açıklayalım.

`OpenFileDialog`: Bu kontrol standart dosya açma ekranını temsil eder. Bir dosyanın yol bilgisi elde edilerek dosya üzerinde çeşitli işlemler yapmak için kullanılır.

`SaveFileDialog`: Klasik dosya kaydetme ekranını temsil eder. OpenFileDialog kontrolünde olduğu gibi bir yol bilgisi elde etmek için kullanılır.

`FontDialog`: Windows'un klasik font seçme penceresini gösterir. Bu pencerede sistemde yüklü olan bütün yazı tipleri gösterilir. FontDialog kontrolünün çeşitli özellikleri seçilen font bilgilerini elde etmek için kullanılır.

`ColorDialog`: Windows'un standart renk seçme paletini göstererek bir renk seçimi yapılmasını sağlar. Bu kontrolün özellikleri diğer .NET sınıfları ile uyumludur. Örneğin ColorDialog kontrolünün Color özelliği herhangi bir kontrolün BackColor özelliğine atanabilir.

`PrintDialog`: Satandart yazdırma ekranını gösterir.

`PrintPreviewDialog`: Bu diyalog penceresi ile varsayılan yazıcıda ilgili dokümanın ne şekilde görüntüleneceği gösterilir.

`PageSetupDialog`: Bu kontrolün açtığı pencerede bir sayfa ile ilgili çeşitli yazıcı ayarları yapılır.

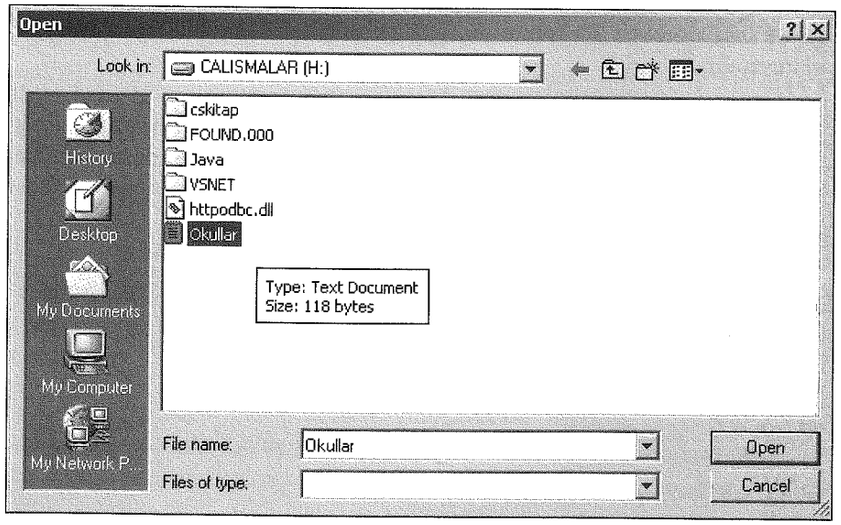
Şimdi ListBox kontrolünü anlatırken verdiğimiz örneğin yeni versiyonunu yazalım. Bu versiyonda ListBox kontrolünde listelenecek elemanları içeren dosya bir FileOpenDialog pencresi yardımıyla seçilecek.

"Dosyayı Oku" yazan butonun Click olayını aşağıdaki şekilde değiştirin.

```
private void button1_Click(object sender, System.EventArgs e)
{
        OpenFileDialog ofd = new OpenFileDialog();
        DialogResult dr = ofd.ShowDialog();

        string dosya = ofd.FileName;

        if(dr == DialogResult.Cancel)
        {
                MessageBox.Show("Bir dosya seçin...",
                "Hata", MessageBoxButtons.OK,
                MessageBoxIcon.Error);
```

```
            return;
        }

        FileStream fs = File.Open(dosya,FileMode.Open);

        StreamReader sr = new StreamReader(fs);

        string okul;
        int toplam = 0;

        while((okul = sr.ReadLine()) != null)
        {
            listBox1.Items.Add(okul);
            toplam++;
        }

        label1.Text = "Toplam okul sayısı = " + toplam;

        fs.Close();
        sr.Close();
}
```

Programı bu hali ile çalıştırdığınızda dosyanın seçilmesi için aşağıdaki gibi dosya açma ekranı açılacaktır.



OpenFileDialog pencerelerini görüntülemek için ShowDialog() metodu kullanılır. Ayrıca açılan dialog penceresinden hangi butona tıklandığını anlamak için bu metodun DialogResult türünden olan geri dönüş değerini kontrol etmek gerekir.

Diğer diyalog pencerelerinin kullanım biçimi de OpenFileDiaolg penceresinin kullanım şekline benzemektedir. Bu diyalog pencereleri ile ilgili detaylı bilgiyi her zaman söylediğimiz gibi MSDN yardım dokümanlarından buluacaksınız.

Timer Kontrolü

Windows programlama ile ilgili bu kitapta değineceğimiz son kontrol, Timer kontrolüdür. Timer kontrolü bir ya da daha fazla metodun belirli bir zaman aralığında defalarca çalıştırılmasını sağlar. Örneğin bir label kontrolü üzerine devamlı değişen saat ve tarih bilgisini yazdırmak için Timer kontrolünden faydalanırız. Bunun için geçen her saniyede label kontrolünün üzerindeki yazıyı değiştirmemiz gerekir.

Timer kontrolünün Interval özelliği timer nesnesinin kaç saniyede bir kendisi ile ilişkilendirilen metotları çağıracağını belirler. Bu özellik milisaniye cinsindendir. Dolayısıyla 1 saniyelik aralık için bu özelliğin 1000 olması gerekir. Timer kontrolü ile ilgili diğer bir önemli nokta ise Timer kontrolünün varsayılan olarak aktif olmamasıdır. Timer kontrolünün çalışmasını başlatmak için bool türünden olan Enabled özelliğinin true olarak değiştirilmesi gerekir.

Timer kontrolünün Tick isimli bir tane olayı vardır. Tick olayı Interval ile belirlenen zaman aralıklarında meydana gelir. Dolayısıyla Tick olayına ilişkin metoda yazacağımız kodlar her Interval ile belirtilen zaman aralığında çalıştırılacaktır.

Yeni bir proje açıp formun üzerine bir **Label** ve **Timer** kontrolü ekleyin. Timer kontrolünün **Enabled** özelliğini **Properties** penceresinden true, Interval özelliğini de 1000 yaptıktan sonra, **Tick** olayına ilişkin metodu aşağıdaki gibi ekleyin.

```
private void timer1_Tick(object sender, System.EventArgs e)
{
        label1.Text = DateTime.Now.ToLongTimeString();
}
```

Programı çalıştırdığınızda dijital görünümlü saatinizin çalıştığını göreceksiniz. "Windows Uygulamaları Geliştirme" başlığı oldukça geniştir. Bu uygulama alanında profesyonelliğe adım atmanız için gereken şart C# dilinin kullanımına hakim olmanızdır. Ayrıca .NET sınıf kütüphanesini kullanarak bolca deneme yapmanız gerekir. Kitabın bu konusu size bu yolda emin adımlarla ilerlemek için bir giriş noktası sundu. Proje geliştirirken bilgisayarınızdan eksik olmayan tek şey derleyiciniz ve MSDN dokümanları olsun. Hemen her bölümde MSDN kütüphanesini kullanmanız yönünde tavsiyem oldu. Bunu her defasında hatırlatmamın sebebi, bu kitapta .NET sınıf kütüphanesinin tamamının anlatılamayacağıdır. .NET sınıf kütüphanesi ile ilgili şu ana kadar yazılmış en iyi kaynak MSDN dokümanlarıdır. Bundan emin olabilirsiniz.

17

LINQ Mimarisi ve Yeni C# Özellikleri

- C# 3.0 Yenilikleri
- LINQ (Language Integrated Query: Dile Entegre Edilmiş Sorgu Yapısı)
- LINQ to Object Mimarisi
- var Bildirimli Değişkenler
- İsimsiz Tipler (Anonymous Types)
- Nesne ve Koleksiyon İlklendirme (Object and Collection Initializers)
- Otomatik Özellik Bildirimi (Automatic Property)
- Genişletme Metotları (Extension Methods)
- Lambda İfadeleri (Lambda Expressions)
- Kısmi Metotlar (Partial Metotlar)
- Gelişmiş LINQ Sorgu İfadeleri
- LINQ to SQL ile Veritabanı Sorgulama

.NET platformu sürekli gelişen ve değişen bir platform olduğu için hemen hemen her iki üç yılda bir .NET platformunun yeni versiyonu çıkmaktadır. .NET platformunun yeni versiyonu ile birlikte C# programlama dilinde de iyileştirmeler ve değişiklikler yapılmaktadır. 2008 yılının ilk çeyreğinde duyrulan Visual Studio.NET 2008 platformu ile birlikte C# 3.0 da resmi olarak yayınlanmış oldu. Yeni bir Visual Studio.NET versiyonu ile birlikte aslında temel olarak 3 ayrı sistemde versiyon değişikliği olmaktadır. Buna göre 2008 yılı itibariyle .NET platformunun yeni ve güncel versiyon bilgileri aşağıdaki gibidir.

Birim	Açıklama	Son Versiyon
Visual Studio	Microsoft'un geliştirmiş olduğu entegre geliştirme aracı. .NET platformu ile doğrudan ilgili değildir. .NET platoformunda yazılım geliştirmek için başka araçlarda kullanılabilir.	9.0 (2008)
.NET Framework	.NET platformu ile birlikte dağıtılan kütüphanelerin ve çalışma zamanı altyapılarının tamamına verilen isim.	3.5
C#	.NET platformunda geliştirme yapılabilecek ve bu kitabın da konusu olan programlama dillerinden sadece bir tanesi.	3.0

Bu kitabında konusu olan C# programlama dilinin son versiyonun 3.0 olduğu görülmektedir. C# programlama dilindeki değişikliklerin **CLR** (*Common Language Runtime*) dediğimiz birimde değişiklik yapmak zorunda değildir. Nitekim C# 3.0 ile birlikte gelen dil yeniliklerinin tamamı .NET 2.0 ile birlikte piyasaya sürülen CLR üzerinde çalışabilmektedir. Tabi bunun için C# 3.0 ile yazılmış kodların C# 3.0 sözdiziminden anlayan bir derleyici ile derlendikten sonra CLR 2.0 üzerinde çalışması sağlanmalıdır. Visual Studio.NET 2008 aracı ile yazılım geliştiriliğinde zaten programın hangi çalışma zamanı kütüphanelerini kullanacağı seçilebilmektedir. Dolayısıyla şunu rahatlıkla belirtebiliriz: C# 3.0 ile gelen yenilikler CLR 2.0 üzerinde herhangi bir değişikliğe neden olmamıştır ve bundan dolayı C# 3.0 ile yazılan kodlar rahatlıkla CLR 2.0 (.net framework 2.0) üzerinde çalışabilmektedir. Bu durum her zaman bu şekilde olmamıştır. Söz gelimi C# 2.0 ile birlikte yeni gelen **Generics** mimarisi gibi üst seviye yenilikler CLR üzerinde köklü değişikliklere neden olmuştur. Bu bakımdan .NET 2.0 ile geliştirilen bir uygulamanın .NET 1.1 üzerinde çalışabilmesine doğal yöntemlerle olanak yoktur.

Kitabın bu bölümünde .NET CLR 2.0 versiyonunu kökten değiştirmeyen ancak kod yazımı açısından üretkenliğimizi artıracak C# 3.0 yenilikleri anlatılmaktadır.

C# 3.0 Yenilikleri

C#, genel amaçlı bir programlama dilidir. Yani C# kullanılarak oyun geliştirebilirsiniz, çok amaçlı bir ERP uygulaması geliştirebilirsiniz, gömülü sistem uygulamaları geliştirebilirsiniz, veritabanı uygulamaları geliştirebilirsiniz hatta isterseniz mobil uygulamaları da geliştirebilirsiniz. C# programlama dilinin bu özelliğinden dolayı herhangi bir **uygulama alanına** (*application domain*) özgü kavramları C# dilinin söz dizimi içerisinde bulmanız mümkün değildir. C# gibi C, C++, VB ve Java dilleri de genel amaçlı programlama dillerine örnek gösterilebilir. Ancak öteden beri kullanılan **Fox Pro**, **RPC**, **Progress** gibi yazılım geliştirme araçları da bulunmaktadır. Bu ortamlar genellikle bir alana özgü ortamlardır. Örneğin **Fox Pro** ve **Progress** platformları ile hızlı bir şekilde veri merkezli uygulamalar geliştirebilir ve bu alandaki üretkenliğinizi artırabilirsiniz. Ya da Prolog ile yapay zeka uygulamalarını çok hızlı bir şekilde geliştirip yapay zeka kavramlarını hızlı bir şekilde uygulamanıza entegre edebilirsiniz.

Konumuz olan C# diline dönecek olursak; her ne kadar C# programlama dili genel amaçlı bir programlama dili olsa da günümüzde C# dilinin kullanım alanlarına baktığımızda büyük bir oranını **veri merkezli iş uygulamaları** (*data centric business applications*) kaplamaktadır. Bu yüzden bu tür uygulamalarda yoğun bir şekilde çeşitli veri ortamlarına erişimler yapılmakta, üstelik bu veriler çeşitli ortamlarda olabilmektedir. Örneğin veri ilişkisel bir veritabanı ortamında, hafızadaki bir koleksiyonda ya da bir XML dosyasında olabilmektedir. Genel amaçlı bir programla dilinde veri merkezli bir uygulama geliştirildiğinde ister istemez bir takım zorluklarla karşılaşılmaktaydı. Çünkü C# dili ile bir veritabanı ortamı arasında birebir bir eşleşme yoktur. Bu hem çalışma prensipleri hem veri tipleri bağlamında düşünülebilir.

C# 3.0 ile birlikte gelen yeniliklerin vizyonunu kavrayabilmek için yaşadığım bir deneyimi paylaşmak istiyorum. 2005 yılı içerisinde Amerika Birleşik Devletlerinde Microsoft kampüsünde C# dilinin baş mimarı Anders Hejlsberg'in yapmış olduğu bir sunuma katılmıştım. Bu sunumda Hejlsberg, C# 3.0'da yapmayı planladıkları yeniliklerden bahsediyordu. Sunuma başlamadan önce dinleyicilere bir soru yöneltti. Soru mealen şuydu: Aranızda kaç kişi C# programala dilini kullanarak veri tabanı erişimi yapıyor? Katılımcılar şaşırmış bir ifadeyle bakıyordu. Çünkü soru hepimize anlamsız gelmişti. Salonda sunumu dinleyen hemen herkes veritabanı uygulaması geliştiriyordu çünkü. Kısa bir durgunluktan sonra herkes elini kaldırdı. Hejlsberg, hafifçe gülümsedi ve ekledi: İşte C# 3.0'a eklemeyi planladığımız yenilikleri bu yüzden düşünüyoruz.

Bu kısa anıdan çıkaracağımız şudur: C# 3.0 programlama diline eklenen özelliklerin temelinde yatan şey **nesne yönelimli programlama** ile **fonksiyonel programlamayı** birleştirmekti. Kısacası genel amaçlı bir programlama dili içerisine belirli bir amaç için eklentiler yapmak. Bu belirli amaç ise tahmin edebileceğiniz gibi veri erişim işlemleridir.

Bundan sonraki bölümlerde C# 3.0 ile birlikte gelen bütün özellikleri ayrıntılı bir şekilde ele alıp örneklerle destekleyeceğim. Bütün bu yeni özellikleri kafanızda daha iyi canlanabilmesi için unutmamanız gereken en temel nokta, bu özelliklerin tamamının aslında bir amaca yönelik olduğudur.

LINQ (Language Integrated Query: Dile Entegre Edilmiş Sorgu Yapısı)

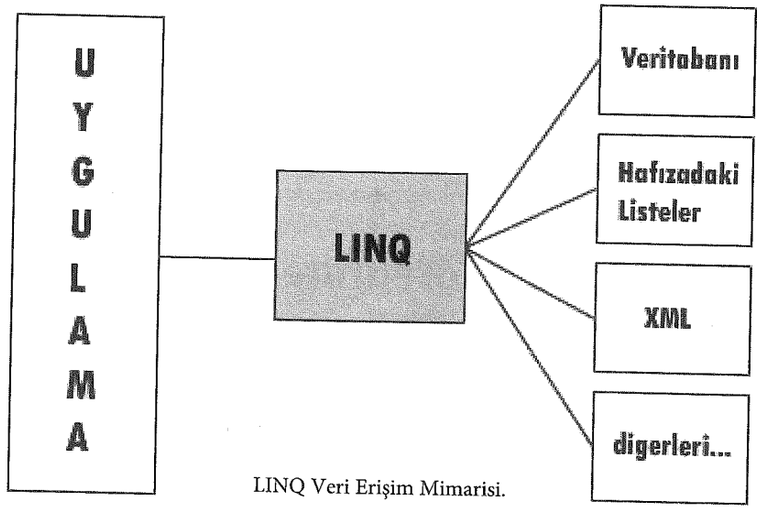
LINQ yani dile entegre edilmiş sorgu altyapısı C# 3.0'daki en önemli yeniliktir diyebiliriz. Hatta öyle ki diğer bütün yeni özelliklerde aslında LINQ merkezinde geliştirilmiştir. LINQ kısaca C# programlama dili ile herhangi bir üçüncü bileşene ihtiyaç duymadan dilin kendi söz dizimi ile veriye erişimi ve veri üzerindeki işlemleri kolaylaştırmaktır. Bahsedilen veri çok çeşitli şekillerde olabilir. Veri veritabanındaki ilişkisel veri, hafızadaki bir liste ya da bir XML olabilir. Hatta istenirse yeni veri kaynakları da tanımlanıp LINQ altyapısına entegre edilebilir. Bununla ilgili de çeşitli örneklerin olduğunu belirtmekte fayda var. İlerleyen aşamlarda örnekleri belirteceğim. LINQ, veri erişim mekanizmasını verinin tipinden bağımsız olarak standart hale getiren söz

dizimi topluluğuna verilen genel bir isimdir. LINQ, barındırdığı çeşitli soyutlama mekanizmaları sayesinde farklı veri kaynaklarına da erişimi kontrol edebilmektedir. .NET Framework 3.5 ile birlikte bu veri kaynaklarının 3 tanesi varsayılan olarak desteklenmektedir. Bunlar aşağıdaki listelenmiştir.

.NET 3.5 içerisinde bulunan **LINQ Veri Sağlayıcıları**:

- **Linq To Object:** Hafızada bulunan List<T> gibi koleksiyon yapılarının sorgulanması ve işlenmesi amacı ile geliştirilmiştir. Bu amaçla IEnumrable<T> arayüzüne birçok extension metot (extension metotları ilerleyen bölümlerde ele alacağız) eklenmiştir.
- **Linq To Sql:** Sql Server gibi ilişkisel veri formatlarında veriyi tutan ortamlardaki verilere erişmek ve o veriler üzerinde işlem yapmak için kullanabileceğimiz bir LINQ sağlayıcısıdır. Kısacası bu LINQ sağlayacısı ilişkisel veriyi nesnel ortamda temsil edebilmemizi sağlıyor. Bu yaklaşıma literatürde **ORM** (*Object Relation Mapping*) denilmektedir.
- **Linq To Xml:** XML veri kaynaklarına herhangi bir **parser** (*ayıklayıcı*) kullanmadan hızlı bir şekilde erişmek için kullanılmaktadır. Temel amacı XML tipli verilerle çalışırken üretkenliği artırmaktır.

Yukarıda kısaca açıklanan **3 LINQ sağlayıcısını** (*provider*) detaylı bir şekilde inceleyeceğiz.



LINQ Veri Erişim Mimarisi.

Bunların dışında üçüncü partilerin geliştirmiş olduğu birçok LINQ sağlayıcısı bulunmaktadır. Bunlara örnek olarak **Linq To MySql, Linq To Amazon, Ling To NHibernate, Linq to Ldap, Linq to Sharepoint, Linq to Streams** verilebilir. Bütün Linq

veri sağlayıcıları arka planda **Expressin Tree** (*İfade Ağaçları*) denilen yapı kullanarak özelleştirilmişlerdir. **Expression Tree** kavramı hakkında ilerleyen sayfalarda bilgi verilecektir. Üçüncü partilerin geliştirmiş olduğu LINQ sağlayıcıların tamamını merak ediyorsanız internetten yapacağınız küçük bir araştırma ile ulaşabilirsiniz. (*Anahtar Sözcük Önerisi: LINQ Providers*)

LINQ, veri erişim mekanizmasını verinin tipinden bağımsız olarak standart hale getiren söz dizimi topluluğuna verilen genel bir isimdir.

Neden LINQ

Bu kısa girişten sonra LINQ yapısının neden C# diline eklendiğini anlayabilmek için bütün seminerlerimde de kullandığım bir örneği inceleyelim. Örnek aynı zamanda Anders Hejlsberg'inde sunumunda kullandığı basit bir veri tabanı işlemi yapan bir kod parçasıdır.

Aşağıda görülen basit bir veritabanı işlemi yapan kod bloğundaki sorunlardan yola çıkarak LINQ'yu tanımladığımızda LINQ mimarisinin faydasını daha net anlamış olacaksınız.

```
static void Main(string[] args)
{
  SqlConnection con = new SqlConnection("Data Source=
.;initial catalog=Northwind;user id=sa;pwd=1");

  con.Open();

  SqlCommand cmd = new SqlCommand(
   @"SELECT Ad, Soyad,Tarih
     FROM Musteriler
     WHERE Sehir = @pSehir",con);

  cmd.Parameters.AddWithValue("@pSehir", "Ankara");

  SqlDataReader dr = cmd.ExecuteReader();

  while (dr.Read()) {
    string name = dr.GetString(0);
    string phone = dr.GetString(1);
    DateTime date = dr.GetDateTime(2);
  }

  dr.Close();
}
```

Örneğin sizin bilgisayarınızda çalışabilmesi için *Connection String* değerini değiştirmeniz ve *Musteriler* isimli tabloyu yaratmanız gerekmektedir.

Yukarıdaki örnekte Sql Server üzerinde bulunan *Musteriler* isimli tablodan *Ad*, *Soyad* ve *Tarih* alanları belirtilen kritere göre uygulama ortamına alınmaya çalışılıyor. Klasik veritabanı verisine erişim yöntemindeki temel sorunları irdeleyecek olursak;

- Kod içerisinde gömülü sorgular bulunmaktadır. Bu yüzden sorgulardaki hatalar ancak ve ancak çalışma zamanında farkedilebilmektedir. Derleme zamanında sorgu kontrolü yapılamamakta. Kod ortamından sorgu string olarak belirtildiği için sorguların yazımında da üretkenlik düşmektedir.
- Sorgular içerisinde **gevşek bağlı** (*loosely coupled*) argümanlar bulunmaktadır. Örneğimizdeki *pSehir* isimli parametre uygulama ortamından veritabanına gönderilmektedir. Ancak bu veritabanı tipinin doğruluk testi derleme zamanında değil **çalışma zamanında** (*run time*) yapılmaktadır. Bu yüzden çalışma zamanında veri tipi uyumsuzlukları ile sıkça karşılaşırız. Ya da en kötü ihtimalle bunun olmaması için programcı olarak ekstra efor sarfederiz.
- *string name = dr.GetString(0);* satırında görüleceği üzere veritabanından gelen sonuçların nasıl alınacağı ya bir index ile ya da string şeklinde kolon ismi ile alınmaktadır. Yani yine index ve kolon isimlerine bağlı bir ilişki var. Bu durumda da bu kodun çalışma zamanında kesin olarak çalışacağının garantisi yoktur. Tip uyuşmazlığı olabilecektir.
- *DateTime date = dr.GetDateTime(2);* Varsayalım ki sonuca ait kolon ismi ya da index doğru. Bu durumda gelecek verinin gerçekten .NET tipine denk gelip gelmeyeceği soru işaretidir. Derleme zamanında tip kontrolü yapılamadığı için örnekte görülen DateTime dönüştürme işlemi çalışma zamanında hataya neden olabilecektir.

Yukarıdaki tespitler, hem program yazımını zorlaştırmak hem de programcının üretkenliğini düşürmektedir. Bu durumun sebebi elbette ki veritabanı ortamı ile programlama ortamının birbirinden farklı olmasıdır. Her iki ortamın da kendine has özellikleri bulunabilir. Örneğin veritabanında int tipinden kolonlar null değer içerebilirken C# dilinde **int** değişkenler kesinlikle **null** değer içeremezler. Platform farklılıklarından kaynaklanan bu farklılıkları ortadan kaldırmak ve programcıyı veri bağımsız hale getirmek amacıyla C# 3.0'dan itibaren LINQ yapısı geliştirilmiştir.

ORM (Object Relation Mapping) Yaklaşımı

Buna göre bir **LINQ sağlayıcısı** olan **LINQ to SQL** kullanılarak yukarıdaki kod tamamen tip güvenliğini ön planda tutacak şekilde aşağıdaki gibi yazılmaktadır.


```
public class Musteri { ... }
public class Northwind: DataContext
{
   public Table<Musteri> Musteriler;
   ...
}

//İlişkisel Nesnelerin Sınıf Olarak Modeli

Northwind db = new Northwind("...connction string");

var contacts = from c in db.Musteriler
          where c.Ad == "Sefer"
          select new { c.Ad, c.Soyad };

//LINQ Sözdizimi ile Sorgulama
```

Yukarıdaki kod bloğu çalıştırılabilir kod bloğu değildir. LINQ yapısının giriş düzeyinde kafanızda canlandırabilmeniz için verdiğim örnek bir prototiptir. LINQ ve LINQ to SQL'in detaylarına zaten değineceğiz.

Yukarıdaki kod bloğunu incelediğinizde bir an için C# diline SQL dilinin gömüldüğünü hissedebilirsiniz. Hemen panik yapmaya gerek yok. Yukarıda gördüğünüz kod parçası içinde var, select, from gibi ifadelerin hepsinin mantıklı birer açıklaması mevcut. Bölümün bundan sonraki kısımlarında ise zaten bunları inceleyeceğiz.

Klasik veri erişimi örneğindeki sorunlardan bahsetmiştik; şimdi ise LINQ erişimi sayesindeki kazanımlarımızı irdeleyelim;

- İlk olarak kod ortamında Musteri isimli bir sınıf yaratılarak ilişkisel veri ortamındaki yapıların nesnel karşılıkları oluşturulmuş. Bu **O/R Mapping** (*Object Relation Mapping*) sayesinde programcı nesnel düşünmekten feragat etmeden veri erişim işlemlerini gerçekleştirebilecek. Tip güvenliğinden kaynaklanan avantajlar ise işin çabası.
- Yine aynı şekilde *Northwind* isimli sınıf tanımlanarak veritabanı kavramı nesnel olarak tasarlanmış durumda. Ayrıca bu veritabanı içerisinde *Musteriler* isimli özellik ile *Musteri* nesnesi bir koleksiyon şeklinde dışarıya sunulmuş. Böylece programcı tip-güvenli bir şekilde veritabanı tablolarına ve kayıtlarına erişebilecektir.
- Veritabanına bağlanılacağı anda *Northwind* isimli nesnenin açıkça yaratılması sağlanmış. Buraya isteseniz bile Northhhwind şeklinde hatalı yazamazsınız. Çünkü derleme zamanında hata alırsınız.
- İçinde *where, select, from* deyimleri geçen satırda ise LINQ işe koyuluyor. Herhangi sql sorgusu yazmadan veritabanı erişimi gerçekleşiyor. Söz dizimi olarak hatalı bir sorgu yazmanıza imkan yok.

- *select new { c.Ad, c.Soyad };* satırında ise veritabanından alınan veriler yine tip güvenliği üst düzeyde olacak şekilde bir nesne olarak döndürülüyor. Böylece gelen verilerin tipleri, isimleri derleme zamanında açıkça belli olmaktadır.

Görüldüğü üzere LINQ yapısı sayesinde üst düzey tip güvenliği sağlayarak ilişkisel verilerin nesne tabanlı olarak temsil edilmesi sağlanmış. Nesneler üzerinde bulunan ilişkisel veriyi hızlı bir şekilde sorgulamak için ise dile entegre edilmiş bir sorgulama aracı geliştirilmiştir. İşte bu yapıya LINQ diyoruz. LINQ to Sql ise LINQ altyapsınının SQL veritabanı üzerine uyarlanmış versiyonudur.

Artık LINQ'nun size daha anlamlı geldiğini düşünüyorum. LINQ'nun tanımını yaptığımıza göre artık işin derinliklerine inip C# söz dizimlerinin ne anlama geldiklerini öğrenelim ve nasıl oluyor da bu yapılar .NET CLR 2.0 üzerinde çalışır hale getirilebiliyor irdeleyelim.

LINQ to Object Mimarisi

Bir önceki bölümde **LINQ** yapısının neden getirildiğini daha iyi kavrayabilmek için örneği **LINQ to SQL** üzerinden verdim. Ancak LINQ mimarisini daha iyi anlayabilmek için LINQ to SQL'i inceleyeceğimiz bölüme kadar bütün örnekleri LINQ to Object veri sağlayıcısı üzerinden vereceğim. Daha önce bahsedildiği gibi **LINQ to Object** sağlayıcısı hafızada bulunan ve IE**numerable<T>** arayüzünü uygulayan bütün koleksiyonlarda uygulayabileceğimiz ve .NET 3.5 ile birlikte gelen bir yapıdır.

Bu alt bölümdeki amacımız LINQ sorgu ifadelerine temel düzeyde giriş yapıp, C# 3.0 ile birlikte gelen yeni özelliklerin aslında LINQ yapısını desteklemek amacıyla yapıldığını göstermektir.

Örneğin aşağıdaki kod parçasını ele alalım. Bu örnekte, içerisinde *Musteri* türünden nesneler bulunan bir liste içinde *MusteriNo*'su 3'ten büyük olanlar ve Ad alanının uzunluğu en az 4 karakter olan müşterilerin tamamı bir liste olarak elde edilmiş.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public class Musteri
{
  private int no;
  public int No
  {
    get { return no; }
    set { no = value; }
  }
```



```
  private string ad;
  public string Ad
  {
    get { return ad; }
    set { ad = value; }
  }
  public Musteri(int no,string ad)
  {
    this.No = no;
    this.Ad = ad;
  }
}

class Program
{
  static void Main()
  {
    List<Musteri> musteriler = new List<Musteri>();
    musteriler.Add(new Musteri(3,"Sefer"));
    musteriler.Add(new Musteri(6,"Can"));
    musteriler.Add(new Musteri(7,"Ahmet"));
    musteriler.Add(new Musteri(1,"Veli"));
    musteriler.Add(new Musteri(8,"Oğuz"));


    var sonuclar = from m in musteriler
            where m.Ad.Length > 4 && m.No>3
            select new { m.Ad, m.No };

    foreach (var obj in sonuclar)
    {
      Console.WriteLine("Ad :" +obj.Ad + " No:" + obj.No);
    }
  }
}
```

Bu örneğin çıktısı aşağıdaki gibi olacaktır:

```
Ad : Ahmet No :7
```

Yukarıdaki kod bloğunu C# 2.0'da bir **for döngüsü** ile kolayca yapabilirdik. Ancak görüldüğü gibi LINQ ile bu işlem çok daha basit ve okunaklı olmaktadır. Ayrıca verinin hangi formatta olduğu da LINQ içinde bizi ilgilendirmiyor. İleriki bir zamanda bu listenin içerisi müşteri veritabanından gelecek veriler ile dolu olsa da kullanacağımız LINQ yapısı yine buna benzer olacaktır.

Bu örnekte odaklanmanız gereken nokta *from, select, where* gibi deyimleri içeren LINQ yapısıdır. Dikkat edilirse *musteriler* isimli koleksiyon üzerinde tıpkı SQL dilinde olduğu gibi çeşitli sorgulamalar yapılmış. (from ve select'in yer değiştirmiş olduğuna ayrıca dikkat edin.)

LINQ yapısının daha derinliklerine inebilmemiz için C# 3.0 ile birlikte dile eklenmiş yenilikleri bilmemiz gerektiği açıkça belli oldu. Yukarıdaki örnekte bulunan *var, select, from, where* gibi ifadelerin aslında ne olduklarını irdelemenin zamanı geldi. C# 3.0 dilindeki yenilikleri detaylı bir şekilde ele aldıktan sonra LINQ sorgu ifadelerine yeniden dönüş yapacağız.

```
var sonuclar = from m in musteriler
    where m.Ad.Length > 4 && m.No>3
    select new { m.Ad, m.No };
```

C# 3.0 dilinin yeniliklerine giriş yapabilmek için yukarıdaki kod bloğunun derleyici tarafından aslında nasıl bir kod çağrımına dönüştüğünü inceleyelim. Ki böylece bu kod bloğunun çalışma zamanında CLR 2.0 üzerinde çalışabileceğini ispatlayalım.

LINQ yapısında bulunan *from, where, select* gibi ifadeler (ileride göreceğimiz join, group by da dahil) aslında C# 3.0 derleyicisi tarafından aşağıdaki metot çağrım ifadelerine dönüştürülmektedir.

```
musteriler.Where(m => m.Ad.Length>4 && m.No>3).Select(m => new{
m.Ad,m.No });
```

Sonuç olarak **Where** ve **Select** ifadelerinin aslında birer metot olduğunu görüyoruz. Bu ifade de bile henüz bilmediğimiz birçok şey var. Mesela **Where** ve **Select** metotlarının parametrelerinin aslında birer **delegate** olduğunu bilmek gerekir. Kısacası yukarıdaki ifade bir kademe daha eski CLR 2.0 yapısına dönüştürülürse aşağıdaki çağrım ifadesi çıkacaktır.

```
sonuclar3 = musteriler.Where(delegate(Musteri m)
        {
                        return m.Ad.Length > 4 && m.No > 3;
                }
            ).Select(delegate(Musteri m)
            {
                return new { m.Ad, m.No };
    }
        );
```

Yukarıdaki çağrım ifadesinde *Where* ve *Select* metotlarının aslında birer isimsiz **delegate** olduğunu anlıyoruz. Hatırlarsanız C# 2.0 ile birilikte gelen isimsiz metotlar sayesinde yukarıdaki gibi **delegate** nesnesi bekleyen parametrelere fonksiyon çağrım ifadesi yazabiliyorduk.

Peki, nasıl oluyorda **IEnumrable<T>** içerisinde bizim yazmadığımız bir Select ve Where metodu çağrılabiliyor. Bunun cevabı ise **genişletme metotları** (*Extension Method*). C# 3.0 diline gelen yeni bir özellikten dolayı mevcut tipleri türetme yapmadan metot ekleyebiliyoruz. Bu sayede yukarıdaki gibi bir çağrım yapılabilmektedir. Zaten yukarıdaki kod satırının çalışabilmesi için:

```
using System.Linq;
```

bildiriminin kodda olması gerekir.

Buna göre, yukarıdaki kod bloğu da aslında çalışma zamanında aşağıdaki çağrım ifadesine dönüştürülecektir.

```
IEnumerable<Musteri> sonuclar =
System.Linq.Enumerable.Where<Musteri>(musteriler,
                delegate(Musteri m)
                {
                        return m.Ad.Length > 4 && m.No > 3;
        }
        ).Select<Musteri,Musteri>(delegate(Musteri m)
         {
                return new Musteri( m.No, m.Ad );
        }
    );
```

Oldukça karışık bir C# 2.0 kodu olduğunu görüyoruz. Görüldüğü gibi yukarıdaki bloğu CLR 2.0 uyumlu çalıştırılabilir bir ifadedir. Sonuç olarak Where metodunun aslında **System.Linq.Enumerable** isimli sınıfın bir üye elemanı olduğunu görüyoruz. C# 3.0'daki extension metotları sayesinde ise bu çağrım ifadesini daha kolay bir şekilde tanımlayabiliyoruz.

Yukarıdaki karışık metot çağrım ifadesini yorumlayıp anlayabilmek için koleksiyonlar, generic mimari ve isimsiz metot konularına hakim olmanız gerekir. Bu konulara hakim değilseniz kitabın ilgili bölümlerini tekrar gözden geçiriniz.

System.Linq isim alanında **Select** ve **Where** genişletme metotlarının bildirimleri bulunmaktadır. Yani **System.Linq** isim alanının erişilebildiği her yerde **IEnumerable<T>** arayüzünden türeyen sınıflarda **Select** ve **Where** metotlar kullanılabilir haldedir. Bunun gibi daha birçok genişletme metodu tanımlanmıştır.

Buraya kadar amacımız genişletme metodunu ya da isimsiz metotların kısa yazılımını anlatmak değil, aslında LINQ yapısı içerisinde bulunan yapıların uzun yoldan da olsa bir şekilde C# 2.0 ile gelen yapılarla ifade edilebileceğini göstermektir.

Bu aşamadan sonra yukarıda bahsedilen LINQ yapısını daha kolay bir şekilde yazabilmemizi sağlayacak C# 3.0 dil yeniliklerine değinebiliriz.

Aşağıda bahsedilecek olan yenilikleri incelerken düşüncelerinizi LINQ etrafında yoğunlaştırırsanız kavramanız daha kolay olacaktır.

var Bildirimli Değişkenler

C# 3.0 ile birlikte bir metot içerisinde tanımlanacak değişkenler tipleri bildirilmeden tanımlanabilmektedir. Buna göre aşağıdaki tanımlama biçimleri tamamen geçerlidir.

```
var i = 4;
var str = "Ali";
var d = 3.4;
var m =  new Musteri();  //Musteri sınıfı varsa
```

Dile eklenen var anahtar sözcüğü sayesinde **değişkene atanan değerin tipinden çıkarsama** (*type inference*) yapılarak derleme aşamasında **IL** içerisine gerçek tip bildirimi eklenir. Yani yukarıdaki tanımlama biçimi ile aşağıdaki tanımlama biçimi arasında çalışma zamanı açısından hiçbir fark yoktur.

```
int i = 4;
string str = "Ali";
double d = 3.4;
Musteri m =  new Musteri();  //Musteri sınıfı varsa
```

Bu noktada, bir değişkeni **object** olarak tanımlamak ile var bildirimi ile tanımlamak arasında hiçbir benzerliğin olmadığını rahatlıkla söyleyebiliriz. Çünkü **object** ile tanımlanan değişkenlerin tür kontrolleri ve ayrışımı çalışma zamanında dinamik olarak yapılır. var ile bildirimde ise bu işlem derleme aşamasında olmaktadır.

var ile bildirilen bir değişkene, aşağıdaki gibi başka bir tipten değişken atanamaz.

```
var no = 4;
no = "Ali"; //bu aşamada derleme hatası alınacaktır.
```

var anahtar bildiriminin dile eklenmesinin temel nedeni yukarıdaki gibi basit veri tiplerini tanımlarken değil de özellikle şablon tipler içeren karmaşık tiplerin tanımlanmasıdır. Özellikle LINQ içinde **Select** ifadesinden geriye döndürülen tipin ne olduğu belli olmadığı durumlarda (*anonymous type:* isimsiz tip) var bildiriminin kullanılması kaçınılmazdır. Zaten bu özellik de daha çok bu amaç için eklenmiştir. Ancak genel amaçlı bir programlama diline eklenen bu yapıların da mutlaka genel amaçlı olma zorunluluğu olduğu için yukarıdaki gibi basit veri tipleri ile de kullanılabilir.

Bu duruma bir örnek vermek gerekirse;


```
var sonuclar = from m in musteriler
               where m.Ad.Length > 4 && m.No>3
               select new { m.Ad, m.Ad.Length};
```

ya da

```
var sonuclar = musteriler.Where(delegate(Musteri m)
        {
          return m.Ad.Length > 4 && m.No > 3;
        }
      ).Select(delegate(Musteri m)
        {
          return new { m.Ad, m.Ad.Length };
              }
      );
```

Yukarıdaki her iki ifade de *Select* metodundan geriye döndürülen nesnenin adı belli değildir. *Select* metodunun geri dönüş değeri içinde iki adet özellik olan (Ad ve Ad alanının uzunluğunu tutan) bir nesnedir. Ancak bu nesnenin tipi belli değildir. Çünkü **Select** ifadesi içinde otomatik üretilmiş isimsiz bir sınıf (isimsiz sınıflar birazdan anlatılacaktır) bulunmaktadır. Bu durumda geri dönen bu isimsiz tipli nesnelerden oluşan listeyi var ile bildirilen bir değişkene atamaktan başka çaremiz yok. Dolayısıyla **var** bildiriminin dile eklenmesinin anlamı bu kadar açık iken başka kullanım yerlerinin aranması anlamsız ve gereksizdir.

var anahtar sözcüğünün diğer bir kullanım yeri ise **foreach** döngüsü ile koleksiyon içinde dolaşırken koleksiyon elemanlarının tipinin bilinmeden kullanılabilmesidir.

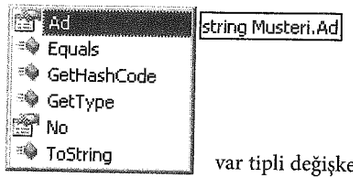
Örneğin aşağıdaki kullanım tamamen geçerlidir;

```
List<Musteri> musteriler = new List<Musteri>(); //Musteri sınıfı var ise
foreach (var m in musteriler)
{
   Console.WriteLine(m.Ad);
}
```

C# derleyicisi generic **List** dizisinin içindeki elemanların tipinden çıkarsama yaparak **foreach** iterasyon elemanının (bu örnekte *m* değişkeni) tipini *Musteri* olarak belirler. Eğer Visual Studio.NET 2008 ile geliştirme yapıyorsanız Intellisense desteğinin de **var** ile bildirilen değişkenlerde çalıştığını aşağıdaki gibi görebilirsiniz.

var anahtar kelimesi ile bildirilen değişkenlerin kısıtları;

```
static void Main()
{
    List<Musteri> musteriler = new List<Musteri>();
    foreach (var m in musteriler)
    {
        m.
    }
```



var tipli değişkenlerin Intellisense desteği

- **var** ile bildirilen değişkenler sadece lokal değişken olabilir. Yani bir sınıfın üye elemanı olamaz.
- **var** ile bildirilen değişkene ilk değer atama işlemi zorunludur. Aksi durumda tip çıkarsaması yapamayacağından derleme hatası alınır.
- Bir metodun herhangi bir parametresi ya da geri dönüş değeri **var** tipinden olamaz.
- **var** ile bildirilen bir değişkene sonradan farklı bir türden değişken atanamaz.

Dizi Tiplerinin Çıkarsama Yolu ile Belirlenmesi

C# 3.0 ile gelen yeniliklerden bir tanesi de dizi tiplerinin, dizi içerisinde bulunan elemanlarının tiplerinden çıkarılabilmesidir. Buna göre bir diziyi aşağıdaki gibi tanımlayabilirsiniz.

var dizi1 = new[] { 3, 4, 5, 3 };
var dizi2 = new[] { 'C', 'A', '5'};
var dizi3 = new[] { 3.4f, 4.4f, 5.2f, 3.0f };

Yukarıdaki örnek tanımlama biçimlerinden de görüleceği üzere dizi elemanlarının **aynı tipten** ya da **bilinçsiz** (*implicit*) bir şekilde birbirine dönüştürülebilen tiplerden olması gerekiyor. Buna göre aşağıdaki tanımlama biçimi derleme aşamasında hata verecektir.

var dizi1 = new[] { "3", 4, 5, 3 }; //Derleme hatası alınır.

Fakat aşağıdaki tanımlama biçimi ise geçerlidir.

public class Taban
{
}

public class Tureyen:Taban
{
}

Sınıflarının varsayımı altında

var dizi4 = new[] { new Taban(),new Tureyen(),new Tureyen()};

Buna göre derleme aşamasından sonra *dizi4* dizisinin gerçek tipi tek ihtimal olan *Taban []* şeklinde belirlenecektir.

İsimsiz Tipler (Anonymous Types)

C# 2.0 ile birlikte gelen isimsiz metotlardan sonra C# 3.0 ile birlikte LINQ yapısını kurabilmek için isimsiz tipler geliştirildi. Program yazarken sık sık 2-3 değişkeni bir arada tutmak için sınıf yaratırız. Sonra bu sınıf türünden nesneler yaratıp koleksiyonlar oluştururuz. Bu işlem esasında programcı için nesne tanımlamaktan ziyade değişkenlere bir grup ismi ile erişmek maksatlı yapılır. İsimsiz tipler sayesinde bu tür tanımlamaları daha kolay yapabileceğiz.

Öyle bir veri tipi düşünün ki, içinde birbiriyle ilişkisi özellikler olsun ve adı olmasın. Ama bir şekilde de erişebilelim. Tahmin edeceğiniz gibi isimsiz veri tiplerinin kullanımı için var bildirimlerine ihtiyaç duyacağız.

İsimsiz tip tanımlama sözdizimi aşağıdaki gibidir.

new { OzellikAdı1 = Degeri,OzellikAdı2=Degeri ..... };

Buna göre; 3 elemandan oluşan ve müşteri verilerini tutan bir veri yapısı aşağıdaki şekilde tanımlanabilecektir.

var nesne = new { Ad = "Ali", No = 5, Maas = 3500.5 }; //sınıf bildirimi, değişken bildirimi ve değişken tanımlama tek satırda yapılıyor.

Görüldüğü üzere oluşturulan 3 özellikli *Musteri* veri yapısının derleme zamanındaki tip adı belli değildir. Bu yüzden nesne tanımlarken var bildirimini kullanmak zorundayız. (Derleyici derleme aşamasında otomatik bir isim belirler).

Yukarıdaki ifadede her ne kadar değişken adı belli değil ise de çalışma zamanında aslında bir sınıfın yaratıldığını ve sınıf tipinin var ile yer değiştirildiğini görürüz. Bunu aşağıdaki **Visual Studio** aracının **debug** ekranından da rahatlıkla görebiliyoruz.

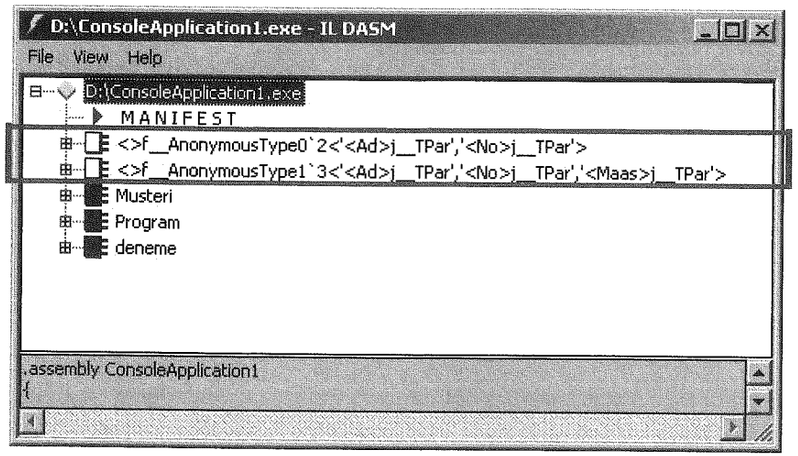
Görüldüğü üzere yaratılan *nesne*'nin tipinin aslında çalışma zamanında bir isme sahip olmuştur. Bu isimde derleyici tarafından "<>f__AnonymousType1`3" şeklinde belirlenmiştir.

```
88          }
89
90          var nesne = new { Ad = "Ali", No = 5, Maas = 3500.5 };
91          string typename =  nesne.GetType().Name;
92                      typename  "<>f__AnonymousType1`3"
93      }
94  }
```

İsimsiz tiplerin çalışma anındaki görüntüsü.

Aynı zamanda yine .NET Framework ile birlikte sisteme kurulan **ILDASM** (*Intermaediate Language Disassmebler*) aracı ile **IL** kodları içine baktığımızda yine aynı şekilde isimsiz bir tipin **IL** içine enjekte edildiğini görürüz.

Bu iki görüntüden isimsiz tiplerin CLR üzerinde bir etki yaratmadığını anlıyoruz. C# derleyicisi, isimsiz tipin new operatörü ile yaratıldığı andaki görüntüsünden çıkarsama yaparak isimsiz tipe ait **set** ve **get** blokları olan özellikleri yaratır.



IL ve isimsiz tipler

Dolayısıyla aşağıdaki kullanım tamamen geçerli bir kullanımdır.

var nesne = new { Ad = "Ali", No = 5, Maas = 3500.5 };
string ad = nesne.Ad;
int no = nesne.No;

Aynı zamanda Visual Studio.NET içindeki Intellisense özelliğinin de isimsiz tipleri ayırt edebildiğini belirtmekte fayda var.

İsimsiz tiplerde diğer .NET sınıflarında olduğu gibi otomatik olarak Object sınıfından türetilmiş bir sınıf olarak tanımlanır.

İsimsiz tiplerin en önemli kullanım yeri tahmin edebileceğiniz gibi **LINQ** sorgularıdır. Özellikle **Select** isimli metodun geriye döndüreceği nesne, koleksiyon elemanın tipinden farklı ise işe yaramaktadır. Nitekim **LINQ to SQL** sağlayıcısında *Musteri* tablosundan sadece *Ad* ve *Soyad* alanlarını çekmek için mecburen **isimsiz tip** yaratmak zorundayız. Aynı şekilde LINQ to Object yapısı içinde de geri dönülecek sonuç listesindeki elemanların prototipinden farklı ise isimsiz tipler kullanılabilir.

Select ifadesinden dönen sonucun bir isimsiz tip koleksiyonu ile aşağıdaki gibi yazılması çok sık karşılaşacağımız bir durumdur.

var sonuclar = from m in musteriler //müşteriler dizisi varsayılıyor.
select new { m.Ad, m.No }; // isim tip yaratılıyor.

tipler kullanılarak nesne yaratıldığında var bildirimini kullanmamız gerekir. Var bildirimi yerine Object bildirimini de kullanabiliriz ancak bu durumda tip güvenliğini kaybetmiş oluruz.

İsimsiz tiplere ait nesneleri var bildirimi ile almadığınız sürece C# içerisinde metot parametre geçişleri dâhil her yerde kullanmanız mümkündür.

İsimsiz tiplerle ilgili diğer önemli durum ise farklı satırlarda yaratılmış aynı kalıba uygun nesnelerin birbirlerine atanıp atanamayacağı. C# kurallarına göre bu mümkündür. Aynı program içinde aynı özellik desenine sahip (isim ve tip bakımından) isimsiz tip nesneleri birbirlerine atanabilir.

Buna göre aşağıdaki kod bloğu tamamen geçerlidir;

var nesne1 = new { Ad = "Ali", No = 5, Maas = 3500.5 };
var nesne2 = new { Ad = "Ali", No = 5, Maas = 3500.5 };
nesne1 = nesne2;

Buna rağmen eğer isimsiz tiplerin desenlerinde aşağıdaki gibi en ufak bir değişiklik olduğunda C# derleyicisinin derleme zamanında yeni bir tip yarattığını görürüz. Buna göre aşağıdaki atama işlemi geçersiz olacaktır;

var nesne1 = new { Ad = "Ali", No = 5, Maas = 3500.5 };
var nesne2 = new { Ad = "Ali", No = 5};
nesne1 = nesne2; //Derleme hatası alınır.

Nesne ve Koleksiyon İlklendirme (Object and Collection Initializers)

C# 3.0 ile birlikte LINQ'da zorunlu olarak kullanılan **nesne ve koleksiyonları ilklendirme** (*oluşturma*) yapısı genel amaçlı olarak da kullanılabilir. C# 3.0 ile birlikte nesneleri ve koleksiyonları çok daha kolay bir şekilde ilklendirebiliyoruz.

Nesne İlklendirme

Sınıflara ait nesneleri yaratırken **yapıcı metotları** (*constructor*) kullanıyoruz. Birçok kere farklı amaçlarda kullanmak üzere farklı parametrelere sahip yapıcı metotları da tanımlarız. Eğer yazdığımız yapıcı metotların tek görevi özelliklerin ya da değişkenlerin atanması ise gereksiz yere yapıcı metot tanımlamak zorunda kalmıyoruz. Buna göre bir nesne **new** operatörü ile yaratılırken public olan değişken ve özelliklerin ilk değerleri süslü parantezler içinde virgül ile ayrılarak aşağıdaki şekilde verilebilir.

```
using System;

public class Personel
{
  public int No;
  public string Ad;

  public Personel()
  {
    Console.WriteLine("yapıcı metot");
  }
}

public class Program
{
  public static void Main()
  {
    var ali = new Personel { No = 12, Ad = "Sefer Algan"};
  }
}
```

Yukarıdaki,

```
Personel ali = new Personel { No = 12, Ad = "Sefer Algan"};
```

kod bloğu ile aşağıdaki kod bloğu çalışma zamanı açısından eşdeğerdedir. Yani her iki durumda da varsayılan **yapıcı metot** çalışmaktadır.

```
Personel ali= new Personel();
p.No = 12;
p.Ad = "Sefer Algan";
```


Her iki durumda da **yapıcı metot** ilk olarak devreye girer. Ardından **public** olan değişken ya da özellikler atanır.
Sınıf içinde başka bir sınıfa ait nesne olduğunda da yukarıdaki kurallar geçerlidir. Örneğin *Personel* sınıfının içinde başka bir sınıf olan *Maas* türünden bir değişken olduğunu düşünelim. Yani;

```
public class Personel
{
  public int No;
  public string Ad;
  public Maas maas;
}

public class Maas
{
  public int Tutar;
  public string Kur;
}
```

Buna göre aşağıdaki *Personel* nesne tanımlama satırı tamamen geçerlidir;

```
Personel nesne = new Personel { No = 12, Ad = "Sefer Algan",
maas = new Maas { Tutar = 200, Kur = "YTL" } };
```

Aynı şekilde aşağıdaki tanımlama biçimi de geçerlidir;

```
var maasNesnesi = new Maas { Tutar = 1000, Kur = "USD" };
var personelNesnesi = new Personel { No = 12, Ad = "Ali Aslan", maas = maasNesnesi};
```

Yukarıdaki örneklerde ilklendirme yapılırken sınıf içerisindeki değişkenler kullanılmıştır. Aynı şekilde özelliklere de (`property`) ilklendirme parantezi içerisinde değer atanabilir.

Öte yandan eğer *Personel* içerisindeki *Maas* nesnesi zaten *Personel* nesnesi yaratıldığında aşağıdaki gibi oluşturuluyorsa;

```
public class Personel
{
  public int No;
  public string Ad;
  public Maas maas = new Maas();
}
```

Bu durumda ilklendirme nesne yaratmaya ihtiyaç duyulmadan aşağıdaki gibi de yapılabilmektedir.

```
var personelNesne = new Personel { maas = { Tutar= 800, Kur= "USD" } };
```

Koleksiyon İlklendirme

Sınıf nesnelerinde olduğu gibi koleksiyon nesnelerinin elemanlarını da koleksiyonu yarattığımız anda belirleyebiliriz. Koleksiyon ilklendirmeye örnek bir tanımlama aşağıda verilmiştir;

```
List<int> liste = new List<int> { 3, 4, 2, 5 };
```

Bu tanımlama biçimi ile aşağıdaki deyimler eşdeğerdir.

```
List<int> liste = new List<int>();
liste.Add(3);
liste.Add(4);
liste.Add(2);
liste.Add(5);
```

Her iki durumda da koleksiyona ait yapıcı metot çağrılmaktadır. Yukarıdaki kod bloğundan da anlaşılacağı üzere koleksiyon ilklendirmede kullanılan elemanların tiplerinin uyumlu olması gerekir. Uyumsuz bir tip olması durumunda derleme zamanında hata alınır.

Koleksiyon ilklendirme sadece System.Collections.Generic.ICollection<T> ve System.Collections.ICollection arayüzlerinden türemiş olan koleksiyonlarda kullanılabilir.

Aynı şekilde *ArrayList* nesneleri de aşağıdaki gibi ilklendirme işlemine tabi tutulabilir.

```
ArrayList liste2 = new ArrayList { 3, "3", 2, 3 };
```

İçiçe geçmiş koleksiyon nesnelerinin ilklendirilmesi de mümkündür. Örneğin aşağıdaki gibi bir *Personel* nesnesi yaratıldığında;

```
public class Personel
{
  private string ad;
  public string Ad
  {
    get { return ad; }
    set { ad = value; }
  }

  private List<string> telefonlar = new List<string>();
  public List<string> Telefonlar
  {
    get{ return telefonlar;}
  }
}
```


Aşağıdaki gibi bir *Personel* nesne ilklendirmesi yapılabilmektedir.

```
List<Personel> personeller = new List<Personel>()
        {
                new Personel{
                        Ad = "Alper",
                Telefonlar = {"212yy","216xx"}
                },
                  new Personel{
                Ad= "İsmail",
                Telefonlar ={"1006","1003"}
        }
                };
```

Yukarıdaki ilklendirme ifadesi ile aşağıdaki çalıştırılabilir satırların eş değer olduğunu söyleyebiliriz.

```
List<Personel> personeller = new List<Personel>();
Personel p1 = new Personel();
p1.Ad = "Alper";
p1.Telefonlar.Add("212yy");
p1.Telefonlar.Add("216xx");
personeller.Add(p1);
Personel p2 = new Personel();
p2.Ad = "Alper";
p2.Telefonlar.Add("1006");
p2.Telefonlar.Add("1003");
personeller.Add(p2);
```

Aynen Listelerde olduğu gibi anahtar değer ikilisi taşıyan *Dictionary* de bildirilirken elemanları kolay bir şekilde eklenebilmektedir. Örnek bir *Dictionary* tanımlaması aşağıdaki gibi yapılabilir.

```
private Dictionary<string, string> Departmanlar = new Dictionary<string, string>()
{
   { "Account","Muhasebe" }, {"Sales","Satış" }
};
```

Dictionary index tabanlı bir koleksiyon olduğu için C# 6.0'dan itibaren ilklendirme için daha okunabilir bir yazım şekli gelmiştir. Buna göre C# 6.0'dan itibaren anahtar değer ikili tutan koleksiyonlar aşağıdaki şekilde tanımlanabilir.

```
private Dictionary<string, string> Departmanlar = new Dictionary<string, string>()
{
  ["Account"] = "Muhasebe",["Sales"] = "Satış"
};
```

Nesne ve koleksiyon ilklendirme ile ilgili gelen yenilikler de diğer yeniliklerde olduğu gibi LINQ yapısında sıkça kullanılmaktadır.

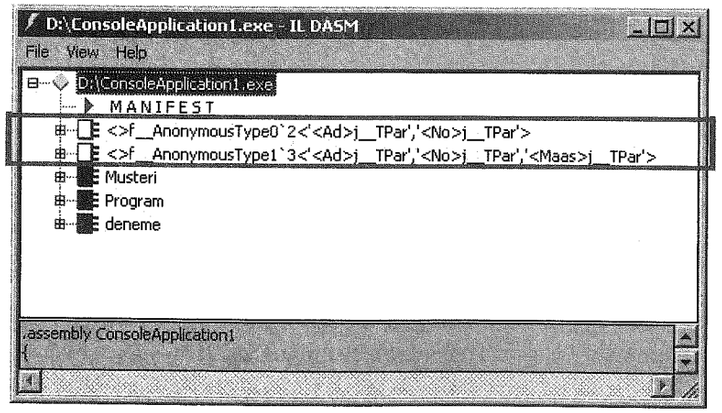
Otomatik Özellik Bildirimi (Automatic Property)

C# 3.0 ile birlikte özellik tanımlama biçimi çok daha kolay hale getirildi. Özellikle yalnızca erişimin kontrol altına alınmak istendiği senaryolarda **get** ve **set** bloklarını tek tek yazmak ve özelliğe ait bir değişken tanımlamaya gerek yok.

Otomatik özellikler daha çok bir sınıfın özellik havuzu şeklinde kullanılması durumunda faydalıdır. Bu durumda C# derleyicisi derleme aşamasında otomatik olarak ilgili özelliğe ait bir değişken yaratıp, set ve get metotlarını IL koduna yerleştirir. Aşağıda *Personel* sınıfının 3 özellik ile birlikte bildirimine örnek verilmiştir.

```
public class Personel
{
  public string Ad { get; set; }
  public int No { get; set; }
  public DateTime Tarih { get; set; }
}
```

Yukarıdaki sınıf bildirimini içeren **assembly ILDASM** aracı ile incelendiğinde aşağıdaki sonuç çıkacaktır.



Görüldüğü gibi 3 özelliğe ait değişkenler "<Ad>k_BackingField" formatında isimlendirilerek yaratılmış. Aynı zamanda **get** ve **set** metotlarının da eklenmiş olduğu görülüyor.

Otomatik özellik tanımlandıktan sonra istenirse daha sonra klasik get ve set bloğu olan özellik bildirimine de dönüştürülebilir. Bu durumda sınıfı kullanan istemci uygulamalarında herhangi bir değişiklik yapmaya gerek yoktur.

Otomatik özellik tanımlamanın daha çok **XML** ve **SQL** yapısında verilere erişmek için kullanılan **eşleştirme sınıflarının** (*mapping class*) bildiriminde kullanılır. Ama yine de otomatik özelliklerin aslında LINQ yapısında sıkça kullanılan isimsiz tiplerin yaratılması sırasında devreye girdiğini belirtmekte fayda var.

Öte yandan 2015 yılında final sürümüne geçmiş olan C# 6.0 sürümü ile birlikte otomatik özellik bildirimine yeni bir tanımlama biçimi daha gelmiştir. Bu tanımlama biçiminde otomatik özelliklere aynen sınıf değişkenlerinde olduğu gibi ilk değer ataması yapılabilmektedir. Örneğin aşağıdaki ifadeler artık geçerlidir;

```
public int No { get; set; } = 0;
public Guid Id { get; set; } = Guid.NewGuid();
public string Ad { get; set; } = "Adsız";
```

İlk değer ataması yukarıdaki gibi yapıldığında arka planda oluşturulan set bloğu çalışmaz. Yukarıdaki yazım şekli aşağıdaki klasik tanımlamanın bir kısa yazım şekli olarak düşünülebilir.

```
private string adEski = "Adsız";
public string AdEski
{
   get { return adEski; }
   set { adEski = value; }
}
```

Otomatik özelliklere ilk değer atama C# 6.0 ile gelmiştir

Genişletme Metotları (Extension Methods)

LINQ mimarisinin bir zorunluluğu olarak C# 3.0'a nesne yönelimli programlama konseptlerinden olmayan yeni bir özellik eklenmiştir. Bu özellik genişletme metotlarıdır. Genişletme metotları mevcut bir tipe yeni bir metodun eklenebilmesine olanak sağlar. Böylece türetme yapmadan ve mevcut tipin kullanılmasını engellemeden aynı tiplere yeni metotlar eklenebilir. Yine bu özelliğinde çıkış noktasında LINQ mimarisinde bulunan **Select, Where** gibi fonksiyonların mevcut koleksiyon tipleri ile çalışabilmesine olanak sağlamaktır.

Örneğin aşağıdaki LINQ ifadesini ele alalım;

```
List<Musteri> musteriler = new List<Musteri>();
musteriler.Add(new Musteri(3,"Sefer"));

var sonuclar = from m in musteriler
        select new { m.Ad, m.No };
```

Daha öncede belirttiğimiz gibi yukarıdaki LINQ ifadesi aşağıdaki gibi bir çağrıma denk gelmektedir.

```
var sonuclar3 = musteriler.Select(delegate(Musteri m)
                                  {
        return new { m.Ad, m.No };
            }
        );
```

Musteriler dizisi, IEnumerable<T>'yi uygulayan bir koleksiyon olmasına rağmen normal şartlarda içerisinde bulunmayan bir *Select* metodunun çağrılıdığını görüyoruz. İşte bu IEnumerable<T> üzerinden çağrılan *Select* metodu bir **genişletme metodudur** (*extension method*). Yukarıda çağrımı görülen *Select* metodu aslında direkt olarak *musteriler* nesnesi üzerinde değildir. Bu metot **System.Linq.Enumerable** sınıfı içinde özel bir şekilde bildirilmiş bir metotdur. Dolayısıyla koleksiyon nesneleri üzerinden **Select**, **Where** gibi metotların kullanılabilmesi için program içinde aşağıdaki isim alanı bildiriminin yapılması şarttır. **System.Linq.Enumerable** sınıfının erişilebildiği her noktada koleksiyon üzerinde **Select** ve **Where** gibi metotlar çağrılabilecektir.

Yukarıdaki **Select** metodunun çağrım ifadesinin ise derleyici tarafından aslında aşağıdaki çağrım ifadesine dönüştürüldüğünü söylersek sanırım genişletme metotlarının da CLR 2.0 uyumlu olduğunu söyleyebiliriz.

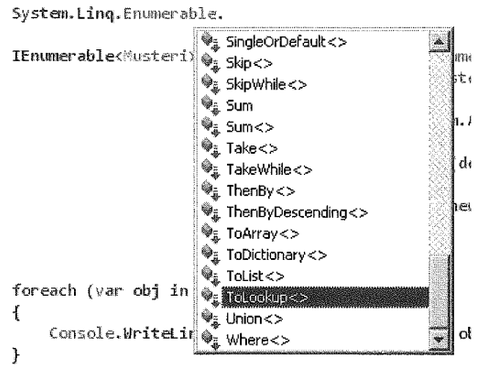
```
IEnumerable<Musteri> sonuclar4 = System.Linq.Enumerable.
        Select<Musteri,Musteri>(delegate(Musteri m)
            {return new Musteri( m.No, m.Ad );}
                );
```

Yani **Select** metodunun aslında .NET 3.5 içerisinde yer alan **Enumerable** isimli sınıf içinde yer aldığını görüyoruz. **Select** metodunun prototip bildirimi ise aşağıdaki gibidir.

```
namespace System.Linq
{
  public static class Enumerable
  {
     public static IEnumerable<TResult> Select<TSource, TResult>(this
IEnumerable<TSource> source, Func<TSource, TResult> selector);
  }
}
```


Select metodunun iki adet şablon tip aldığı görülmektedir. İlk şablon tip koleksiyondaki elemanların tipi, ikinci şablon tip ise **Select** metodunun döndüreceği koleksiyondaki elemanların tipini belirtiyor. Bu bildirimde yabancı olduğumuz kısım **Select** metodunun ilk parametresinin bildirim biçimidir. İlk parametre hangi tip için bir genişletme metodu tanımlanmak isteniyorsa o tipten olmalıdır. Böylece derleyici ilgili nesneyi bu metodu ilk parametre olarak geçirmektedir. Bildirim şekli, normal parametre bildiriminden farklı olarak this anahtar sözcüğü ile belirtilir.

System.Linq.Enumerable isimli sınıfın içerisinde tanımlı olan diğer **Extension** metotları ise aşağıdaki gibi gösterilebilir. Bu metotların tamamı **LINQ** mimarisi içerisinde kullanılan metotlardır. Bu metotların ne işe yaradıkları ileriki sayfalarda detaylı olarak anlatılacaktır.



Genişletme metotları ile ilgili önemli durumlar;

Genişletme metotları yapılar, sınıflar ve arayüzlere uygulanabilir.

Genişletme metotları yapı, sınıf ve arayüzler için uygulanabilir.

- Genişletme metotları static bir sınıf içerisinde tanımlanmalıdır.
- Genişletme metotlarının bildirimi static olarak yapılmalıdır.
- Genişletilecek tipin bildirimi, genişletme metodunun ilk parametresi olmak zorundadır. Bu parametre bildiriminin önüne **this** anahtar sözcüğü eklenmelidir.
- Genişletme metoduna erişilebilmesi için ilgili noktadan genişletme metoduna ait sınıfın erişilebilir olması gerekir.
- Sınıfın genişletme metodunun prototipine uygun başka bir metodu varsa çağrım sırasında öncelik sınıfın kendi metodudur.

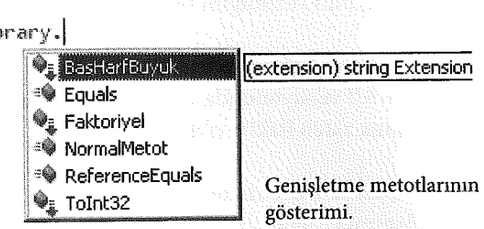
- Genişletme metotları da normal metotlar gibi **aşırı yüklenebilir** (*overload*).
- Bir tipe farklı static sınıflar içerisinde genişletme metodu eklenebilir. Bu durumda farklı yerlerde tanımlanan metotların imzalarının farklı olması gerekir.
- Genişletme metotları sadece C# diline özgü bir yapı değildir. Genişletme metotları IL içerisinde de **System.Runtime.CompilerServices** isim alanında bulunan **ExtensionAttribute** isimli nitelikle işaretlenirler. Böylece C# ile yazılan bir genişletme metodu herhangi bir .NET uyumlu programlama dili içinden de erişilebilir durumdadır.
- Taban sınıfa eklenen genişletme metodu, taban sınıftan türeyen sınıflardan da erişilebilir durumdadır. Örneğin **object** sınıfına eklenmiş bir genişletme metodu bütün tiplerde gözükür.

Genişletme metotlarının LINQ mimarisi içerisinde çok yoğun kullanıldığını ve ihtiyacın da bu noktadan çıktığını belirttik. Ancak, genişletme metotları LINQ haricinde de kullanışlı olabilecek bir özelliktir. Örneğin kendimiz .NET'in temel veri tipleri olan int, string gibi türetilemeyen veri tiplerine çok istediğimiz ama bir türlü yapılmayan metotları ekleyebiliriz. Örneğin string tipine, *ToInt32()* ve *BasHarfBuyuk()* ya da **int** tipine *Faktoriyel()* gibi genişletme metotlarını aşağıdaki gibi tanımlayabiliriz. Aşağıdaki kodları aynen yazdığınızda çalıştırılabilirdir.

```
using System;
using System.Linq;
using BTAkademi;

namespace BTAkademi
{
    public static class ExtensionLibrary
    {
        public static int ToInt32(this string str)
        {
            return Convert.ToInt32(str);
        }

        public static string BasHarfBuyuk(this string str)
        {
            return str.Substring(0, 1).ToUpper() +                    str.
Substring(1).ToLower();
    }

   public static int Faktoriyel(this int sayi)
   {
            int sonuc = 1;
            for (int i = sayi; i > 1; i--)
            {
                sonuc *= i;
```



```
            }
            return sonuc;
    }

                public static void NormalMetot(int a) { }
   }
}

namespace ExtensionMethods
{
  class Program
  {
    static void Main(string[] args)
    {
      string str = "Bu Bir Genişletme mEtodudur";
      string strSayi = "6";

      Console.WriteLine(str.BasHarfBuyuk());
      int sayi = strSayi.ToInt32();

      int faktoriyel = sayi.Faktoriyel();
    }
  }
}
```



Genişletme metotlarının gösterimi.

Genişletme metotlarına normal yöntemlerle de erişilebilir. Bu durumda genişletme metodunun içinde bulunduğu static sınıfı kullanılır. Visual Studio ile kod yazıyorsanız bu araç genişletme metotlarını farklı bir ikon ile göstermekte ve diğer metotlardan ayrıştırmaktadır.

Genişletme metotları normal yollarla aşağıdaki gibi de çağrılabilir. Zaten C# derleyicisi derleme aşamasında genişletme metotlarının çağrım ifadesini bu şekle dönüştürür.

```
int i = 10;
int sonuc = ExtensionLibrary.Faktoriyel(i);
//ya da
int sonuc2 = i.Faktoriyel();
```

Genişletme metotları LINQ mimarisinin .NET 2.0 uyumlu koleksiyonlar üzerinde uygulanabilmesi amacıyla eklenmiş sözdizimsel bir iyileştirmedir. Böylece genişletme metotlarının da CLR 2.0 uyumlu olduğunu görmüş olduk.

Lambda İfadeleri (Lambda Expressions)

Lambda ifadeleri C# 3.0'a eklenmiş belki de en önemli yeniliklerden biridir. Öyle ki LINQ yapısının tam anlamıyla kullanılabilmesi için olmak zorunda olan bir özelliktir. Lambda ifadeleri fonksiyonel programlama tekniğinde kodun veri olarak temsil edilmesini sağlayan ifadelerdir. **Lambda** ifadeleri C# dilinde yeni olmasına rağmen **Lisb**, **Python** ve **Schema** gibi programlama dillerinde öteden beri olan yapılardır. Lambda ifadelerinin algoritmayı daha doğrusu kodu veri olarak temsil edebilmesinden kaynaklanan gücü sayesinde çok karışık algoritmaların daha hızlı yazılmasına imkan tanımaktadır.

Peki, nedir Lambda ifadeleri? **Lambda ifadeleri** C# dilinin ilk versiyonlarından beri kullandığımız **Delegate** (*temsilci*) tipi ile doğrudan ilgilidir. Lambda ifadeleri, delegate nesnesi bekleyen fonksiyon parametrelerini metot bildirimi yapmaksızın kullanmamızı sağlar.

LINQ mimarisi içerisinde **lambda** ifadelerinin yeri çok önemlidir. Özellikle koleksiyonlar üzerinde sorgulama yaparken **Select** metoduna verilen seçme kriterlerinin aslında birer fonksiyon olduğunu ve bu fonksiyonların hızlı bir şekilde tanımlanabilmesi için **lambda** ifadelerinin yazıldığını belirtmeliyim.

Lambda ifadelerini kafanızda daha kolay canlandırabilmek için aynı örneğin C# 1.0, C# 2.0 ve C# 3.0 versiyonunda ne şekilde yapılabileceğini inceleyelim. C# 1.1 ve C# 2.0 versiyonuna zaten hakim olduğunuz için üzerinde çok durmayacağım.

Örneğimiz .NET 2.0 ile birlikte gelen **List<T>** sınıfının FindAll() isimli metodun bildirimi ve kullanımı ile ilgili olacak. **List<T>** şablon tipli sınıfın **FindAll** metodu parametre olarak bir temsilci nesnesi alır ve bu temsilciye göre seçim kriterleri belirlenir.

FindAll() metodunun prototipi aşağıdaki gibidir;

```
using System.Collections.Generic;
public class List<T>
{
    public List<T> FindAll(Predicate<T> match)
     { ... }
}

public delegate bool Predicate(T val);
```


C# 1.1'de generic yapılar olmadığı için C# 1.1 versiyonu yaklaşımı C# 2.0 derleyicisi kullanılarak yapıldığını varsayıyoruz. İsimsiz metotların olmadığı varsayımı altında örneği gerçekleştiriyoruz.

Buna göre C# 1.1 yaklaşımında **FindAll()** metodu aşağıdaki gibi kullanılır. Örnekte **int** tipinden elemanlar barındıran bir liste içerisinde filtreleme işlemi yapan bir fonksiyon tanımlanarak delegate yardımıyla **FindAll** metoduna gönderilmiştir.

```
static void Main()
{
    List<int> sayilar = new List<int>();
    sayilar.Add(1);
    sayilar.Add(2);
    sayilar.Add(3);
    sayilar.Add(4);
    sayilar.Add(5);
    sayilar.Add(6);

    Predicate<int> f = new Predicate<int>(KararVer);

    List<int> ciftSayilar = sayilar.FindAll(f);

    foreach (int i in ciftSayilar)
    {
        Console.WriteLine(i);
    }
}

static bool KararVer(int deger)
{
      return deger % 2 == 0;
}
```

Örneği derleyip çalıştırdığınızda ekrana sadece çift sayıların yazıldığını göreceksiniz. İstenirse farklı filtreleme fonksiyonları da yapılarak **FindAll** metodu ile birlikte çalışmasını sağlayabilirsiniz. Buradaki tek koşul filtreleme fonksiyonunun **Predicate** isimli temsilcinin prototipine uygun olmasıdır.

C# 2.0 ile birlikte gelen isimsiz metotlar sayesinde ise bu kullanım aşağıdaki hale getirilebilmektedir. C# 2.0 yaklaşımında görüldüğü üzere yeni bir fonksiyon tanımlaması yapılmaksızın fonksiyon çağrımı noktasında fonksiyon bloğu veri olarak gönderiliyor.

```
static void Main()
{
    List<int> sayilar = new List<int>();
    sayilar.Add(1);
    sayilar.Add(2);
    sayilar.Add(3);
    sayilar.Add(4);
    sayilar.Add(5);
    sayilar.Add(6);
    List<int> ciftSayilar = sayilar.FindAll( delegate(int val) {return val%2==0;} );

    foreach (int i in ciftSayilar)
    {
        Console.WriteLine(i);
    }
}
```

Yukarıdaki kod bloğunun da aynı sonucu vereceğini görebiliriz.

C# 3.0 ile brilikte ise lambda ifadelerinin yardımıyla fonksiyon parametresini çok daha basit ve sade bir şekilde parametre olarak gönderebiliyoruz. C# 3.0 ile gelen lambda ifadesi ile **Predicate** temsilcisi türünden nesneler aşağıdaki gibi tanımlanabilir.

```
Predicate f =  val =>  val % 2 = = 0; //lambda ifadesi
```

Yukarıdaki ifadede görüldüğü gibi => şeklinde yeni bir simge ile karşılaşıyoruz. =>, metodun gövdesi ile parametre listesini birbirinden ayırır.

Buna göre örneğimizi C# 3.0 yaklaşımında aşağıdaki gibi değiştirebiliriz;

```
static void Main()
{
    var sayilar = new List<int>(){1,2,3,4,5,6}; //koleksiyon ilklendirme
    var ciftSayilar = sayilar.FindAll( val => val % 2 ==0);

    foreach (int i in ciftSayilar)
    {
        Console.WriteLine(i);
    }
}
```

Görüldüğü gibi **lambda** ifadesi yardımıyla temsilciler birer veriymiş gibi fonksiyonlara parametre olarak gönderilebilirler. IL kodu içerisine bakıldığında lambda ifadelerinin birer isimsiz metot bildirimlerine dönüştürüldüğü görülecektir. Sonuç olarak lambda ifadeleri de CLR 2.0 uyumludur diyebiliriz.

Diğer Tanımlama Biçimleri

Aşağıda örnekleri verilmiş bütün lambda ifade tanımları uygun temsilciler tanımlı ise geçerlidir.

Kalıp:

(parametreler) => (ifade gövdesi)

Yukarıdaki **lambda** ifadesi tanımlama kalıbında bulunan ifade gövdesinde üretilen sonucun tipi neyse tanımlanan metodun geri dönüş değer tipi o olmaktadır. Metot parametrelerinin tipi ise çoğunlukla atanan delegate tipindeki şablon tiplerden çıkarsama yapılarak verilir.

Lambda ifade örnekleri:

```
//Expression Gövdeli Lambda
x => x + 2;
(int x) =>  x+1;

//Statement Gövdeli Lambda
x=> {return x+3};
(int x) =>  {return x+3};

(x,y) =>  x+ y;  //çok parametreli lambda
( ) => {return true};   //parametresiz lambda ifadesi
```

Lambda Gövdeli Metot ve Özellik Bildirimi

C# 6.0 ile birlikte ise Lambda ifadelerinin gücünden faydalanarak yazdığımız kod satır sayısını azaltacak güzel bir özellik gelmiştir. Buna göre eğer set bloğuna yazacağımız kod ya da metot gövdesine yazacağımız kod satırı sayısı 1 ya da daha az ise kod bloğunu süslü parantezler içine değil de lambda ifadesi ile bildirim satırına yazabilmekteyiz. Örneğin *Object* sınıfındaki *ToString()* isimli metodu TabloAdı isimli özelliği aşağıdaki gibi *"override"* edebiliriz.

```
abstract class Entity
{
  protected virtual string TabloAdı { get;}
  public abstract void Insert();
}

class Personel : Entity
{
  public override string ToString() => this.Ad;
  protected override string TabloAdı => "TBL_Personel";
}
```

Önemli Not : Lambda gövdeli metot ve özellik tanımlama C# 6.0 ile gelmiştir ve yalnızca tek satırlık kod blokları için geçerlidir.

C# geliştiricilerinden gelen yoğun istek üzerine ise C# 7.0 versiyonu ile birlikte metot ve özelliklere ek olarak lambda gövdeli yapıcı metot, yıkıcı metot ve get set blokları aşağıdaki gibi bildirilebilmektedir.

```
class Personel
{
    private string isim;
    public Personel(string isim) => this.isim =isim; // yapıcı metot
    ~Personel() => isim=""; // yıkıcı metot
    public string Isim
    {
        get => isim;
        set => isim = value;
    }
}
```

Lambda İfadeleri ve LINQ

Elbetteki lambda ifadelerinin de en güzel kullanım alanı LINQ mimarisindeki genişletme metotlarında (*select*, *where*) bulunan parametrelerdir.

Daha önce incelediğimiz **Enumerable** sınıfının **Select** isimli genişletme metodunu bu vesile ile tekrar inceleyelim. **Select** metodunun prototipi;

```
namespace System.Linq
{
  public static class Enumerable
  {
     public static IEnumerable<TResult> Select<TSource, TResult>(this
IEnumerable<TSource> source, Func<TSource, TResult> selector);
  }
}
```

Görüldüğü üzere *Select* genişletme metodunun ilk parametresi filtreleme yapılacak listeyi, ikinci parametresi *Func* isimli bir filtre fonksiyonudur. *Func* isimli delegate iki adet şablon tipe sahip, birinci şablon tip *Select* metodunun ilk şablon tipi ile aynı tip,

diğeri ise *TResult* isimli geri dönüş listesinin tipini belirten bir tip. Buna göre bu *Select* fonksiyonu *List<int>* nesnesi üzerinden aşağıdaki gibi çağrılabilir.

```
List<int> sayilar = new List<int> { 2, 3, 4, 5, 6, 7, 8 };
IEnumerable<int> ciftSayilar = sayilar.Where(x => x % 2 == 0);
```

Son olarak lambda ifadelerini ve genişletme metotlarını daha iyi anlayabilmek için **Enumerable** içinde bulunan **Where** metodunu biz nasıl yapabiliriz, inceleyelim.

Select metodunun ilk parametresinin genişletme metodu tanımından geldiğini ve yukarıdaki çağrımda bulunan sayilar isimli koleksiyona denk geldiğini unutmamak gerekir.

Lambda İfadeleri ve Genişletme Metotlarına Bir Örnek

Aşağıdaki program ile **IEnumerable** içerisinde bulunan **Select** isimli metodun başka bir isimle nasıl geliştirilebileceği örneklendirilmiştir.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace MyQueryExpression
{
    class Program
    {
        static void Main(string[] args)
        {
                List<Personel> personeller = new List<Personel> { new Personel
{ Ad = "Ali", No = 3 } };

                personeller.Add(new Personel { Ad = "Ali", No = 27 });
                personeller.Add(new Personel { Ad = "Sefer", No = 5 });
            personeller.Add(new Personel { Ad = "Ali", No = 6 });
                personeller.Add(new Personel { Ad = "Oguz", No = 8 });

        //SelectWhere List<T> sınıfına bizim eklediğimiz genişletme metodudur.
        var aliOlanlar = personeller.BTASelect(p => p.Ad == "Ali");

                var digerFiltre = personeller.BTASelect(p => p.No > 5 &&
p.Ad.Length>3);
```

```
    foreach (var ali in aliOlanlar)
      {
       Console.WriteLine("Ad : " + ali.Ad + " No:" +
ali.No.ToString());
          }
    }
}

public delegate bool ConditionTester<T>(T val);

public static class MyExtensions
{
    public static IEnumerable<T> BTASelect<T>(this List<T> liste,
ConditionTester<T> secici)
  {
          foreach (T val in liste)
    {
                    if (secici(val))
                    {
                              yield return val;
                    }
          }
    }
}

public class Personel
{
    public string Ad;
    public int No;
}
}
```

Bu örnek lambda ifadeleri, genişletme metotları, **var** deyimi, koleksiyon ilklendirme gibi yeni özelliklerin bir arada kullanıldığı duruma güzel bir örnek olarak verilebilir. Zaten LINQ merkezinde toplanan birçok metot da buna benzer şekillerde tanımlanmıştır.

Kısmi Metotlar (Partial Metotlar)

C# 2.0 ile birlikte bir sınıfı birden fazla fiziksel dosyaya bölen kısmi sınıflar kavramı geldiğini hatırlayalım. Böylece otomatik olarak üretilen (*designer generated*) kodlarla, programcının yazmış olduğu kodların birbirinden ayrıştırılması sağlandı. Fiziksel olarak farklı dosyalarda bulunan kısmi tipler çalışma zamanında birleştirilerek tek bir sınıf gibi ele alındıkları için programatik olarak herhangi bir kısıtı bulunmamaktadır. Kısmı tiplerin tanımlanması ile ilgili örnekler önceki bölümlerde açıklanmıştır.

Kısmi metot kavramı ise C# 3.0 ile gündememize giren bir kavramdır. Kısmi metotlar yine özellikle otomatik olarak üretilen kod blokları ile bu kod bloklarını kullanan istemci uygulama arasında **entegrasyon** (*mesaj alışverişi*) sağlayacak bir yapı olarak düşünülebilir. Kısmi metotlar şu anda daha çok LINQ to SQL sağlayıcısında sıkça kullanılmaktadır. Temel kullanım yeri **entity** denilen tabloyu temsil eden nesnelerin özelliklerinde bir güncelleme olmadan önce ve olduktan sonra uygulama ortamına mesaj göndererek istemcinin ek fonksiyoneliteler katmasıdır. Normal şartlarda bu olay event tabanlı mekanizma ile yapılabilir. Ancak event mekanizması hem uygulanması açısından zordur hem de çalışma zamanı ile ilgili olduğu için performans problemleri vardır. Öte yandan kısmi metotlarda bu tarz bir problem yoktur. Daha fazla uzatmadan bir örnek ile kısmi metotların nasıl tanımlandığını inceleyelim.

Öncelikle vurgulamak gerekir ki kısmi metotların kullanımı ancak sistemde otomatik olarak üretilen bir kod (*designer generated*) var ise anlamlı. Çünkü hiçbir programcı araçların ürettiği koda müdahale etmek istemez, ancak onları da bir şekilde genişletmek ister. İşte kısmi metotlar tam bu noktada devreye girer; basit anlamda event mekanizması kurmak istiyorsak kullanabiliriz.

Örneğin, yaptığımız bir araçla ile (bu araç zaten VS.NET 2008 içerisinde mevcut) **Linq to Sql** sağlayıcısı için otomatik olarak sınıflar yaratmak istiyoruz. Yaratacağımız bu sınıflar veritabanı tablolarını temsil edecekler. Ve aracımızın aşağıdaki gibi bir *Personel* sınıfını otomatik olarak ürettiğini varsayalım.

```
public partial class Personel
{
  private string ad;
  public string Ad
  {
    get {
      return ad;
    }
    set {
      ad = value;
    }
  }
}
```

Bu sınıfı kullanacak istemci uygulamanın ad değişkeni değiştirilmeden önce veya değiştirildikten sonra haberdar edilmek istendiğini düşünelim. Bu durumda aracımızın ürettiği koda müdahale etmeden bunu yapmanın tek yolu event mekanizmasını oluşturmaktır. **Set** bloğu içinde değişken atanmadan önce ve atandıktan sonra olmak üzere iki kez **event** tetiklenerek istemci tarafa mesaj iletilmelidir.

İşte kısmi metotlar bu görevi daha hızlı ve daha efektif bir şekilde yapmak için tasarlanmıştır. Yine kısmi tipler içerisinde tanımlanan metotlar gövdesi olmaksızın aşağıdaki şekilde tanımlanırlar;

```
public partial class Personel
{
  private string ad;
  public string Ad
  {
    get {
      return ad;
    }
    set {
      BeforeUpdate();
      ad = value;
      AfterUpdate();
    }
  }
  partial void BeforeUpdate();
  partial void AfterUpdate();
}
```

Görüldüğü üzere set bloğu içinde önce **BeforeUpdate()** metodu ardından **AfterUpdate()** metodu çağrılmaktadır. Ancak her iki metodun da herhangi bir gövdesi bulunmamaktadır. Buradaki amaç ise bu metotların yine partial olarak tanımlanacak başka bir *Personel* sınıfı içerisinde kullanmak.

Bu aşamadan sonra (otomatik üretilen *Personel* sınıfı oluştuktan sonra) bu sınıfın bulunduğu isim alanında yine *Personel* isimli başka bir sınıf aşağıdaki şekilde kısmi olarak bildirilebilir.

```
public partial class Personel
{

}
```

Bu durumda Ad özelliğinin set bloğundaki **BeforeUpdate()** metodu ve **AfterUpdate()** metodu çağrım ifadeleri derleme aşamasında es geçilerek **IL**'e eklenmesi engellenir. Yani bu metotlar sanki yokmuş gibi davranılır.

Buna rağmen istenirse aşağıdaki gibi kısmi metotlar bildirilerek Ad alanının her değiştirilmesi durumunda kendi kod bloklarımızın çalışmasını sağlayabiliriz.

```
public partial class Personel
{
  partial void BeforeUpdate()
```



```
  {
    //değer atanmadan önce bişeyler yapmak istiyorsak.
  }

  partial void AfterUpdate()
  {
    //değer atadıktan sonra bişeyler yapmak istiyorsak.
  }
}
```

Linq to Sql sağlayıcısında kısmi metotlardan oldukça fazla faydalanılmaktadır. Kısmi metotlarla ilgili bilinmesi gerekenler;

- Kısmi metotlar sadece kısmi tipler içerisinde bildirilebilir.
- Kısmi metotların gövdesi her zaman olmak zorunda değildir. Gövdesi olmayan kısmi metotlar derleme aşamasında IL içerisine eklenmez.
- Kısmi metotlar **partial** anahtar sözcüğü ile bildirilir.
- Kısmi metotlar ile ilgili en önemli kısıt bu metotların geri dönüş değerlerinin **void** olma zorunluluğudur.
- Kısmi metotlar override, virtual, new, abstract, sealed sözcükleri ile nitelendirilemezler.
- Kısmi metotlar **static** olabilir.
- Kısmi metotlar erişim belirleyicilerle ile bildirilemezler. Varsayılan olarak private olarak davranırlar.
- Kısmi metotların parametreleri params, ref, this gibi anahtar sözcükler ile tanımlanabilir.

Gelişmiş LINQ Sorgu İfadeleri

Bu bölümün başında incelediğimiz **LINQ sorgularının** daha detaylı inceleneceği bu bölümde farklı kaynak veriler ile ne şekilde çalışıldığı, ileri düzey sorgulamarın nasıl yapılacağı detaylı olarak ele alınacaktır.

Bir **LINQ sorgusu** her zaman için from deyimi ile başlar, **from** deyimi ile sorgulama yapılacak kaynak liste belirtilir. Ve genellikle de **select** deyimi ya da ileride göreceğimiz group deyimi ile sonlanır. **Select** deyimi ile sonuç döndürme **group** deyimi ile ise bir anahtar alana göre gruplama yapılır. İlk **from** deyiminden sonra opsiyonel olarak **let**, yine **from** ya da **where** deyimleri gelebilir. Birden fazla from kullanıldığında birleştirme işlemi yapmak için iki kaynağın kullanıldığı belirtilir. **Where** deyimi filtreleme yapmaya, **let** deyimi ise sorgu içinde değişken tanımlamaya yarar. Ayrıca opsiyonel olarak **select** ve **group** deyimlerinden önce sıralama işine yarayan **orderby** deyimi kullanılabilir. Son olarak **into** deyimi kullanılarak sql'deki gibi iç içe sorgulamalar yapılabilir.

Yukarıda bahsi geçen *let, orderby, select, from, group, into, where* gibi komutların tamamı metot çağrımlarına dönüştürülür. Bu metotların dışında da kullanılabilecek birçok fonksiyonda bulunmaktadır. Örneğin **aggregation** fonksiyonları ve **sayfalama** fonksiyonları bunlara örnek olarak verilebilir. Bütün bu fonksiyonlar işlevlerine göre aşağıdaki gibi listelenmiştir.

Açıklama	Metotlar
Kısıtlama Operatörleri (Restriction Operators)	Where
İzdüşüm Operatörleri (Projection Operators)	Select
Gruplama Operatörleri (Grouping Operators)	Group
Sıralama Operatörleri (Ordering Operators)	OrderBy, ThenBy, Reverse
Bölümleme Operatörleri (Partitioning Operators)	Take, Skip, TakeWhile, SkipWhile
Dönüştürme Operatörleri (Conversion Operators)	ToArray, ToList, ToDictionary, OfType
Küme Operatörleri (Set Operators)	Distinct, Union, Intersect, Except
Grup Operatörleri (Aggregate Operators)	Count, Sum, Min, Max, Averaga, Fold
Ölçüm Operatörleri (Quantifiers Operators)	Any, All
Eleman Operatörleri (Element Operators)	First, FirstOrDefault, ElementAt
Ve Diğer Operatörler	Concat, EqualAll ...
Kaynak: *MSDN & www.csharpnedir.com*	

LINQ sorgularının arka planda genişletme metodu çağrımlarına dönüştürüldüğünü unutmamak gerekir.

Bu aşamadan sonra çeşitli örnekler yaparak her bir linq deyiminin nasıl yazıldığını inceleyelim: Öncelikli olarak bütün örneklerde kullanılabilecek bir liste oluşturalım. Bu listeyi oluşturmak için aşağıdaki gibi *Urun* sınıfı, ve *UrunKategori* sınıfı tanımlıyoruz. Konumuz LINQ ifadelerini öğrenmek olduğu için listeyi manuel bir şekilde oluşturuyoruz, istersek veritabanından da oluşturabiliriz. Bütün örneklerde listeyi elde etmek için ise *UrunleriGetir()* ve *KategorileriGetir()* metotlarını kullanacağız.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace YardimciSiniflar
```



```
{
  public class Urun
  {
    public int UrunId { get; set; }
    public string UrunAdi { get; set; }
    public double Fiyat { get; set; }
    public int KategoriId { get; set; }
    public DateTime SatisBaslangicTarihi { get; set; }
    public DateTime SatisBitisTarihi { get; set; }

    public override string ToString()
    {
return "UrunId: " + UrunId.ToString() + " Urun Adı:" + UrunAdi + " Fiyat:" + Fiyat.ToString();
    }
  }

  public class Kategori
  {
    public int KategoriId { get; set; }
    public string KategoriAdi { get; set; }

    public override string ToString()
    {
      return "Kategor ID : " + KategoriId.ToString() +" Kategori : "+ KategoriAdi;
    }
  }

  public class Helper
  {
    public static List<Urun> UrunleriGetir()
    {
      List<Urun> urunler = new List<Urun>()
      {
        new Urun{UrunId=1,UrunAdi="Coca Cola",KategoriId=2,Fiyat=3,
SatisBaslangicTarihi=new DateTime(2008,3,1),
 SatisBitisTarihi
=new DateTime(2008,4,13)},

        new Urun{UrunId=1,UrunAdi="Masa",KategoriId=6,
Fiyat=200, SatisBaslangicTarihi=new DateTime(2007,12,23),                SatisBitisTarihi =new
DateTime(2008,1,4)},

        new Urun{UrunId=1,UrunAdi="C#",KategoriId=1,
Fiyat=49,  SatisBaslangicTarihi=new DateTime(2005,3,1),
SatisBitisTarihi =new DateTime(2007,4,7)},
```

```
            new Urun{UrunId=1,UrunAdi="VB.NET",KategoriId=1,
Fiyat=25, SatisBaslangicTarihi=new DateTime(2007,1,1),                SatisBitisTarihi =new
DateTime(2007,3,12)},

            new Urun{UrunId=4,UrunAdi="MeyveSuyu",KategoriId=2,Fiyat=4,
SatisBaslangicTarihi=new DateTime(2006,2,14),
SatisBitisTarihi =new
DateTime(2007,6,23)},

            new Urun{UrunId=5,UrunAdi="Buzdolabı",KategoriId=4,Fiyat=800,
SatisBaslangicTarihi=new DateTime(2007,1,4),
SatisBitisTarihi =new
DateTime(2007,7,7)},

            new Urun{UrunId=8,UrunAdi="Dell Bilgisayar",
KategoriId=3,Fiyat=1200,
SatisBaslangicTarihi=new DateTime(2008,3,14),                         SatisBitisTarihi =new
DateTime(2008,9,12)},

            new Urun{UrunId=54,UrunAdi="Hesap Makinası",
KategoriId=3,Fiyat=300,
SatisBaslangicTarihi=new DateTime(2006,3,1),                          SatisBitisTarihi =new
DateTime(2007,3,28)},

            new Urun{UrunId=32,UrunAdi="Koltuk Takımı",
KategoriId=6,Fiyat=3500,
SatisBaslangicTarihi=new DateTime(2008,1,1),                          SatisBitisTarihi
=new DateTime(2008,2,2)},

            new Urun{UrunId=54,UrunAdi="T.C. Anayasası",
KategoriId=1,Fiyat=15,
SatisBaslangicTarihi=new DateTime(2005,11,11),                        SatisBitisTarihi
=new DateTime(2005,10,15)},

            new Urun{UrunId=12,UrunAdi="Pocket PC",KategoriId=3,Fiyat=990,
SatisBaslangicTarihi=new DateTime(2005,1,4),
SatisBitisTarihi =new DateTime(2006,11,6)},

            new Urun{UrunId=10,UrunAdi="XML Kitabı",KategoriId=1,Fiyat=38,
SatisBaslangicTarihi=new DateTime(2008,1,1),
SatisBitisTarihi =new DateTime(2008,4,9)}
        };
        return urunler;
    }

    public static List<Kategori> KategorileriGetir()
    {
```



```
        List<Kategori> kategoriler = new List<Kategori>()
        {
            new Kategori{KategoriAdi="Kitaplar",KategoriId=1},
        new Kategori{KategoriAdi="İçecek",KategoriId=2},
        new Kategori{KategoriAdi="Elektronik",KategoriId=3},
        new Kategori{KategoriAdi="Beyaz Eşya",KategoriId=4},
        new Kategori{KategoriAdi="Mobilya",KategoriId=6},
        new Kategori{KategoriAdi="DVD Medya",KategoriId=7}
        };
        return null;
    }
  }
}
```

Operatörlerden sadece sık kullanılanlara ait örnekler aşağıda verilmiştir.

Kısıtlama Operatörünün Örnek Kullanımı

Bu örnekte Kitaplar kategorisinde fiyatı *15*'den büyük olan ürünler listeleniyor.

```
List<Kategori> kategoriler = Helper.KategorileriGetir();
List<Urun> urunler = Helper.UrunleriGetir();

var sonuclar1 = from c in urunler
        where c.KategoriId == 1 && c.Fiyat > 15
        select c;

foreach (var eleman in sonuclar1)
{
  Console.WriteLine(eleman);
}
```

Sorgunun çıktısı aşağıdaki gibi olacaktır.

```
UrunId: 1 Urun Adı:C# Fiyat:49
UrunId: 1 Urun Adı:VB.NET Fiyat:25
UrunId: 10 Urun Adı:XML Kitabı Fiyat:38
```

Bundan sonraki örneklerde urunler ve kategoriler koleksiyonunun **Helper** sınıfı yardımıyla alındığı varsayılarak yapılacaktır.

İzdüşüm (Select) Operatörünün Örnek Kullanımı

Bu örnekte Select ifadesinden koleksiyon içinde olmayan bir takım verilerin nasıl döndürüleceği belirtilmiştir.

```
var sonuclar2 = from c in urunler
      where c.KategoriId == 1 && c.Fiyat > 15
      select new {Ad= c.UrunAdi.ToUpper(),c.KategoriId,
                              KDVli = c.Fiyat*1.18};

foreach (var eleman in sonuclar2)
{
   Console.WriteLine(eleman);
}
```

Sorgunun çıktısı aşağıdaki gibi olacaktır.

```
{ Ad = C#, KategoriId = 1, KDVli = 57,82 }
{ Ad = VB.NET, KategoriId = 1, KDVli = 29,5 }
{ Ad = XML KİTABI, KategoriId = 1, KDVli = 44,84 }
```

İzdüşüm (SelectMany) Operatörünün Örnek Kullanımı

SelectMany genişletme metodu ile birden fazla from sözcüğünü kullanma imkanına sahibiz.

```
string[] cumleler = { "Merhaba Ali", "Ben Geldim", "Yarın maç var.", "C#
3.0 çok iyi olmuş" };
var kelimeler= from cumle in cumleler
         from kelime in cumle.Split(' ')
         where kelime.Length>4
         select kelime;

foreach (var eleman in kelimeler)
{
   Console.WriteLine(eleman);
}
```

Sorgunun çıktısı aşağıdaki gibi olacaktır.

```
Merhaba
Geldim
olmuş
```

Yukarıdaki örnekten de görüldüğü üzere iki farklı veri kaynağından **from** deyimleri yardımıyla veri çekilebilmektedir.

Bölümleme (Take) Operatörünün Örnek Kullanımı

Gelen sonuç listesinden ilk 3 kaydı döndüren sorgu;

```
var sonuclar4 = (from urun in urunler
         select urun).Take(3);

foreach (var eleman in sonuclar4)
{
   Console.WriteLine(eleman);
}
```

Bu sorgunun çıktısı aşağıdaki gibi olacaktır;

```
UrunId: 1 Urun Adı:Coca Cola Fiyat:3
UrunId: 1 Urun Adı:Masa Fiyat:200
UrunId: 1 Urun Adı:C# Fiyat:49
```

Bölümleme (Skip) Operatörünün Örnek Kullanımı

Gelen sonuç listesinden atlanacak eleman adedini belirttiğimiz ve geriye kalan sonuçları döndüren sorgu;

```
var sonuclar4 = (from urun in urunler
         select urun).Skip(5);

foreach (var eleman in sonuclar4)
{
   Console.WriteLine(eleman);
}
```

Bu sorgunun çıktısı aşağıdaki gibi olacaktır; (İlk 5 kayıttan sonrakiler alınmıştır.)

```
UrunId: 5 Urun Adı:Buzdolabı Fiyat:800
UrunId: 8 Urun Adı:Dell Bilgisayar Fiyat:1200
UrunId: 54 Urun Adı:Hesap Makinası Fiyat:300
UrunId: 32 Urun Adı:Koltuk Takımı Fiyat:3500
UrunId: 54 Urun Adı:T.C. Anayasası Fiyat:15
UrunId: 12 Urun Adı:Pocket PC Fiyat:990
UrunId: 10 Urun Adı:XML Kitabı Fiyat:38
```

Bölümleme (SkipWhile ve TakeWhile) Operatörünün Örnek Kullanımı

Atlanacak ve alınacak eleman adedinin bir koşula bağlandığı sorgu ifadeleridir. Örneğin, fiyatı 10 TL'nin altına kadar düşen ürünleri al (*TakeWhilw*) ya da atla (*SkipWhile*) ifadeleri aşağıdaki şekilde belirtilebilir. Her iki metodun da parametresi lambda ifadesi olarak belirtilmiştir.

```
var sonuclar5 = (from urun in urunler
        select urun).TakeWhile( u => u.Fiyat > 10);

var sonuclar6 = (from urun in urunler
        select urun).SkipWhile( u => u.Fiyat > 10);
```

Sıralama (OrderBy, OrderByDescending) Operatörünün Örnek Kullanımı

Listenin istenilen bir kritere göre sıralanmasını sağlayan operatörlerdir. Örneğin ürünlerin isimlerinin içerdiği karakter sırasına göre bütün ürünler aşağıdaki gibi sıralanabilir.

```
var sonuclar7 = from urun in urunler
        orderby urun.UrunAdi.Length
        select urun.UrunAdi;

foreach (var eleman in sonuclar7)
{
  Console.WriteLine(eleman);
}
```

Sorgunun çıktısı aşağıdaki gibi olacaktır.

```
C#
Masa
VB.NET
Coca Cola
Buzdolabı
Pocket PC
Meyve Suyu
XML Kitabı
Koltuk Takımı
Hesap Makinası
T.C. Anayasası
Dell Bilgisayar
```

Tersten sıralama ise **OrderByDescending** genişletme metodu ile yapılır. LINQ ifadesinde aşağıdaki gibi kullanılabilir.

```
var sonuclar7 = from urun in urunler
        orderby urun.UrunAdi.Length descending
        select urun.UrunAdi;
```


Sıralama (ThenBy, ThenByDescending, Reverse) Operatörünün Örnek Kullanımı

Birden fazla kritere göre sıralama yapılacak ise **ThenBy** genişletme metodu kullanılır. LINQ ifadesinde ise aşağıdaki gibi gösterilir;

Örnekte bütün ürünler kategori numarasına göre küçükten büyüğe doğru sıralanmış, aynı kategoride olan ürünlerde kendi içinde büyükten küçüğe doğru sıralanmıştır.

```
var sonuclar8 = from urun in urunler
        orderby urun.KategoriId,urun.Fiyat descending
        select new{urun.KategoriId,urun.Fiyat,
urun.UrunAdi};

foreach (var eleman in sonuclar8)
{
  Console.WriteLine(eleman);
}
```

Sorgunun çıktısı aşağıdaki gibi olacaktır;

```
{ KategoriId = 1, Fiyat = 49, UrunAdi = C# }
{ KategoriId = 1, Fiyat = 38, UrunAdi = XML Kitabı }
{ KategoriId = 1, Fiyat = 25, UrunAdi = VB.NET }
{ KategoriId = 1, Fiyat = 15, UrunAdi = T.C. Anayasası }
{ KategoriId = 2, Fiyat = 4, UrunAdi = Meyve Suyu }
{ KategoriId = 2, Fiyat = 3, UrunAdi = Coca Cola }
{ KategoriId = 3, Fiyat = 1200, UrunAdi = Dell Bilgisayar }
{ KategoriId = 3, Fiyat = 990, UrunAdi = Pocket PC }
{ KategoriId = 3, Fiyat = 300, UrunAdi = Hesap Makinası }
{ KategoriId = 4, Fiyat = 800, UrunAdi = Buzdolabı }
{ KategoriId = 6, Fiyat = 3500, UrunAdi = Koltuk Takımı }
{ KategoriId = 6, Fiyat = 200, UrunAdi = Masa }
```

Diğer bir operatör ise **Reverse** genişletme metodu ile çalıştırılır. **Reverse** mevcut koleksiyonun ters çevrilmesini sağlar. Örnek kullanım aşağıdaki gibidir.

```
var sonuclar8 = (from urun in urunler
        orderby urun.KategoriId,urun.Fiyat descending
        select new{urun.KategoriId,urun.Fiyat,
urun.UrunAdi}
        ).Reverse();
```

Yukarıdaki sorgunun çıktısı ise bir önceki sorgunun çıktısının tam tersi olacaktır.

Gruplama (Group, Count, Sum) Operatörünün Örnek Kullanımı

LINQ içerisinde bulunan **group .. by.. into** deyimi ile sql dilinde bulunan groupby ifadeleri benzerdir. Gruplanmış bir veri kümesinde **Sum, Count** gibi gruplamalı fonksiyonlar kullanılabilir. Gruplamanın yapıldığı alan ie **Anahtar** (*Key*) olarak adlandırılır.

Örnek sorgumuzda, her bir ürün kategorisinde toplam kaç adet ürün bulunduğu ve bu ürünlerin neler olduğu listelenmektedir.

```
var sonuclar9 = from urun in urunler
        group urun by urun.KategoriId into grup
        select new {grup.Key,Urunler = grup ,Adet=grup.Count()};

foreach (var grupKey in sonuclar9)
{
  Console.WriteLine("Kategorisi " + grupKey.Key + " olan
ürünler.Toplam "          + grupKey.Adet + " ürün.");

  foreach (var urun in grupKey.Urunler)
  {
    Console.WriteLine("    " +urun.UrunAdi);
  }
}
```

Örnekten de görüleceği üzere iki adet döngü kurulmuştur. Birinci döngü grupları (kategori bazında) dolaşırken, ikinci döngü ile ilgili gruptaki ürünler dolaşılmaktadır. Ayrıca **Select** ifadesinden grup nesnesi üzerinden **Count()** isimli grup fonksiyonu çağrılarak her bir kategoride kaç ürünün olduğu belirlenmiştir.

Sorgumuzun çıktısı ise aşağıdaki gibi olacaktır.

```
Kategorisi 2 olan ürünler.Toplam 2 ürün.
  Coca Cola
  Meyve Suyu
Kategorisi 6 olan ürünler.Toplam 2 ürün.
  Masa
  Koltuk Takımı
Kategorisi 1 olan ürünler.Toplam 4 ürün.
  C#
  VB.NET
  T.C. Anayasası
  XML Kitabı
Kategorisi 4 olan ürünler.Toplam 1 ürün.
  Buzdolabı
```



```
Kategorisi 3 olan ürünler.Toplam 3 ürün.
  Dell Bilgisayar
  Hesap Makinası
  Pocket PC
```

Gruplama birden fazla alana göre de yapılabilir. Bu durumda her bir grup için ayrı ayrı grup değişkeni tanımlanmalıdır.

Aşağıdaki örnek sorguda iki seviyeli gruplama yapılmaktadır. Hem ürün kategorisi bazında hem de ürünlerin satışa çıkarıldığı yıl bazında gruplama yapılmaktadır. Yani her bir ürün kategorisinde hangi yıllarda hangi ürünlerin çıktığı merak ediliyorsa aşağıdaki gibi çoklu grup yapısı oluşturulabilir. Çoklu gruplamanın nasıl yapıldığına dikkat ediniz.

```
var sonuclar10 =from urun in urunler
        group urun by new {k1= urun.KategoriId,
            k2=urun.SatisBaslangicTarihi.Year }
into grup
        orderby grup.Key.k1
        select new { grup.Key.k1,grup.Key.k2 , Urunler = grup,          Adet = grup.
Count() };

foreach (var grupKey in sonuclar10)
{
  Console.WriteLine("Kategorisi "+grupKey.k1 + " Satış Yılı :" +          grupKey.
k2 + " olan ürünler\n");
  foreach (var item in grupKey.Urunler)
  {
    Console.WriteLine("    " + item.UrunAdi);
  }
  Console.WriteLine();
}
```

Yukarıdaki sorgu ifadesinin çıktısı ise aşağıdaki gibi olacaktır.

```
Kategorisi 1 Satış Yılı :2005 olan ürünler
  C#
  T.C. Anayasası
Kategorisi 1 Satış Yılı :2007 olan ürünler
  VB.NET
Kategorisi 1 Satış Yılı :2008 olan ürünler
  XML Kitabı
Kategorisi 2 Satış Yılı :2008 olan ürünler
  Coca Cola
Kategorisi 2 Satış Yılı :2006 olan ürünler
  Meyve Suyu
Kategorisi 3 Satış Yılı :2008 olan ürünler
```

```
    Dell Bilgisayar
  Kategorisi 3 Satış Yılı :2006 olan ürünler
    Hesap Makinası
  Kategorisi 3 Satış Yılı :2005 olan ürünler
    Pocket PC
  Kategorisi 4 Satış Yılı :2007 olan ürünler
    Buzdolabı
  Kategorisi 6 Satış Yılı :2007 olan ürünler
    Masa
  Kategorisi 6 Satış Yılı :2008 olan ürünler
    Koltuk Takımı
```

Her bir ürün kategorisinde bulunan ürünlerin toplam fiyatı ise *Sum* isimli grup fonksiyonu yardımıyla aşağıdaki gibi bulunabilir.

```
var sonuclar11 = from urun in urunler
        group urun by urun.KategoriId into grup
        select new { grup.Key, Urunler = grup, ToplamFiyat =
grup.Sum(u=>u.Fiyat) };

foreach (var grupKey in sonuclar11)
{
  Console.WriteLine("Kategorisi " + grupKey.Key + " olanların toplamı: "
+ grupKey.ToplamFiyat);
}
```

Sum genişletme metodunun parametresinin bir **lambda** ifadesi olduğuna dikkat ediniz.

Diğer grup fonksiyonları olan **Max** grubun içindeki en büyük değeri, **Min** grubun içindeki en küçük değeri, **Average** ise ortalama değeri hesaplar.

Küme (Distinct, Union, Intersect, Except) Operatörlerinin Örnek Kullanımı)

Hangi ürün kategorilerine ait ürünlerin olduğunu bulabilmek için **Distinct** genişletme metodu kullanılabilir. **Distinct** metodu, koleksiyon içinde tekrarlanan kayıtları tek bir kayda indirgemeye yarar.

```
var sonuclar12 = (from urun in urunler

        select urun.KategoriId).Distinct();

Console.Write("Ürün bulunan kategoriler : ");
foreach (var eleman in sonuclar12)
{
  Console.Write(eleman+" ");
}
```


Programın çıktısı aşağıdaki gibi olacaktır.

```
Ürün bulunan kategoriler : 2 6 1 4 3
```

Union genişletme metodu iki koleksiyonu birleştirir. SQL'deki **Union** ifadesi gibidir. Örnek kullanım aşağıdaki gibidir.

```
int[] sayilar1 = { 4, 2, 1, 2 };
int[] sayilar2 = { 3, 2, 3, 6 };

var sonuclar13 = sayilar1.Union(sayilar2);
foreach (var eleman in sonuclar13)
{
  Console.WriteLine(eleman);
}
```

Yukarıdaki örnekte *sonuclar13* dizininin içerisindeki elemanlar 1, 2, 3, 4, 6 olacaktır.

Intersect metodu ise iki koleksiyonda da bulunan yani kesişen elemanları alır. Örnek kullanım;

```
var sonuclar13 = sayilar1.Union(sayilar2);
```

Bu durumda *sonuclar13* dizisinin {2} olmak üzere tek bir elemanı olacaktır.

Diğer bir metot olan **Except** ise, birinci dizide (nesne) olup da ikinci dizide (parametre) olmayan elemanların sonuç kümesini döndürür.

Çevirme (ToArray, ToList, ToDictionary, OfType) Operatörlerinin Örnek Kullanımı

Herhangi bir LINQ ifadesini istediğimiz anda bir **Array**, **List** ya da **Dictionary** tipinden bir koleksiyona dönüştürebiliriz. Örnek bir LINQ ifadesinin List nesnesine dönüştürülmesi aşağıdaki gibi yapılabilir;

```
var sonuclar = (from urun in urunler
        select urun.KategoriId).ToList();
```

OfType isimli genişletme metodu ise liste içerisinde uygun tipe dönüşebilen elemanların filtrelenmesini sağlar. Örneğin aşağıdaki örnekte tipi **double** olan elemanlar filtreleniyor.

```
object[] numbers = { 2, null, "ali", 'C', 5,3.0,"Veli",3};
var doubleOlanlar = numbers.OfType<double>();
```

Yukarıdaki örnekte **doubleOlanlar** dizisinde sadece 3.0 değerinin olduğu görülecektir.

Element (First, FirstOrDefault, ElementAt) Operatörlerinin Örnek Kullanımı

Koleksiyon içerisindeki tek bir elemana erişmek için kullanılan metotlardır. *ElementAt* metodu bir parametre alır ve hangi indeks'teki elemana erişileceği belirtilir. Örnek kullanım,

Urun u = urunler.ElementAt(3);

First metodu listedeki ilk elemanı, *FirstOrDefault* ise listede hiçbir eleman yoksa dizi tipinin default değeri (0, null vs) varsa ilk nesneyi döndürür.

Her iki metodun da koşullu ve koşulsuz olmak üzere iki versiyonu vardır. Listedeki ilk ürün bilgisine erişmek için kullanılabilecek sorgu;

Urun u = urunler.First();

Fiyatı 100'den büyük olan ilk ürün bilgisine erişmek içinse aşağıdaki koşullu versiyon kullanılır;

Urun u = urunler.First(u => u.Fiyat > 100);

Ölçüm (Any, All) Operatörlerinin Örnek Kullanımı

Bir koleksiyon içinde herhangi bir elemanın bir koşulu sağlanması isteniyorsa **Any**, tamamının koşulu sağlanması isteniyorsa **All** metodu kullanılır. Kullanımları oldukça basit olan bu iki metodun örnek sorgularının yazılmasını pratik kazanmanız açısından sizlere bırakıyorum.

Diğer Grup Operatörleri (Concat, EqualAll)

İki koleksiyonun elemanları arka arkaya birleştirilmek isteniyorsa **Concat** metodu aşağıdaki gibi kullanılabilir. Bu durumda yeni koleksiyon, iki koleksiyonun arka arkaya eklenmesinden oluşacaktır. Örnek kullanım;

```
int[] sayilar1 = { 4, 2, 1, 2 };
int[] sayilar2 = { 3, 2, 3, 6 };
var sonuc = sayilar1.Concat(sayilar2);
```

EqualAll genişletme metodu ise, iki koleksiyonun bütün elemanlarının karşılıklı olarak birbirine eşit olup olmadığını test eder.

IEnumerable arayüzüne genişletme metotlara ekleyerek yukarıdaki metotlara benzer geliştirmeler yapabiliriz.

Join (Birleştirme) Operatörleri

Birbiriyle ilişkilendirilmiş iki veri kümesi üzerinden (*one to many ya da parent child ilişkileri*) **join** operatörü kullanılarak işlem yapılabilir. LINQ içerisindeki join kullanımı SQL dilindeki join işlemine oldukça benzemektedir.

Aşağıdaki örnekte join operatörü kullanılarak urunler ve kategoriler koleksiyonları birleştirilmiştir. Bu birleşim sonucunda kitapların kategori isimleriyle beraber listelenmesi gösterilmiştir.

```
var sonuclar14 = from u in urunler
        join k in kategoriler
        on u.KategoriId equals k.KategoriId
        select new { u.UrunAdi, k.KategoriId, u.Fiyat };
```

LINQ sorgularını detaylı incelediğimiz bu bölümün de sonuna geldik. LINQ mimarisi içerisinde farklı sorgulama biçimlerinin de olabileceğini belirtmek gerekir. Zira, yukarıdaki her bir sorgu biçiminin kombinasyonlarını kodlayabilir, hatta iç içe sorgular da yazabilirsiniz.

LINQ to SQL ile Veritabanı Sorgulama

Bölümün başlarında söylendiği gibi LINQ mimarisi genişleyebilir bir mimaridir. LINQ ifadeleri arka planda **IQueryable** arayüzü ile uyumlu nesneler üretir. Bu arayüz içinde LINQ ifadelerinin ağaç şeklinde yapısını saklayan Expression nesnesi bulunmaktadır. **IQueryProvider** isimli arayüz yardımıyla ise dinamik olarak sorgu cümlecikleri çalışma zamanında oluşturulabilir. Bu genişleyebilir yapı sayesinde **LINQ To Sql** mimarisi kurulabilmiştir. **LINQ To Sql** yapısında sorgular çalışma zamanında **t-sql** cümleciklerine çevrilerek veritabanı üzerinden direkt olarak sorgulama yapılabilmesine olanak sağlanır. Bu genişleme arayüzleri kullanılarak üçüncü parti firma ya da kişilerde kendilerine özgü LINQ sağlayıcıları geliştirmiştir.

ORM (Object Relation Mapping) Nedir?

LINQ To Sql yapısını genel olarak veritabanı erişimini ilişkisel modelden çıkarıp nesne modeline çeviren ara bir katman gibi düşünebiliriz. LINQ entegrasyonu ile de C# dili içinden sanki t-sql komutlarını çalıştırıyormuşuz gibi kullanabiliriz. Literatürde veritabanı erişiminin bu şekilde nesneler ile otomatik olarak yapılmasına **ORM** (*Object Relation Mapping*) denir. Piyasada ücretsiz ve ücretli dağıtılan birçok ORM aracı zaten mevcuttur. İstenirse mevcut C# 2.0 özellikleri ile de kendi ORM altyapılarımızı geliştirebiliriz. Buna rağmen C# 3.0 ile gelen bu yapının diğer ORM araçlarından farkı direkt olarak dile entegre edilmiş bir **sorgulama dilinin** (*LINQ*) olmasıdır. Diğer ORM araçlarının en büyük dezavantajı sorgulama yapılarının oldukça karşık olmasıdır. C# 3.0 ile bu zorluklar da bertaraf edilmiştir.

Buraya kadar LINQ mimarisini kavradığımız için **LINQ To SQL** yapısı oldukça kolay ve anlaşılır gelecektir.

LINQ To Sql Örnekleri

LINQ To SQL, yapısı ile ilgili sınıflar, nitelikler ve arayüzler **System.Data.Linq** ve **System.Data.Linq.Mapping** isim alanları altındadır.

Sınıfların özellikleri ile tablo kolonları arasındaki ilişkilendirme çeşitli **nitelikler** (*attribute*) yardımıyla yapılır. Bu nitelikler **System.Data.Linq.Mapping** isim alanında bulunur.

System.Data.Linq isim alanında ise veritabanı erişimi yapan **DbContext** nesnesi ve tabloları temsil eden koleksiyon olan **Table** nesnesi bulunur. Örneğimize dönecek olursak; **SQL Server** üzerinde **Northwind** isimli veritabanında aşağıdaki gibi iki tablonun yeraldığını varsayıyoruz.

Kategori

Column Name	Data Type	Allow Nulls
KategoriId	int	☑
KategoriAdi	nvarchar(50)	☑
		☐

Urun

Column Name	Data Type	Allow Nulls
UrunId	int	☐
UrunAdi	nvarchar(50)	☑
Fiyat	decimal(18, 0)	☑
KategoriId	int	☑
SatisBaslangicTarihi	datetime	☑
SatisBitisTarihi	datetime	☑
		☐

KategoriId ve **UrunId** birincil **anahtar** (`primary key`) olarak belirlenmelidir.

Bu iki tablo yaratıldıktan sonra önceki örneklerde bulunan **Urun ve Kategori** bilgilerini tablolara aşağıdaki gibi ekliyoruz.

Results | Messages

1	1	Coca Cola	3	2	2008-03-01 00:00:00.000	2008-04-13 00:00:00.000
2	4	Meyve Suyu	4	2	2006-02-14 00:00:00.000	2007-06-23 00:00:00.000
3	5	Buzdolabı	800	4	2007-01-04 00:00:00.000	2007-07-07 00:00:00.000
4	8	Dell Bilgisayar	1200	3	2008-03-14 00:00:00.000	2008-09-12 00:00:00.000
5	54	Hesap Makinası	300	3	2006-03-01 00:00:00.000	2007-03-28 00:00:00.000
6	32	Koltuk Takımı	3500	6	2008-01-01 00:00:00.000	2008-02-02 00:00:00.000
7	54	T.C. Anayasası	15	1	2005-11-11 00:00:00.000	2005-10-15 00:00:00.000
8	12	Pocket PC	990	3	2005-01-04 00:00:00.000	2006-11-06 00:00:00.000
9	10	XML Kitabı	38	1	2008-01-01 00:00:00.000	2008-04-09 00:00:00.000
10	2	Masa	200	6	2007-12-23 00:00:00.000	2008-01-04 00:00:00.000
11	3	C#	49	1	2005-03-01 00:00:00.000	2007-04-07 00:00:00.000
12	27	VB.NET	25	1	2007-01-01 00:00:00.000	2007-03-12 00:00:00.000

	KategoriId	KategoriAdi
1	1	Kitaplar
2	2	İçecek
3	3	Elektronikl
4	4	Beyaz Eşya
5	6	Mobilya
6	7	DVD Me...

Veritabanı tarafındaki işlemlerimiz bittiğine göre tabloları temsil edecek sınıfları ve veritabanını temsil edecek **DataContext** sınıfından türeyen **Northwind** isimli sınıfı yaratabiliriz. Tablo koleksiyonları **Northwind** sınıfının içerisine koleksiyon şeklinde eklenecektir.

Personel, Urun ve Northwind sınıflarını içeren kod bloğu aşağıdaki gibi bildirilir.

```
using System;
using System.Data;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Collections.Generic;

namespace LINQSQLTest
{
   [Database(Name = "Northwind")]
   public partial class Northwind : DataContext
   {
      public NorthwindDataContext() : base("Data Source=.;
Initial Catalog=Northwind;User ID=sa;Password=1")
      {
      }

      public Table<Urun> Urunler
```

```
    {
      get
      {
        return this.GetTable<Urun>();
      }
    }

    public Table<Kategori> Kategoriler
    {
      get
      {
        return this.GetTable<Kategori>();
      }
    }
  }

[Table(Name = "dbo.Urun")]
public partial class Urun
{

  private int _UrunId;
  private string _UrunAdi;
  private System.Nullable<decimal> _Fiyat;
  private System.Nullable<int> _KategoriId;
  private System.Nullable<System.DateTime> _SatisBaslangicTarihi;
  private System.Nullable<System.DateTime> _SatisBitisTarihi;

  public Urun()
  {
  }

  [Column(Storage = "_UrunId", DbType = "Int NOT NULL", IsPrimaryKey=true)]
  public int UrunId
  {
    get{return this._UrunId;}
    set{ this._UrunId = value;}
  }

  [Column(Storage = "_UrunAdi", DbType = "NVarChar(50)")]
  public string UrunAdi
  {
    get { return this._UrunAdi; }
    set { this._UrunAdi = value; }
  }

  [Column(Storage = "_Fiyat", DbType = "Decimal(18,0)")]
  public System.Nullable<decimal> Fiyat
  {
    get { return this._Fiyat; }
    set { this._Fiyat= value; }
```



```
  }

  [Column(Storage = "_KategoriId", DbType = "Int")]
  public System.Nullable<int> KategoriId
  {
    get { return this._KategoriId; }
    set { this._KategoriId = value; }
  }

  [Column(Storage = "_SatisBaslangicTarihi", DbType = "DateTime")]
  public System.Nullable<System.DateTime> SatisBaslangicTarihi
  {
    get { return this._SatisBaslangicTarihi; }
    set { this._SatisBaslangicTarihi = value; }
  }

  [Column(Storage = "_SatisBitisTarihi", DbType = "DateTime")]
  public System.Nullable<System.DateTime> SatisBitisTarihi
  {
    get { return this._SatisBitisTarihi; }
    set { this._SatisBitisTarihi = value; }
  }
}

[Table(Name = "dbo.Kategori")]
public partial class Kategori
{

  private System.Nullable<int> _KategoriId;
  private string _KategoriAdi;

  public Kategori()
  {
  }

  [Column(Storage = "_KategoriId", DbType="Int NOT NULL", IsPrimaryKey=true)]
  public System.Nullable<int> KategoriId
  {
    get { return this._KategoriId; }
    set { this._KategoriId = value; }
  }

  [Column(Storage = "_KategoriAdi", DbType = "NVarChar(50)")]
  public string KategoriAdi
  {
    get { return this._KategoriAdi; }
    set { this._KategoriAdi = value; }
  }
}
}
```

Yukarıdaki kodların tamamını bir *.cs* dosyasına yazdıktan sonra veritabanından sorgulama işlemi aşağıdaki gibi yapılabilir.

```
LINQSQLTest.Northwind db = new LINQSQLTest.Northwind();
var urunler = from p in db.Urunler
              select p;

foreach (var urun in urunler)
{
   Console.WriteLine(urun.UrunAdi);
}
```

Örnek derlenip çalıştırıldığında ekrana bütün ürünlerin listesi gelecektir.

Yukarıdaki sorgu ifadesinde **foreach** ile iterasyona başlandığı anda **IQueryProvider** nesnesi devreye girerek **LINQ** sorgu ifadesinden **t-sql** sorgu cümleciklerini oluşturur. **Northwind** isimli sınıf ise bu sorguyu veritabanı sisteminde çalıştırılmasını sağlar ve gelen sonuçları *Urun* nesnesinin özelliklerine tek tek atar. Bu atama işlemi yapılırken ise *Urun* sınıfında bulunan **özelliklerin** (*Property*) **Column** isimli niteliğinden faydalanılır.

Northwind nesnesi içinde ise yine **Table** türünden iki adet koleksiyon tanımlanmıştır. Veritabanına bağlantı yapılabilmesi için gereken bağlantı cümleceği ise yapıcı metot içinde tanımlanmıştır. Bağlantıyı **Northwind** nesnesine vermenin birçok yolu mevcuttur. İstenirse harici bir **Connection** nesnesi de **Northwind** sınıfına gönderilebilir.

Örnekte yazılan kod oldukça basit ve anlaşılır olduğu için kod üzerinde çok daha fazla detaya girmeye gerek görmüyorum. Detayları öğrenmenin en iyi olur bu örnek üzerinde değişiklik yaparak kendi örneklerinizi geliştirmenizdir. **LINQ to SQL** sorgu cümleciklerinin çalışma zamanında t-sql sorgularına dönüştüğünü söylemiştik. **Debug** modunda iken sorgu cümleceğine bakıldığında üretilen sorgunun ne olduğu aşağıdaki gibi görülebilir.

```
var personeller = from p in db.Urunler
                  | ⊞ personeller {SELECT [t0].[UrunId], [t0].[UrunAdi], [t0].[Fiyat], [t0].[KategoriId], [t0].[SatisBaslangicTarihi], [t0].[S
foreach (var item in personeller)
{
    Console.WriteLine(item.UrunAdi);
}
```


Kompleks LINQ To Sql Sorguları

LINQ To Object mimarisinde kullanılan hemen hemen bütün genişletme metodu **LINQ To Sql** sorguları içinde de kullanılabilir. Aşağıda farklı şekillerde yazılmış **LINQ To Sql** sorgu örnekleri verilmiştir.

Kısıtlama sorgusu:

```
var urunler = from p in db.Urunler
    where p.Fiyat>100
          select p;
```

Yukarıdaki ifadeden sonra üretilen t-sql sorgusu,

```
SELECT [t0].[UrunId], [t0].[UrunAdi], [t0].[Fiyat], [t0].[KategoriId], [t0].[SatisBaslangicTarihi],
[t0].[SatisBitisTarihi]
FROM [dbo].[Urun] AS [t0]
WHERE [t0].[Fiyat] > @p0
```

Kategori ve *Urun* tablosunun Join ile birleşimi

```
var urunler = from u in db.Urunler
        join k in db.Kategoriler
        on u.KategoriId equals k.KategoriId
        select new { u.UrunAdi, k.KategoriAdi, u.Fiyat };
```

Aynı **db** nesnesi üzerinde bulunan Urunler ve Kategoriler tabloları birleştirilerek iki farklı tablodan veri alınması sağlanmış. Bu sorgunun üreteceği t-sql cümleceği aşağıdaki gibi olacaktır.

```
SELECT [t0].[UrunAdi], [t1].[KategoriAdi], [t0].[Fiyat]
FROM [dbo].[Urun] AS [t0]
INNER JOIN [dbo].[Kategori] AS [t1] ON [t0].[KategoriId] = [t1].[KategoriId]
```

Aşağıdaki kompleks sorguda ise hiçbir ürünü olmayan Kategorilerin listesi sorgulanmıştır.

```
var kategoriler = (from k in db.Kategoriler select k.KategoriId)
      .Except((from u in db.Urunler select u.KategoriId).Distinct());
```

Sorgu çalıştırıldığında hiçbir ürünü olmayan sadece 7 numaralı kategorinin olduğu gözükecektir. Yukarıdaki ifadenin **t-sql** çıktısı ise aşağıdaki gibidir kompleks bir yapı olacaktır.;

```
SELECT DISTINCT [t0].[KategoriId]
FROM [dbo].[Kategori] AS [t0]
WHERE NOT (EXISTS(
   SELECT NULL AS [EMPTY]
```

```
  FROM (
    SELECT DISTINCT [t1].[KategoriId]
    FROM [dbo].[Urun] AS [t1]
    ) AS [t2]
  WHERE (([t0].[KategoriId] IS NULL) AND ([t2].[KategoriId] IS NULL)) OR (([t0].[KategoriId]
IS NOT NULL) AND ([t2].[KategoriId] IS NOT NULL) AND ([t0].[KategoriId] = [t2].
[KategoriId]))
  ))
```

Ürün adında 'A' karakteri geçen ürünlerin sorgusu ise aşağıdaki gibi yazılabilir.

```
var urunler = from u in db.Urunler
        where u.UrunAdi.Contains("A")
        select u.UrunAdi;
```

Üretilen **t-sql** çıktısı ise aşağıdaki gibi olacaktır;

```
SELECT [t0].[UrunAdi]

FROM [dbo].[Urun] AS [t0]
WHERE [t0].[UrunAdi] LIKE @p0
```

String sınıfının **Contains** isimli metodun t-sql olarak "like" deyimine ne şekilde dönüştürüldüğüne dikkat ediniz.

Alınacak ürün verileri üzerinde sayfalama yapmak isterseniz **Take** ve **Skip** metotları kullanılabilir. Örneğin her sayfada 5 kayıt görüntülenen bir ekranda 4. sayfaya konulacak ürünlerin listesi aşağıdaki sorgu ile alınabilir;

```
var urunler  = (from u in db.Urunler select u)
                        .Skip(15)
                        .Take(5);
```

Üretilecek olan **t-sql** sorgusu ise aşağıdaki gibi olacaktır.

```
SELECT [t1].[UrunId], [t1].[UrunAdi], [t1].[Fiyat], [t1].[KategoriId], [t1].
[SatisBaslangicTarihi], [t1].[SatisBitisTarihi]
FROM (
  SELECT ROW_NUMBER() OVER (ORDER BY [t0].[UrunId], [t0].[UrunAdi], [t0].[Fiyat],
[t0].[KategoriId], [t0].[SatisBaslangicTarihi], [t0].[SatisBitisTarihi]) AS [ROW_NUMBER],
[t0].[UrunId], [t0].[UrunAdi], [t0].[Fiyat], [t0].[KategoriId], [t0].[SatisBaslangicTarihi], [t0].
[SatisBitisTarihi]
  FROM [dbo].[Urun] AS [t0]
  ) AS [t1]
WHERE [t1].[ROW_NUMBER] BETWEEN @p0 + 1 AND @p0 + @p1
ORDER BY [t1].[ROW_NUMBER]
}
```


Sıralama için **SQL Server** veritabanı sisteminde bulunan **ROW_NUMBER()** isimli fonksiyonun kullanıldığına dikkat ediniz. **LINQ** ile yazılan çok basit sorguların aslında **t-sql** ile ne kadar kompleks bir hale dönüştüğünü gördükçe **LINQ**'ya olan inancınızın artacağını düşünüyorum. Görüldüğü üzere oluşturulabilecek sorgu cümleceği kombinasyonu oldukça fazladır. Bundan sonrası hayal gücünüze kalmış diyor bu alt başlığı burada sonlandırıyoruz.

LINQ To Sql Diğer Veritabanı İşlemleri (Insert, Update vs)

LINQ To SQL, yukarıda yapıldığı gibi sadece veritabanı verilerini nesneler üzerinden sorgulamak değil aynı zamanda temel veritabanı işlemlerinden kayıt ekleme, güncelleme, silme, saklı yordam çağırma gibi diğer birçok operasyonu da yerine getirmektedir. Her ne kadar **LINQ To SQL**'in en güçlü yani veri sorgulamada olsa bir **ORM**'in olmazsa olmazlarından **CRUD** (*Create*, *Read*, *Update*, *Delete*) işlemlerini yerine getirmesi gerekir.

Aşağıdaki örnekte veritabanına yeni ürün kategorisi eklenmektedir. Görüldüğü üzere tek bir satır **t-sql** cümleciği ya da veritabanı erişim kodu bulunmamaktadır.

```
LINQSQLTest.Northwind db = new LINQSQLTest.Northwind ();

LINQSQLTest.Kategori yeniKategori = new LINQSQLTest.Kategori();
yeniKategori.KategoriId = 9;
yeniKategori.KategoriAdi = "Hediyelik Eşya";

db.Kategoriler.InsertOnSubmit(yeniKategori);

db.SubmitChanges();
```

Yukarıdaki örnek kodda görülen **InsertOnSubmit()** metodu ile yeni bir satırın eklenmesi sağlanıyor. **SubmitChanges()** metodu ile ise yapılan değişiklikler veritabanına doğru gönderiliyor. **SubmitChanges()** metodu çağrımından önce gerçekleşen bütün işlem tek bir **transaction** içerisinde ele alınırlar. Veritabanında bütün ürünlerin fiyatını 5 TL artıracak olan sorgu aşağıdaki gibi yazılabilir.

```
LINQSQLTest.Northwind db = new LINQSQLTest.Northwind ();
var urunler = from u in db.Urunler
        select u;

foreach (var u in urunler)
{
  u.Fiyat += 5;
}

db.SubmitChanges();
```

Veritabanından 8 nolu ürünün silinmesini gerçekleştirecek kod bloğu ise aşağıdaki şekilde yazılabilir:

```
LINQSQLTest.Northwind db = new LINQSQLTest.Northwind ();
LINQSQLTest.Urunsilinecek = (from u in db.Urunler
                 where u.UrunId == 8
                 select u).Single();

db.Urunler.DeleteOnSubmit(silinecek);
db.SubmitChanges();
```

LINQ To Sql, aynı zamanda **saklı yordamlarn** (*stored procedure*) çalıştırılmasınıda **DataContext** nesnesi üzerinen sağlar. Örnek olarak veritabanında aşağıdaki saklı yordamını oluşturalım.

Bu saklı yordam en uzun süreli satışta bulunan *Urun*'ün bilgilerini geri döndürmektedir.

```
create proc EnUzunSatistakiUrun
as

select top 1 * from Urun
order by datediff(d,SatisBaslangicTarihi,SatisBitisTarihi) desc
```

Geri döndürülen sonuç yine Urun sınıfı içine doldurulabilecek formatta olduğu için **Northwind** sınıfı içine aşağıdaki gibi bir fonksiyon eklediğimizde saklı yordamı otomatik olarak çağırıp sonuçlarını *Urun* nesnesi şeklinde alabiliriz.

```
[Function(Name = "dbo.EnUzunSatistakiUrun")]
public ISingleResult<Urun> EnUzunSatistakiUrun()
{
  IExecuteResult result = this.ExecuteMethodCall(this, ((MethodInfo)(MethodInfo.
GetCurrentMethod())));
  return ((ISingleResult<Urun>)(result.ReturnValue));
}
```

Bu fonksiyon aşağıdaki şekilde çağrılarak en çok satışta kalan ürünün adı alınabilir.

```
LINQSQLTest.Northwind db = new LINQSQLTest.Northwind ();
string urunadi = db.EnUzunSatistakiUrun().Single().UrunAdi;
```

Eğer saklı yordamdan dönen sonuçlar Urun sınıfı ile eşleştirilemiyorsa bu durumda yeni bir sınıf yaratılarak eşleşme bu sınıf üzerinden yapılır.

LINQ To SQL nesneleri üzerinden yapılabilecek daha birçok aksiyon bulunmaktadır. Bu kitapta bunların tamamını anlatmak neredeyse imkansızdır. Ana görevlerini

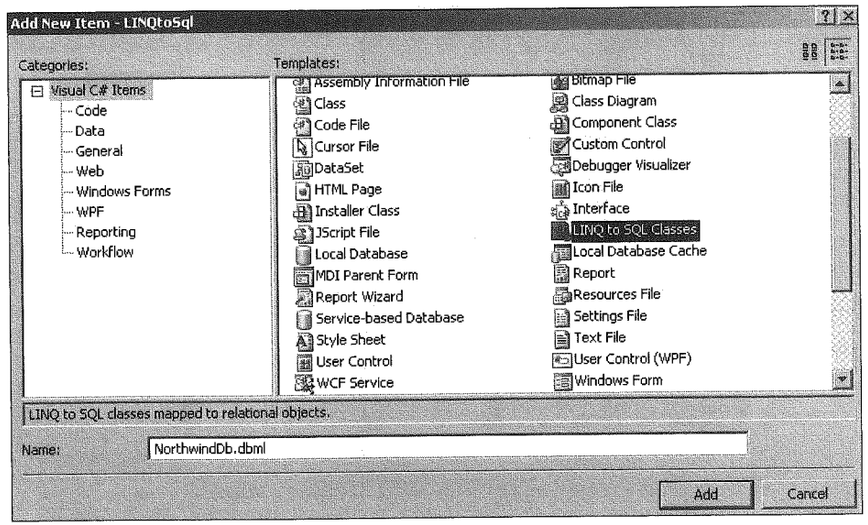
anladıktan sonra diğer kalan görevleri anlamak sizin için çok zor olmayacaktır. Tek yapmanız gereken **DataContext** sınıfının ve **Table<T>** sınıfının üye elemanlarını incelemek ve çeşitli testler gerçekleştirmek olacaktır.

Visual Studio.NET LINQ To Sql Araçları

Bir önceki bölümlerde yaptığımız **LINQ To SQL** sorgularında kullanılan *Urun* ve *Kategori* nesnelerini kendimiz manuel bir şekilde yarattık. Ancak bu nesneler veritabanından çeşitli araçlarla otomatik bir şekilde de üretilebilir. Çünkü yaratılan sınıflara ait kodlar oldukça standart olmakla beraber farklı olan veriler zaten veritabanı tablo ve kolonları üzerinde olan verilerdir. Nitekim son çıkan **Visual Studio.NET** içerisinde **LINQ To Sql** nesnelerini otomatik olarak üreten araçlar bulunmaktadır. Aşağıda bir veritabanında bulunan tablo ve saklı yordamların uygulama ortamında sınıf ve metotlarla nasıl temsil edilebileceği adım adım anlatılmaktadır.

Adım 1

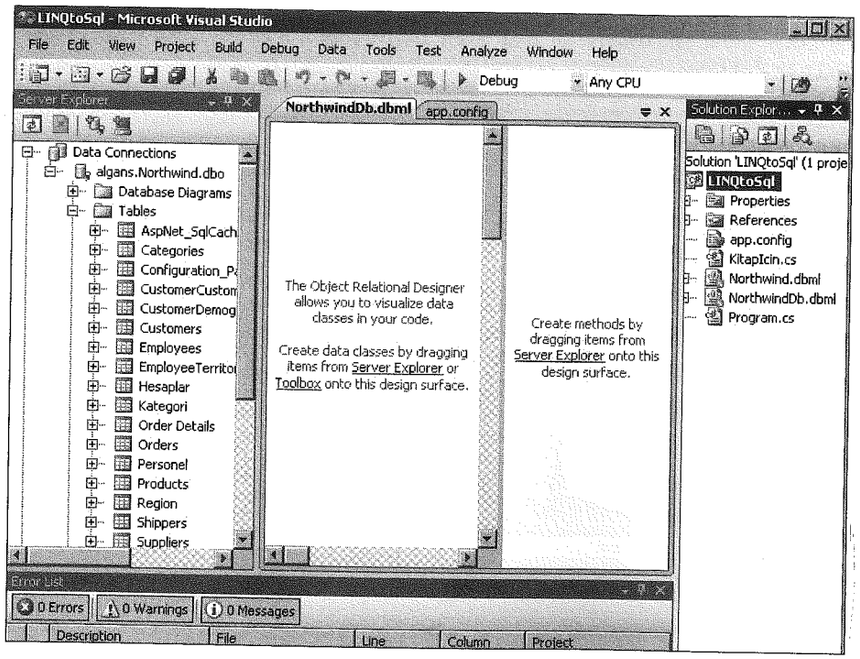
Proje dosyasına sağ tıklayarak **Add New Item** dedikten sonra aşağıdaki ekrandan **LINQ To Sql Object** seçeneği seçilir.



Name alanına **DataContext** nesnesinden türeyecek sınıfımızın yani veritabanını temsil edecek sınıfın adını belirliyoruz. **Add** düğmesine bastığımız anda projemize "NorthwindDB.dbml" dosyasının eklendiğini göreceksiniz. Bu dosyaların içinde **tablo sınıfları** ve **dbcontex** sınıfı yer alacaktır.

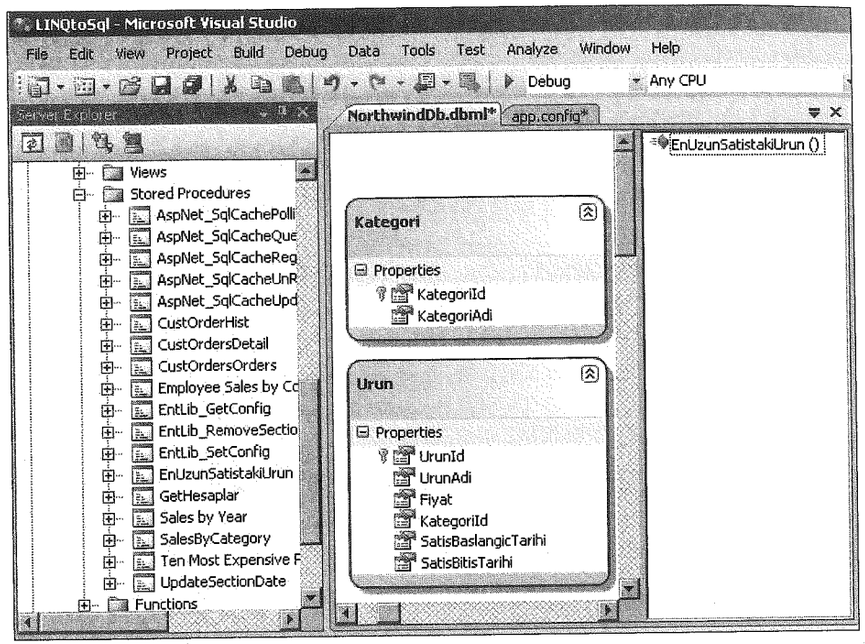
Adım 2

Ardından gelen ekranda 4 bölüm görülecektir. 1. Bölüm'de veritabanına erişmek için gereken **Data Connections** sekmesi bulunur. Eğer burada hiç bir veritabanı bağlantınız yoksa bu bölümdeki menüde buluna "Connect To Database" seçeneği seçilerek veritabanı bağlantısı yapılmalıdır. Ardından burada veritabanında bulunan bütün nesneler listelenecektir. 2. Bölüm ise veritabanı tablolarının sürüklenip bırakılabileceği bir alandır. Buraya Bölüm 1'den herhangi bir tabloyu sürükleyip bıraktığınız anda o tablo ile ilgili sınıflar "NorthwindDb.Designer.cs" dosyasında otomatik olarak oluşturulacaktır. Bölüm 3'te ise saklı yordamların sürüklenip bırakılacağı bir alan vardır. Bu alanda Bölüm 1'den bir saklı yordamı sürükleyip bırakdığınız anda **Data Context** nesnesine daha önce yaptığımız gibi bir metot eklenecektir ki bu metot çağrıldığında arka planda saklı yordam çağrılmaktadır. Bölüm 4 ise standart "Solution Explorer" penceresidir. Sürükleme yaptıkça Bölüm 4'te bulunan **NorthwindDb.dbml** içerisinden ilgili sınıfları inceleyebilirsiniz.



Adım 3

Bölüm1'den *Urun* ve *Kategori* tablolarını ve **EnUzunSatistakiUrun** isimli saklı yordamları ilgili alanlara sürükleyelim. Sürükleme işlemi bittikten sonra aşağıdaki ekran görüntüsü elde edilecektir.



Adım 4

İstenilen tablo ve kolon ile ilgili olan özellikler değiştirilmek isteniyorsa iligili nesneye tıklayarak "Özellik Penceresi"nin açılması sağlanır. Eğer özellik penceresi açılmıyorsa **F4** kısa yol tuşuna basılabilir.

Properties

KategoriAdi Member Property

Access	Public
Auto Generated Value	False
Auto-Sync	Never
Delay Loaded	False
Inheritance Modifier	(None)
Name	**KategoriAdi**
Nullable	**True**
Primary Key	False
Read Only	False
Server Data Type	**NVarChar(50)**
Source	**KategoriAdi**
Time Stamp	False
Type	string (System.String)
Update Check	Always

Properties

Urun Data Class

Access	Public
Delete	Use Runtime
Inheritance Modifier	(None)
Insert	Use Runtime
Name	**Urun**
Source	**dbo.Urun**
Update	Use Runtime

Linq To Sql bölümünün de sonuna geldik. **Linq To Sql** altyapısının burada bahsedilemeyen diğer yeteneklerini keşfetmek isterseniz (*türetme mimarisi, delay load, concurency, transaction yönetimi*) **MSDN**'ye ya da Türkiye'nin en kaliteli C# içeriğine sahip *www.csharpnedir.com* sitesine başvurabilirsiniz.

LINQ To XML Altyapısı

.NET Framework ile gelen varsayılan **LINQ** sağlayıcılarından biri de **LINQ To XML**'dir. **LINQ To XML**, **LINQ** mimarisinin XML verilerden sorgulama ve güncelleme yapmak için uyarlanmış versiyonudur. Bütün LINQ sağlayıcılarının kendilerine has yapıları olmakla beraber temelde LINQ mimarisi olduğu için benzer kullanımları vardır. Bu bakımdan **LINQ To XML** ile ilgili bir iki örnek dışında çok fazla detaya girilmeyecektir. Nitekim **LINQ To SQL**'de geçerli olan birçok sorgulama metotları burada da geçerlidir. **LINQ To XML** ile ilgili sınıfların tamamı **System.Xml.Linq** isim alanında bulunur. Dolayısıyla bu isim alanının koda eklenmesi gerekir.

İlk örnek olarak **LINQ To XML** sınıfları yardımıyla projemize bir XML verisi hazırlayalım. Aşağıdaki örnekte **System.XML.Linq** isim alanında bulunan sınıflar kullanılarak bir XML verisi hazırlanıp dosyaya kaydedilmektedir.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace LINQToXML
{
  class Program
  {
    static void Main(string[] args)
    {
      XElement xml = new XElement("menuler",
          new XElement("menu",
            new XAttribute("menuNo", "1"),
            new XAttribute("fiyat", "15"),
            new XElement("anayemek", "Adana Kebabı"),
            new XElement("corba", "Ezogelin")
          ),
          new XElement("menu",
            new XAttribute("menuNo", "2"),
            new XAttribute("fiyat", "34"),
            new XElement("anayemek", "Karışık Kebap"),
            new XElement("corba", "Domates")
          )
        );
```



```
      xml.Save("Yemekler.xml");
    }

  }
}
```

Program derlenip çalıştırıldığında aşağıdaki gibi bir XML dosyasının hazırlandığı görülecektir.

```
<?xml version="1.0" encoding="utf-8"?>
<menuler>
 <menu menuNo="1" fiyat="15">
  <anayemek>Adana Kebabı</anayemek>
  <corba>Ezogelin</corba>
 </menu>
 <menu menuNo="2" fiyat="34">
  <anayemek>Karışık Kebap</anayemek>
  <corba>Domates</corba>
 </menu>
</menuler>
```

Görüldüğü gibi **System.Xml.Linq** isim alanındaki **XElement** ve **XAttribute** gibi sınıfları yardımıyla XML veriler çok daha hızlı ve kolay bir şekilde hazırlanabilmektedir.

Veritabanımızda bulunan **Urunler** tablosunda bulunan bütün ürünleri alarak aşağıdaki gibi XML verisi oluşturmak ise oldukça basit.

```
<?xml version="1.0" encoding="utf-8"?>
<urunler>
 <urun UrunId="1" KategoriId="2">
  <UrunAdi>Coca Cola</UrunAdi>
  <Fiyat>13</Fiyat>
 </urun>
 <urun UrunId="4" KategoriId="2">
  <UrunAdi>Meyve Suyu</UrunAdi>
  <Fiyat>14</Fiyat>
 </urun>
 <urun UrunId="27" KategoriId="1">
  <UrunAdi>VB.NET</UrunAdi>
  <Fiyat>35</Fiyat>
 </urun>

 ......
 <urunler>
```

XElement ve **LINQ To SQL** sorguları iç içe kullanılarak ilişkisel veri çok hızlı bir şekilde XML verisine dönüştürülebilmektedir.

```
Northwind db = new Northwind();
XElement xmlUrunler = new XElement("urunler",
    from u in db.Urunler
    orderby u.Fiyat
    select new XElement("urun",
        new XAttribute("UrunId", u.UrunId),
        new XAttribute("KategoriId", u.KategoriId),
        new XElement("UrunAdi", u.UrunAdi),
        new XElement("Fiyat", u.Fiyat))
    );

xmlUrunler.Save("Urunler.xml");
```

XML veri üzerinden sorgulama yapmak da oldukça basit bir işlemdir. Ancak XML varlıklarına ilişkin sınıflar yaratılmadığı için LINQ sorgularında **XElement** ve **XDocument** sınıfının bir takım metotları kullanılmak zorundadır. Örneğin bir xml veri içindeki bütün "Urun" isimli düğümleri elde etmek isterseniz **XDocument** sınıfının **Descendants** isimli metodunu kullanmalısınız. Aşağıdaki örnekte daha önce oluşturduğumuz **Urunler.xml** dosyasından ürün fiyatı *100*'den büyük olan ürünlerin bilgisi elde ediliyor.

```
XDocument doc = XDocument.Load("Urunler.xml");
var sonuclar = from u in doc.Descendants("urun")
        where (int)u.Element("Fiyat") > 100
        select new Urun {
            UrunId = (int)u.Attribute("UrunId"),
            UrunAdi=(string)u.Element("UrunAdi")
            };

foreach (var urun in sonuclar)
{
  Console.WriteLine(urun.UrunAdi);
}
```

Mevcut bir XML dosyasını belleğe yüklemek için **System.Xml.Linq** isim alanında bulunan **XDocument** sınıfı kullanılır.

Yukarıdaki örnekten de görüldüğü üzere düğüm ve nitelik değerlerine **Linq To Sql**'den farklı olarak **XElement** sınıfın *Attribute* ve *Element* isimli metotları ile erişilmiştir.

Linq To Xml nesneleri kullanılarak veri üzerinde güncelleme yapmak da oldukça kolaydır. Veri güncelleme, silme ve ekleme işlemleri için de yine **XElement** sınıfının **Add, ReplaceAttribute, SetAttributeValue, Remove** gibi metotlar kullanılır.

Aşağıdaki örnekte xml dosyasında bulunan ürünlerden ürün kategori numarası 2 olan ürünlerin tamamının fiyatı 10 TL artırılıp kaydediliyor.

```
XDocument doc = XDocument.Load("Urunler.xml");
var guncelle = from u in doc.Descendants("urun")
        where (int)u.Attribute("KategoriId") == 2
        select u;

foreach (var urun in guncelle)
{
  int fiyat = (int)urun.Element("Fiyat");
  urun.SetElementValue("Fiyat", fiyat + 10);
}

doc.Save("Urunler.xml");
```

XElement sınfının daha birçok metodu vardur. İhtiyacınıza göre bu metotları veri güncelleme, silme ve ekleme amaçlı kullanabilirsiniz.

İfade Ağaçları (Expression Trees)

LINQ mimarisinin genişleyebilir olmasının altında yatan şey aslında **.NET Framework 3.5** ile birlikte gelen ifade ağaçlarıdır. İfade ağaçları yani expression tree denilen yapı sayesinde çalıştırılabilir kod ağaç yapısı şeklinde veri olarak bellekte saklanabiliyor. Bu sayede **LINQ sorgu ifadeleri** derleme zamanında veri olarak ifade edilebilmektedir. Dolayısıyla çalışma zamanında bu ağaç yapısı çözümlenerek üretilmesi gereken kod (*sql, xpath* ya da *metot çağrımları*) belirlenir. Kısaca LINQ mimarisinin genişleyebilir yapıda sunulabilmesini sağlayan ifade ağaçlarıdır.

IQueryable arayüzü içinde **Expression** isimli bir üye eleman bulunmaktadır. Buradan LINQ ile sorgulanabilen her veri sağlayıcının (`linq to sql`, `linq to xml` vs) **Expression** yapısını uygulaması gerektiği çıkarılabilir.

İfade ağaçlarına somut bir örnek vermek gerekirse;

İfade ağaçlarını temsil eden **Expression** isimli sınıf **System.Linq.Expressions** isim alanında bulunur.

Parametre olarak bir int değişken alan ve bu değişkenin 5 fazlasını geri döndüren fonksiyon verisi aşağıdaki şekillerde ya kod yapısı ile ya da veri yapısı şeklinde tanımlanabilir;

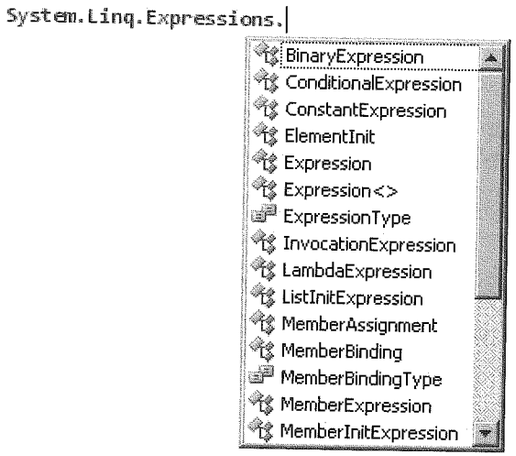
Kod yapısı şeklinde;

Func<int,int> exp = x => x + 5;

Veri yapısı şeklinde;

Expression<Func<int,int>> exp = x => x + 5;

Yukarıda veri yapısı şeklinde bildirilen ifade aslında **System.Linq.Expressions** isim alanı altında bulunan aşağıdaki sınıflar yardımıyla ağaç yapısına dönüştürülür.



Ağaç yapısında kullanılan çeşitli ifade sınıfları.

Ertelenmiş Çalışma (Deferred Execution)

LINQ ile birlikte gelen **ağaç yapısı** (*expression tree*) sayesinde LINQ sorguları kod içerisinde geçtiği noktada icra edilmezler. LINQ sorgularından geri döndürülen **IQueryable** arayüzüne sahip nesne kullanıldığı ilk noktada çalıştırılır. İlk kullanımdan kasıt bir iterasyon ya da bir metot çağrımı olabilir. Bu duruma ertelenmiş çalışma (*Deferred Execution*) denilmektedir.

Bunun gerçekten bu şekilde olduğunu görmek için aşağıdaki örnek **Linq To Sql** sorgusunu çalıştırabilirsiniz. Bu LINQ sorgusunda kısıt olarak verilen veri lokal bir değişken üzerinden alınmaktadır. LINQ sorgusunun hemen ardından bu lokal değişken değiştiriliyor. Ertelenmiş çalışma mekanizmasına göre sorgu ifadesinin ilk kullanıldığı nokta olan **foreach** iterasyonunda sorgulama işlemi gerçek manada çalıştırılacağı için koşul ifadesinde geçen "UrunId" değerine lokal değişkenin tam o noktadaki değeri atanmaktadır. Eğer ertelenmiş çalışma olmasaydı, koşuldaki *UrunId 5* yerine *1* değerini alacaktı.

```
int UrunId = 1;
var urunler = from u in db.Urunler
```



```
        where u.UrunId = UrunId
        select u;

UrunId = 5;

foreach (var urun in urunler)
{

}
```

Yukarıdaki örnek çalıştırıldığında *UrunId* değeri *5* olan ürünlerin getirildiği görülecektir.

Eğer ertelenmiş çalışma mekanizması devre dışı bırakılmak isteniyorsa LINQ sorgusunun sonucunda dönen koleksiyon üzerinden **ToList()**, **ToDictionary()** gibi metotlar çağrılarak ifade ağacının çalıştırılması sağlanır. Bu duruma örnek kullanım aşağıda verilmiştir;

```
int UrunId = 1;
var urunler = (from u in db.Urunler
        where u.UrunId = UrunId
        select u).ToList();
UrunId = 5;

foreach (var urun in urunler)
{
}
```

Yukarıdaki örnekte ise *UrunId* değeri *1* olan ürünlerin listelendiği görülecektir. Çünkü sorgu ifadesi tanımlandığı satırda *ToList()* fonksiyonu yardımıyla LINQ sorgusu icra ediliyor.

C# 5.0 ile Gelen Asenkron Programlama Özellikleri

- C# 5.0 Yenilikleri
- C# 5.0 ve Asenkron Programlama
- Async ve Await Anahtar Kelimeleri
- Asenkron Programlama ve Multi-Thread Programlama
- Caller Information Nitelikleri

C# 5.0 Yenilikleri

Microsoft, C# 1.0 sürümünden sonra yayınlanan her sürümündede belli konu ve ihtiyaçlara yönelik geliştirmeler yapmıştır. C# 2.0 sürümüyle gelen Şablon Tipler (Generics) sayesinde kod tekrarının azaltılıp, daha az kod yazılarak daha çok iş yapılmasına imkân sağlanmış, Null Değer Alabilen Tipler (Nullable Value Types) sayesinde, C# tiplerinin Veri Tabanı Tipleri ile olan uyumu arttırılmıştır. C# 3.0 sürümüyle gelen Dile Entegre Edilmiş Sorgu (LINQ) yapısıyla beraber, farklı veri kaynaklarıyla ilgili işlemler oldukça kolaylaştırılmıştır. C# 4.0 ile, dile dinamik programlama yeteneği kazandırılmıştır. Bu konuları bu kitabın değişik bölümlerinde ele almıştık.

Bu kitabın konusu olan C# programlama dilinin 2013 yılı itibariyle son sürümü 5.0'dır. Bu bölümde 5.0 sürümü ile gelen yenilikler anlatılmaktadır.

Günümüzde kurumsal uygulamaların birbirleriyle olan entegrasyon ihtiyaçları giderek artmaktadır. Özellikle e-ticaret ve bankacılık sektöründe bu ihtiyacın daha yo-

ğun olduğunu söyleyebiliriz. Kurumsal uygulamalarda Servis Yönelimli Mimarilerin (SOA) uygulandığını artık daha sık görüyoruz. Bu entegrasyon ihtiyaçlarından dolayı sistemler artık kendi kontrolünde olmayan sistemlerle daha sık haberleşmek durumunda kalıyorlar. Sistemlerinin birbirlerinden yanıt beklemelerinin uygulamalarda ve kullanıcı arayüzlerinde donma gibi kullanıcı tecrübesini ve sistemi olumsuz etkileyen sonuçlarından dolayı bu tip mimarilerde geleneksel eş zamanlı (synchronized) programlama yöntemleri yerlerini eş zamanlı olmayan asenkron (asynchronized) programlama yöntemlerine bırakmış durumdadırlar.

.Net Framework uzun süredir eş zamanlı olmayan programlama modelini desteklemekte fakat bu yapıların aşırı kullanımı hem kod okunurluğunu azaltmakta hem de uygulamaların bakım maliyetlerini arttırmaktadır.

Microsoft, asenkron programlamaya olan ihtiyacın arttığını görerek, C#'ın yeni sürümünde asenkron işlemlerin daha az eforla yapılabilmesine olanak sağlayan özellikler yayımlamıştır.

Kitabın bu bölümünde C# diline kazandırılan eş zamanlı olmayan programlamayla ilgili yenilik ve *Caller Information* nitelikleri anlatılacaktır.

C# 5.0 ve Asenkron Programlama

.Net Framework, 1.0 sürümünden bu yana eş zamanlı ya da eş zamanlı olmayan programlama modelini desteklemekteydi. C# 5.0 sürümünden önce **IAsyncResult** arayüzünden (Interface) türeyen sınıflar, bize bu konuda yardımcı oluyorlardı. Bu arayüzden türemiş sınıfların tipik olarak Begin ve End ön ekleriyle başlayan metotları vardır. Örneğin bize dosya okuma yazma işlemlerinde yardımcı olan **FileStream** sınıfının BeginRead ve EndRead metotları bize asenkron işlemlerde yardımcı olurlar.

Fakat bu tip asenkron kullanımların çeşitli dezavantajları vardır. Begin ön ekiyle başlayan metotlar, asenkron işlemin tamamlandığını bildirmek için bizden **AsyncCallback** tipinde bir temsilci (delegate) isterler, işlem tamamlandığında yapılmasını istediğimiz işlemleri temsilcinin işaret ettiği metot içerisinde yaparız. Bu şekilde bir kullanım, kodun okunurluğunu azaltır, **çünkü yapmak istediğimiz işlemi ikiye böler, iki farklı yerde yaparız.** Oysa ki yapılmak istenilen işlem mantıksal olarak birbirini izlemektedir. Ayrıca, Begin ön ekiyle başlayan metotları çağırdığımız zaman uygulama bu asenkron işlem için ayrı bir **Thread** başlatır. Ön yüz işlemleriyle uğraştığımız uygulamalarda, ön yüz tarafındaki güncelleme işlemlerinin ön yüzün **Thread'i** içerisinde gerçekleşmesi gerekmektedir. Bu sorunun üstesinden gelmek için **SynchronizationContext** nesnesinden yararlanabiliriz fakat bu nesnenin kullanımı da kod içerisindeki karmaşıklığı arttırır.

Yukarıda bahsedilen Thread, SynchronizationContext, AsyncCallback gibi konular bu kitabın kapsamı dışındadır. Bu konularla ilgili daha detaylı bilgiyi MSDN kaynaklarından edinebilirsiniz.

C# 5.0 sürümüyle beraber eş zamanlı olmayan işlemleri daha sade ve anlaşılır şekilde yapabiliyoruz. Yeni yapının kalbinde ***async*** ve ***await*** anahtar kelimeleri bulunmaktadır. Bu iki kelime, eş zamanlı metot kolaylığında ve anlaşılırlığında eş zamanlı olmayan metotlar yazmamıza olanak sağlamaktadır.

async ve ***await*** anahtar kelimelerini sadece .Net 4.0 sürümüyle beraber kullanabilmekteyiz. Çünkü bu iki anahtar kelime arka planda .Net 4.0 ile Framework'e dahil olan ***Task*** alt yapısını kullanmaktadır.

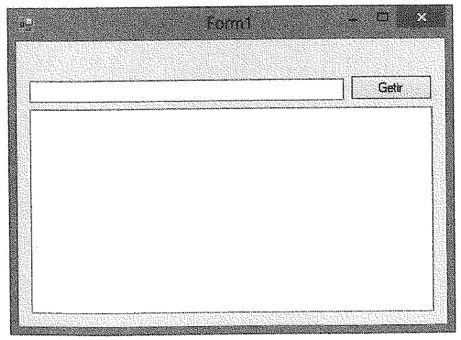
async ve await Anahtar Kelimeleri

async ve await anahtar kelimeleri aslında C# derleyicisine, koda nerelerde müdahale edileceğinin bilgisini vermektedir. Derleyici, bu iki anahtar kelimenin kullanıldığı yerlerde, derleme zamanında müdahale ederek kodu asenkron bir yapıya dönüştürmektedir.

async anahtar kelimesi, derleyiciye metot içerisinde ***await*** anahtar kelimesinin kullanılmak istendiğini belirtir. ***async*** anahtar kelimesinin kullanıldığı metodun içerisinde ***await*** anahtar kelimesi kullanılmadığı sürece ***async***, metodun davranışında bir değişikliğe sebep olmayacaktır. Bir metodun async olarak işaretlenebilmesi için geri dönüş tipi mutlaka ***void***, ***Task*** veya ***Task<TResult>*** olmalıdır. Ayrıca ***async*** ile işaretlenen metotlara, ***ref*** veya ***out*** anahtar kelimelerini kullanarak parametre geçemeyiz.

await anahtar kelimesi derleyiciye kullanıldığı yerde asenkron bir işlem gerçekleşeceğini bildirir. ***await***'den sonra kullanılacak metodun geri dönüş tipi mutlaka ***Task*** veya ***Task<TResult>*** tipinden olmalıdır. **await** anahtar kelimesi, **async** ile işaretlenmiş metotlar dışında kullanılamaz.

await ve ***async*** kelimelerinin nasıl kullanıldığını daha iyi anlamak için hem eş zamanlı hem de eş zamanlı olmayan paralel senaryoyu içeren iki örnek yapalım.



Yukarıdaki Windows Form uygulaması, önyüzde yer alan bir butona tıklandığında metin kutusu içerisindeki adrese HttpWebRequest nesnesini kullanarak bir istekte bulunmaktadır. Gelen web sayfası içeriğini ise metin olarak okuyup, başka bir metin kutusu üzerinde göstermektedir.

Bu işlemi önce eş zamanlı yöntemle yapalım.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void btnGetir_Click(object sender, EventArgs e)
    {
        string url = txtUrl.Text;

        HttpWebRequest webRequest = (HttpWebRequest) WebRe-
quest.Create(url);

        WebResponse webResponse = webRequest.GetResponse();

        using (StreamReader sr = new
StreamReader(webResponse.GetResponseStream()))
        {
            txtIcerik.Text = sr.ReadToEnd();
        }
    }
}
```

Yukarıdaki kod bloğunu incelediğimiz zaman ***webRequest.GetResponse*** metodu üzerinden senkron bir çağrım yapıldığını görüyoruz. İdeal şartlarda bu uygulamada kullanıcı tecrübesini etkileyecek kadar bir gecikme olmayacaktır. Fakat, istek gönderdiğimiz adreste yaşanacak yoğunluk, bizim uygulamamızın çalışmasını doğrudan etkileyecektir. Yukarıdaki kod bloğundaki temel problem, önyüz kodları ve bizim yazdığımız kodlar aynı Thread (İşlem Parçacığı) içerisinde çalışmaktadır, dolayısıyla GetResponse metodundan cevap gelene kadar arayüz ile ilgili işler bekletilecektir, çünkü Thread metottan gelecek olan cevabı beklemektedir. GetResponse gibi metotlar **Blocker** metotlar olarak da tanımlanmaktadırlar. Çünkü kodun ilerleyişini bloklamaktadırlar. Bunun bir sonucu olarak, ekranda donmalar yaşanabilir hatta işlemin ne kadar sürdüğüyle bağlantılı olarak Windows uygulamayı "Not Responding (Uygulama Yanıt Veremiyor)" durumuna çekebilir.

Bu durum elbette ki kullanıcı tecrübesi açısından istenmeyen bir durumdur. Ayrıca aynı kod bloğu içerisinde, konudan bağımsız başka işleri yürütecek olan metotların da işlerini yapmasını engelleyecektir. Böyle bir durumdan sistemimizin sağlıklı çalışması da etkilenebilir.

Yukarıdaki işlemi ***async*** ve ***await*** anahtar sözcüklerini kullanarak güncelleyelim.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private async void btnGetir_Click(object sender, Even-
tArgs e)
    {
        string url = txtUrl.Text;

        HttpWebRequest webRequest = (HttpWebRequest) WebRe-
quest.Create(url);

        WebResponse webResponse = await webRequest.GetRe-
sponseAsync();

        using (StreamReader sr = new
StreamReader(webResponse.GetResponseStream()))
        {
            txtIcerik.Text = sr.ReadToEnd();
        }
    }
}
```

Yukarıdaki kod bloğu incelendiğinde ilk fark edilen btnGetir_Click metodunun async ile işaretlenmiş olduğudur. İkinci fark edilen ise artık webRequest nesnesinden önce ***await*** kullanıldığı ve GetResponse metodu yerine ***GetResponseAsync*** çağrıldığıdır. ***await*** kelimesinin, ancak geri dönüş tipi Task veya Task<Result> olan metotlarla kullanılabileceğinden bahsetmiştik. "Async" son ekiyle biten metotlar .Net 4.5 ile beraber **IAsyncResult** arayüzünden türemiş sınıflara gelmiştir. Bu metotların en önemli amacı hem **IAsyncResult** ara yüzünden türemiş nesnelerin .Net 4.0 ile gelen Task altyapısıyla çalışabilmesini hem de geriye uyumluluğu sağlamaktır. Yukarıdaki kod bloğunu eş zamanlı örneğimizle karşılaştırdığınız zaman çok az bir değişiklik yapıldığını fark edeceksiniz. GetResponseAsync metodu ***await*** anahtar sözcüğü ile kullanıldığı için derleme zamanında asenkron çalışacak şekilde yeniden düzenlenecektir.

Peki, yukarıdaki kod bloğu çalıştığı zaman neler yaşanmaktadır? Uygulama çalışma esnasında ***await*** anahtar kelimesinin kullanıldığı GetResponseAsync metoduna geldiği anda GetResponseAsync işini bitirene kadar uygulama arayüz iş parçacığı (UI Thread) btnGetir_Click metodundan çıkar. Bu süre zarfı içerisinde arayüz iş parçacığı (UI Thread) kullanımda olmadığından dolayı kullanıcı uygulamanın diğer fonksiyonlarını sorunsuz bir şekilde kullanmaya devam edebilir. GetResponseAsync metodu işini tamamladıktan sonra, btnGetir_Click metodu kaldığı yerden çalışmaya devam edecektir.

Asenkron Programlama ve Multi-Thread Programlama

Genelde asenkron programlamayla, multi-thread programlama aynı şey olarak düşünülür. Bu her zaman doğru değildir. Mesela HTTP kullanarak veri çekmeyle ilgili bir network operasyonunu ele alalım. Sık sık bu aşamada arka planda ayrı bir thread açıldığı ve veri çekme operasyonu süresince bu thread'in çalıştığı düşünülür. Aslında bu işlem sırasında sadece belli noktalarda thread'in devreye girmesi gerekmektedir ve teknik olarak thread'i bloklamadan bu işin aynı thread içerisinde yapılması mümkündür. Öncelikle data çekme isteğini gönderdiğimiz andan itibaren zamanın büyük kısmı sunucudan gelecek olan cevabı (response) beklemekle geçer, sunucu tarafından cevap gelene kadar herhangi bir şey yapamayız. Dolayısıyla thread'in yapacağı herhangi bir iş yoktur. Zaten bundan dolayı, bu işlemi senkron programlama yöntemiyle yapmaya kalkıştığımızda sistem, thread'i uyku (sleep) durumuna geçirir, çünkü thread'in yapacağı herhangi bir iş yoktur. Asenkron programlamada bilgisayarın birden çok thread'e ihtiyaç duymamasının bir diğer sebebi de, bilgisayarın network donanımının bu işin çoğunu tek başına yapabilecek kapasitede olmasıdır. Veri çekmek için istekte bulunduğumuzda network sürücüsü (driver), network donanımına bu işi yapması için gerekli komutları verir. Bilgisayarın işlemcisi sadece bu işi yaparken meşgul olur ki, bu işi yapmakta gelecek olan cevabı beklemekten çok daha az vakit alır. Aynı şekilde Network donanımı, sunucu tarafından cevabı aldığında, tekrar işlemci devreye girer ama yine burada harcanan süre çok azdır. Yani işin büyük kısmını network donanımı halleder. Dolayısıyla birden fazla thread'in çalışmasına gerek yoktur. Çoğu I/O işlemleri bu şekilde gerçekleşir. Aynı durumu sabit diskten veri okuma, yazma işlemleri için de uyarlayabiliriz. Bu aşamada işlemci, disk sürücüsüne neyin nereye yazılacağını söyler. Zamanın büyük kısmı sabit diskin çalışıp verileri yazması esnasında geçer.

Önemli Not: C# 5.0 ile gelen await ve async anahtar kelimelerinin daha verimli bir şekilde anlaşılabilmesi için bu kitabın kapsamında olmayan Thread, Asenkron Programlama gibi konuların detaylı olarak incelenmesi ve öğrenilmesi gerekir.

Caller Information Nitelikleri

Bazen uygulamalarımızda kimin hangi metodu veya özelliği (property) çağırdığı bilgisine ihtiyaç duyarız. Bu bilgiler özellikle büyük uygulamalarda, uygulamanın bakım yapılabilirliği (**maintainability**) açısından çok önemlidir. Çağıran Bilgileri (Caller Information), uygulamamızdaki bug'ların tespiti, hataların kaydedilmesi, performans

ölçümü, debug gibi konular için bazen hayati önem taşır. Bu bilgileri elde etmek için C# 5.0 sürümünde dile kazandırılan bir diğer yetenek olan **Callar Information** niteliklerinden yararlanılabilir. Nitelikler (Attribute) ile ilgili detaylı bilgi bu kitabın geçmiş bölümlerinde bulunmaktadır.

C# 5.0 sürümünden önce de bu bilgileri, **System.Diagnostic** isim alanı içerisinde bulunan **StackTrace** ve **StackFrame** sınıflarını kullanarak **çalışma-zamanında** (runtime) elde edebiliyorduk. Fakat bu iki sınıf da **call stack** (metot çağrım yığını) ile ilgili bilgilere ulaştıkları için performans sorunlarına sebep oluyorlar. Call stack ile ilgili detaylı bilgilere ulaşmak açısından bu iki sınıfı kullanmak hala en geçerli yoldur. Fakat bazen, sadece çağıran bilgilerine ihtiyaç duyarız. Caller Information nitelikleri bu bilgilere ulaşmak için kolay ve daha okunaklı bir alternatif sağlıyor.

Caller Information nitelikleri, metotların sadece opsiyonel parametrelerinde kullanılabilirler. Bu işlemler için üç adet nitelik vardır. Bunlar ;

- [CallerMemberName]: Çağrımı yapan metot veya özellik hakkında bilgi verir.
- [CallerFilePath]: Çağrımı yapanın kaynak kodunu içeren dosyanın dosya yolunu verir.
- [CallerLineNumber]: Çağrım yapan kodun satır numarasını verir.

Bu nitelikler kullanıldığı zaman C# derleyicisi, **derleme-zamanında** (compile-time) koda müdahale ediyor ve bu niteliklerin kullanıldığı metodun çağrıldığı yerde parametrelere **ilgili değerleri** belirliyor. Dolayısıyla bu niteliklerin kullanımında herhangi bir performans sıkıntısı yaşamıyoruz.

Kullanımıyla ilgili bir örnek yapalım;

```
public class CallerInformationOrnegi
{
    public static void Main(string[] args)
    {
        CagiranBilgileriniGoster();
    }
    public static void CagiranBilgileriniGoster (
      [CallerMemberName] string cagiranIsmi = null,
      [CallerFilePath] string cagiranDosyaYolu = null,
      [CallerLineNumber] int cagiranSatirNumarasi = 0)
    {
      Console.WriteLine("Çağıran İsmi: {0}", cagiranIsmi);
      Console.WriteLine("Çağıran Dosya Yolu: {0}", cagiranDosyaYolu);
        Console.WriteLine("Çağıran Satır Numarası: {0}", cagi-
ranSatirNumarasi);
    }
}
```

Yukarıdaki kod bloğunu çalıştırdığımız zaman aşağıdaki sonucu elde ederiz.

Çağıran İsmi: Main
Çağıran Dosya Yolu: c:\Users\UserBilgisi\Documents\Visual Studio 2012\Projects\ConsoleApplication1\ConsoleApplication1\Program.cs
Çağıran Satır Numarası: 5

Bu niteliklerden CallerMemberName niteliği, INotifyPropertyChanged arayüzünün implementasyonunu oldukça kolaylaştırıyor. INotifyPropertyChanged arayüzü, WPF ve Windows Forms gibi teknolojilerin **binding** alt yapısı tarafından kullanılır. Bu konular kitabın kapsamı dışındadır ancak CallerMemberName kullanımına güzel bir örnek teşkil etmektedir. INotifyPropertyChanged arayüzünden türettiğimiz nesneler, üzerlerinde bir değişiklik olduğu zaman değişiklikleri kullanıcı ön yüzüne bildirirler.

Aşağıdaki kullanım CallerMemberName olmaksızın yapılan bir örnektir;

```
public class Kullanici : INotifyPropertyChanged
{
    private string _isim;
    private int _yas;

    public event PropertyChangedEventHandler PropertyChanged;

    public string Isim
    {
        get { return _isim; }
        set
        {
            _isim = value;
            OnPropertyChanged("Isim");
        }
    }

    public int Yas
    {
        get { return _yas; }
        set
        {
            _yas = value;
            OnPropertyChanged("Yas");
        }
    }

    protected virtual void OnPropertyChanged(string propertyName)
    {
```



```
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
        {
        handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

INotifyPropertyChanged arayüzü kullanılırken en büyük risk, özellikler (property) içerisinde, özellik ismini OnPropertyChanged metoduna elle parametre olarak geçmek zorunda olmamızdan kaynaklanıyor. Çünkü özellik isminin yanlış yazılması veya özellik isminin değiştirilip parametre olarak geçilen ismin değiştirilmemesi gibi durumlar söz konusu olabiliyor. Bu riskten kaçınmak için CallerMemberName niteliğinden faydalanabiliriz.

```
public class Kullanici : INotifyPropertyChanged
{
    private string _isim;
    private int _yas;

    public event PropertyChangedEventHandler PropertyChanged;

    public string Isim
    {
        get { return _isim; }
        set
        {
            _isim = value;
            OnPropertyChanged();
        }
    }

    public int Yas
    {
        get { return _yas; }
        set
        {
            _yas = value;
            OnPropertyChanged();
        }
    }

    protected virtual void OnPropertyChanged([CallerMemberName]
string propertyName = null)
    {
```

```
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyN
ame));
        }
    }
}
```

C# 6.0 ile ilgili Güncelleme Notu

C# 5.0 ilk çıktığında await kullanımını catch bloklarında yapamıyorduk. C# dilinin tasarımcısı bu durumu "bunu o zaman yapabileceğimize inanmıyorduk" şeklinde açıklamıştı. Ancak C# 6.0 ile birlikte zaten olması gereken bir durum yapılmış oldu. Buna göre herhangi bir catch bloğunda da normal bloklardaki gibi await kullanabilmekteyiz.

Cacth bloğunda örnek bir await kullanımı aşağıdaki şekildedir.

```
public async void DeleteAll()
{
    try
    {

    }
    catch (Exception exc)
    {
        await new Task(() => LogDb());
    }
}

private void LogDb()
{
    //Db loglama işlemleri
}
```

Daha önce de dediğimiz gibi async kullanabilmemiz için ilgili fonksiyonun void olması ya da Task gibi tipler geri döndürmesi gerekirdi.

Bu konunun sonuna dolayısıyla da kitabın sonuna gelmiş bulunmaktayız. Kitaptaki bütün uygulamaları yapmış olduğunuzu düşünürsek C# dilini öğrendiniz demektir. C# dilinin yanı sıra .NET sınıf kütüphanesini de ana hatlarıyla öğrendiniz. Kodlama yaparken bu kitabın size önemli bir referans kaynağı olacağı kanaatindeyim. Kitapta değinilmeyen konular hakkında yazmış olduğum çeşitli makaleleri www.csharpnedir.com internet adresinden okuyabilirsiniz.

Kaynakça

1. C# Primer: A Practical Approach, Stanley B. Lippman (Addison Wesley Professional)
2. Thinking in C#, Burce Eckel, Larry O'Brien (Prentice Hall PTR)
3. C# and The .NET Platform, Andrew Troelsen (APress)
4. Inside C#, Tom Archer, Andrew Whitechapel (Microsoft Press)
5. C# Programming Tips & Techniques, Charles Wright (McGraw-Hill Osborne Media)
6. A'dan Z'ye C Klavuzu, Kaan Aslan (Pusula Yayıncılık)
7. XML, Zafer Demirkol (Pusula Yayıncılık)
8. Microsoft MSDN Yardım Dokümanları
9. *www.csharpnedir.com*
10. *www.csharphelp.com*
11. *www.microsoft.com*
12. *msdn.microsoft.com*

Dizin

Semboller

.NET 7-20, 27, 30, 38, 66, 127, 165, 173, 174, 207, 253, 254, 260, 261, 267, 271, 273, 274, 285, 286, 290, 350, 352, 387, 391, 405, 406, 407, 408, 410, 411, 414, 415, 422, 427, 441, 446, 454, 455, 464, 486, 487, 488, 500, 505, 510, 511, 514, 515, 522, 534, 544, 546, 559, 589, 591, 592, 597, 598, 617, 618, 619, 629, 632, 633, 634, 636, 638, 639, 641, 645, 647, 648, 650, 655, 660, 676, 677, 679, 681, 682, 684, 686, 688, 696, 697, 704, 706, 708, 744, 747, 760, 761
.NET sınıf kütüphanesi 27, 261, 267, 655, 679
.NET ve olaylar 27, 261, 267, 655, 679
:: operatörü 27, 261, 267, 655, 679

A

Add 563, 565, 566, 569, 570, 574, 605, 614, 616, 640, 644, 645, 646, 654, 657, 658, 667, 668, 676, 741, 747
ADO 563, 565, 566, 569, 570, 574, 605, 614, 616, 640, 644, 645, 646, 654, 657, 658, 667, 668, 676, 741, 747
ADO.NET 563, 565, 566, 569, 570, 574, 605, 614, 616, 640, 644, 645, 646, 654, 657, 658, 667, 668, 676, 741, 747
ADO.NET nesne modeli 563, 565, 566, 569, 570, 574, 605, 614, 616, 640, 644, 645, 646, 654, 657, 658, 667, 668, 676, 741, 747
akış kontrol mekanizmaları 563, 565, 566, 569, 570, 574, 605, 614, 616, 640, 644, 645, 646, 654, 657, 658, 667, 668, 676, 741, 747
Alias 253, 264, 265, 269
anahtar sözcük 50, 253, 264, 265, 269
Anonymous Methods 405, 440
Anonymous Types 681, 695
application domain 682
Arama İşlemleri 291, 328
arayüz 38, 138, 194, 260, 347, 381, 382, 391, 392, 396, 397, 398, 399, 401, 505, 516, 534, 540, 541, 545, 560, 561, 566, 589, 658, 731
Arayüz Bildirimi 347, 391
Arayüz Referansları 396, 397
Aritmetik Operatörler 69, 71, 75
Aritmetik Operatörlerinin Aşırı Yüklenmesi 179, 225
ArrayList 127, 261, 522, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 572, 573, 575, 582, 588, 589, 591, 675, 700
ArrayList sınıfı 127, 261, 560, 562, 565
Arrays 123, 133

ASP.NET 12, 16, 637, 638
assembly 14, 16, 20, 269, 486, 487, 488, 489, 493, 599, 638, 640, 641, 643
Assembly Sınıfı 405, 486
Atama Operatörleri 69, 71, 74, 90, 91
Atlama (Jump) Deyimleri 95, 119
Attributes 405, 499, 630
Automatic Property 681, 702

B

Backward Compatibility 464
Bilinçli Tür Dönüşümü 51, 57
Bilinçli Tür Dönüşümünün Sakıncaları 51, 58
Bilinçsiz Tür Dönüşümü 51, 52
Binary Operatörler 70, 71
BinaryReader 291, 310, 321
BinaryWriter 291, 310, 321
Binder 513, 514, 515
BitArray 274, 559, 561, 577-579, 580
BitArray sınıfı 578
BitConverter Sınıfı 253, 286
Bitsel operatörler 72, 82, 83
blok 33-35, 61, 97, 108, 109, 289, 359, 419, 420, 422, 424, 466, 477, 478, 635
bool 41, 45, 47, 55, 67, 82, 100, 125, 135, 140, 198, 200, 230, 274, 275, 293, 316, 501, 518, 536, 546, 566, 568, 572, 573, 577-580, 679
boxing 48, 49, 62, 64, 65, 135, 147, 522, 525, 526, 530, 546, 562, 564, 608
break 95, 102, 103, 119-121, 591, 595
Budama ve Doldurma İşlemleri 291, 330
Buffer Sınıfı 253, 288
Build 270, 272, 469, 506, 633, 635, 646, 650, 651
Büyük Türün Küçük Türe Dönüştürülmesi 51, 56
byte 12, 41-44, 47, 51-59, 67, 83-89, 114, 160, 161, 236, 249, 250, 274, 286-293, 310-317, 321, 354, 464, 466, 468, 471-475, 479-481, 577, 578
byte code 12

C

C# 7-39, 47, 48, 51, 54-57, 62, 63, 66, 70, 72, 73, 76, 91, 93, 95, 96, 100, 111, 121, 144, 150-154, 162-165, 180, 181, 191, 193, 271, 197, 204, 206-208, 214, 219, 221, 222, 230, 233, 237, 243, 244, 249, 251-255, 269, 271-275, 279, 280, 281, 291, 292, 309, 323-325, 340, 347, 349, 351, 352, 373, 398, 401-408, 420, 422, 428, 431, 440, 442, 443, 455, 457, 459, 464-467, 470, 476, 478, 483, 485, 486, 499, 500, 502, 505, 510-518, 521, 522, 526, 527, 531, 536, 538, 543, 545, 546, 548, 549, 551, 559, 587, 591, 593, 597, 611, 619, 621, 633, 637-639, 645, 660, 679, 681, 682-685, 686-698, 702, 703, 706-710, 714, 715, 731, 744, 760, 761
C++ 8, 15, 17, 27, 32, 38, 48, 93, 100, 103, 119, 123, 124, 126, 149, 150, 154, 156, 162, 163, 182, 204, 206, 207, 218, 254, 273, 348, 349, 351, 406, 455, 464-467, 470, 510, 522, 527, 531, 560, 645, 682
C# 3.0 Yenilikleri 681, 682
C# 5.0 ve Asenkron Programlama 751
C# 7.0 47, 50, 105, 107, 157, 169, 170, 554-556
Çalışma Zamanında Metot Çağırma 405, 497
call by reference 154, 156
call by value 154
case 32, 102, 103, 104, 121
catch bloğu 410-428
Char 40, 47, 274, 275, 277, 278
checked 51, 60, 61, 69, 71, 72, 94
C# Komut Satırı Derleyicisi 19
Class 27, 269, 270, 350, 502, 506, 521, 532, 638, 641-644
Class Overloading 521, 532
Clear metodu 140
CLR 12-14, 124, 422, 430, 486, 511-516, 526, 527, 532, 551, 600, 682, 688, 690, 691, 696, 704, 708, 710
CLS 12, 15
çok biçimlilik 350, 387, 412
çok boyutlu diziler 130, 131
Çok Boyutlu İndeksleyici 179, 242
çok çeşitlilik 380
Çoklu Temsilciler 405, 436
Console I/O İşlemleri 291, 322
const 35-37, 179, 214, 216-218
Constraints 521, 536
ConstructorInfo sınıfı 495, 497
Constructors 179, 199
Continue 653, 655
Contravariance 521, 545
Convert Sınıfı 51, 66, 287, 288
Copy Constructor 179, 204
Covariance 521, 545
CreateInstance 123, 135, 136, 513
csc.exe 19, 170, 271
CTS 12, 15, 38, 39, 47, 48, 50, 274

D

DbProviderFactory 597, 616, 618
Debug 633, 635, 650, 651, 652, 653, 654, 736
Debugger 465, 635, 650-655
decimal 40, 44, 45, 47, 53, 55, 56, 67, 279, 280, 281, 546
Deconstruct 555
Deconstructer 557
default 102, 104, 200, 219, 521, 536, 730
Default Constructor 179, 202
default Operatörü 521, 536
değer tipleri 32, 38, 40, 41, 62-64, 93, 154, 155, 180, 186, 466, 483, 530, 546, 547, 548
değişkenler 29, 33, 34, 37, 38, 51, 55, 61, 96, 150, 174, 175, 180, 182, 187, 199, 200, 208, 210, 213, 216, 218, 223, 248, 511, 515, 516, 547, 634, 686, 692, 694, 699, 702
değişken tanımlama 40, 50, 695
delegates 405, 406, 430
derleyici 19, 30, 34, 59, 60, 80, 160, 161, 222, 233, 255, 262, 353, 380, 420, 422, 461, 462, 466, 501, 526, 548, 682, 690, 695, 704, 705
Desen Eşleştirme ix, 105, 107
Destructors 179, 206
deyimler 97, 105, 116, 147, 467, 665, 700
DirectoryInfo Sınıfı 291, 300
Directory Sınıfı 291, 294
Dispose 179, 206, 262, 263, 391, 667
diziler 92, 124, 126, 130, 131, 133, 139, 153, 163, 164, 180, 238, 289, 401, 478, 480, 559, 561, 562, 566-568
Dizilerde Arama 123, 139
Dizileri Kopyalama 123, 137
Dizileri Sıralama 123, 138
dizileri sıralamak 567
dizin 48, 270, 296, 300-303, 307, 567, 643
dizi tanımlama 559
dll 13, 15, 16, 20, 269, 270-274, 353, 354, 486-488, 493, 505-509, 599, 638
DLR 405, 510-516
Document Object Model 597, 629
DOM 597, 629
Döngülerle Dizi İşlemleri 123, 127
Döngü Yapıları 95, 105, 108
Dönüşüm Operatörünün Aşırı Yüklenmesi 179, 233
Dosya ve Klasör İşlemleri 291, 293
Dosya Yazma ve Okuma İşlemleri 291, 309
double 32, 38, 40-47, 54, 55, 59, 67, 80, 128, 146, 148, 160-162, 165, 182, 198, 223-226, 234-236, 275-279, 287, 467, 468, 483, 522, 526, 538, 547, 548, 729
do while 95, 108, 117, 118, 120
Düzenli İfadeler 291, 339
Düzensiz Diziler 123, 133
Dynamic 510, 516, 642, 643
DynamicObject Sınıfı 516

E

Emniyetsiz (unsafe) Kod 405, 464
Enumeration 179, 249
EventInfo 496
Events 405, 443
Exception Filter 429
Exception Handling 61, 405, 406
Extension Methods 681, 703
External Alias 253, 269

F

FieldInfo 496
FileInfo Sınıfı 291, 303
File Sınıfı 291, 297
FileStream ile Yazma ve Okuma 291, 311
FileStream sınıfı 310
fixed 476, 477, 478, 485
float 37-38, 41, 43, 47, 52, 55, 59, 60, 67, 75, 160, 276-280, 335, 543
fonksiyonlar 143-144, 348, 349, 464, 718, 726
for 33-34, 95, 108-120, 125, 127-129, 131, 132, 144, 211, 229, 285, 479, 689
foreach 95, 108, 119, 123, 127, 129, 130, 135, 163, 342, 391, 399, 400, 560, 564, 570, 591-593, 629, 693, 736, 748

G

garbage collector 207, 290, 391, 464, 476
Garbage Collector 206, 253, 290
Generics 521, 522, 536, 682
Genişletme Metotları 681, 703
Geriye Uyumluluk 464
get 193-200, 220-224, 239-241, 248, 325, 358, 384, 392, 401, 412, 448, 566, 579, 696, 702
global 207-210, 217, 273, 371
Gösterici 69, 93, 326, 405, 464, 473, 475, 479
Gösterici Aritmetiği 405, 473
Gösterici Bildirimi 405, 466
Gösterici Operatörleri 69, 93, 405, 466
goto 95, 102, 103, 104, 119, 121

H

Hashtable 559, 561, 569, 570-575, 588, 628
Hashtable Sınıfı 559, 569
hata ayıklayıcı 635
Heap Bölgesi 30, 187

I-İ

İç içe (nested) Geçmiş İsim Alanları 253, 267
ifadeler 35, 45, 74, 79, 104, 110, 221, 292, 335, 339, 340, 342, 467, 690
if deyimi 96, 102
if-else 91, 96, 102, 104, 418, 459, 549
IL 12, 14, 650, 692, 696, 702, 706, 710, 716, 717
İlişkisel operatörler 228
İlişkisel Operatörlerin Aşırı Yüklenmesi 179, 228
İmza 143, 159
indeksleyiciler 237, 238, 242
Indexers 237, 238
inheritance 347, 517
int 15, 32, 37-43, 47, 51-58, 60-67, 80, 83, 84, 93, 109, 126-129, 135, 136, 140, 141, 145-148, 153, 154, 157, 160-166, 170, 221, 228, 233-238, 246, 249, 250, 255, 274, 275-288, 292, 311, 318, 321, 322, 323, 327, 402, 417, 431, 433, 466-483, 504, 512, 522, 525, 526, 527, 529, 530, 535, 536, 538, 543-549, 566-569, 573, 574, 577-585, 593, 607, 608, 686, 706, 709, 713, 747
I/O Sistemi 291, 292
is 106
İşaretçi 609
isim alanı 17-19, 27, 66, 253, 255-257, 260-262, 265-269, 274, 292, 322, 406, 587, 593, 600, 704, 748
İsim Alanı Bildirimi 253, 256
isim alanları 27, 28, 254, 260, 267-269, 597, 599, 642, 732
İsimlendirilmiş (Named) Parametreler 143, 166
isimlendirilmiş parametreler 165
İsim Saklama 347, 365
isimsiz metotlar 690, 709
İsimsiz Tipler 681, 695
İstisna Filtreleme 429
İstisnai Durum Sınıfları 405, 406, 422, 423
İstisnai Durum Yakalama 405, 407
İstisnai Durum Yönetimi 405, 406

J

Jagged Arrays 123, 133
Java 8, 11, 12, 17, 254, 510, 8
Java Sanal Makinası 12
Java Virtual Machine 12
JIT derleyicileri 14, 650
Join() metodu 332
Jump 95, 119

K

kalıtım 62, 181, 214, 347, 350, 354, 355, 358, 396, 412, 423, 424, 491, 495, 502, 560
karşılaştırma operatörleri 79, 589
kaynak kod 12, 26, 161, 367, 455, 461, 625, 633, 641-648
Kısıtlar 521, 536
Koleksiyonlar 522, 545, 559, 589, 628
Kompleks Sınıfı 179, 223
Kopyalayıcı Yapıcı Metot 179, 204
Koşul İfadeleri 95
Küçük Türün Büyük Türe Dönüştürülmesi 51, 52

L

Lambda İfadeleri 681, 708, 711, 713
LINQ 516, 681-698, 702-708, 712-718, 724, 725, 726, 729, 731, 732, 736-741, 744, 746-749
LINQ mimarisi 705, 706, 708, 731, 744
LINQ to Object Mimarisi 681, 688
LINQ to SQL 681, 686-688, 697, 715, 731, 736
Liste Koleksiyonu 521, 522
Local functions 169, 170
long 43, 47, 53, 67, 250, 279, 280, 283, 287, 288, 293, 316, 475
Loops 95, 108
LSB 286

M

Main 27, 34, 35, 157, 160, 170, 171, 176, 183, 198, 208, 223, 255, 262, 268, 352, 371, 390, 397, 402, 413, 416, 427, 430-435, 443, 445, 462, 474, 484, 487, 507, 636, 643, 657, 667, 669, 709
makine kodu 11, 430
Makine kodu 430
Mantıksal Operatörler 69, 71, 81
Mantıksal Operatörlerinin Aşırı Yüklenmesi 179, 230
Matris Dizileri 123, 130
Member Binding 514
MemberInfo sınıfı 490
metadata 16, 486
MethodInfo sınıfı 487, 491
method signature 159
metot 14, 27, 28, 137, 138, 139, 144, 145, 147, 148, 149, 150, 151, 153-156, 159-171, 175, 188, 193, 196, 198, 200, 207-219, 222, 225, 226, 233, 234, 239, 247, 260, 276-281, 284-296, 301-307, 311-319, 322, 326-331, 337, 342, 352, 354, 355, 360, 374-397, 400-406, 412, 415, 423, 431-451, 465, 486-505, 510-516, 525, 529, 535, 537, 541, 542, 547, 561, 566-569, 573-586, 598, 605, 609, 612, 623, 628, 629, 634, 638, 642, 659, 661, 665, 667, 684, 690, 697-708, 710, 714, 718, 729, 736, 742, 747
metot argümanları 144
Metot Bildirimi 143, 145
Metot imzası 159
Metotlar 135, 143, 148, 150, 159, 162, 167, 179, 187, 194, 199, 206, 208, 211, 316, 347, 374, 402, 405, 431, 440, 521, 542, 619, 681, 714, 718
Metotların Aşırı Yüklenmesi 143, 159
Metot Parametresi 152, 153
MSB 286
msdn 285, 761
Multicast Delegate 405, 436, 438
Multidimensional 123, 130

N

Name Hiding 347, 365
namespace 17, 27, 61, 253, 256
nesne 8, 12, 15, 38, 49, 52, 54, 56, 62, 65, 79, 93, 121, 124, 127, 139, 145, 160, 173, 180, 184, 188, 199, 206-222, 233, 238, 240, 244, 246, 248, 255, 257, 261, 265, 274, 280, 294, 301, 321, 328, 337, 342,

347-357, 360-380, 387, 390, 396, 397, 402, 408, 410, 414, 416, 423, 432, 445, 447, 453, 456, 464, 467, 476, 478, 489, 497, 512, 515, 518, 522, 525, 531, 533, 535, 539-545, 559, 567, 583, 588, 592, 598, 616, 623, 657, 683, 688, 695, 697, 701, 703, 729, 731, 748
nesne yönelimli programlama 121, 144, 180, 199, 347, 348, 349, 350, 351, 360, 371, 375, 522, 683, 703
nitelikler 621, 625, 732
Null Condition Operatörü (?) 549
Numaralandırmalar 179, 249

O-Ö

Object 12, 39, 48, 310, 350, 373, 374, 407, 490, 494, 495, 522, 523, 561, 597, 629, 681, 684, 686, 687, 688, 697, 698, 731, 737, 741
olaylar 196, 406, 440, 443, 445, 452, 454, 485, 535, 544, 649, 658, 659, 661, 666
Önişlemci 405, 455, 456, 457, 635
Önişlemci Komutları 405, 455
operatör 70, 72, 74, 82, 87, 90-93, 176, 221-225, 228, 229, 230, 233, 234, 237, 273, 285, 467, 484, 536, 540, 549, 725
Operatör Aşırı Yükleme 179, 221
Operatör Önceliği 69, 72
Operator Overloading 179, 221
Özel Nitelik Bildirimi 405, 502
Özyineli (Recursive) Metotlar 167, 168

P

ParameterInfo sınıfı 491, 493
parametre aktarımı 154, 156
Pattern Matching 105
Plugin 405, 505
Plugin (Eklenti) Tabanlı Programlama 405, 505
Pointer 405, 464
polymorphism 443
property 510, 516
public 433, 484, 491, 504, 670
Python 510, 514, 518, 708

Q

Queue 559, 561, 585, 586, 589
Queue Sınıfı 559, 585

R

ReadOnly 566
ref 143, 154, 155, 156, 157, 223
referans ile parametre aktarımı 154, 156
Referans (Reference) Tipleri 38
referans tipleri 32, 38, 63, 93, 154, 180, 185, 218
Referans ve Değer Türleri Arasındaki Dönüşüm 51
ref return 143, 158
Register Bölgesi 30
Regular Expressions 291, 339
Remove 563, 565, 572, 574, 747
Ruby 510, 514, 518

S-Ş

Sabit Bölge 31
sabit veriler 35
Şablon Tipler 521, 522, 533, 534, 551
Şablon Tipler Arasında Türetme 521, 533
Şablon Tipli Koleksiyon Nesneleri 559, 587
Şablon Tipli Metotlar 521, 542
Şablon Tipli Temsilciler 521, 543
Sanal Metotlar 347, 374
sbyte 42, 47, 250, 274
sealed 385, 386, 387, 403, 404, 717
Set 579, 580, 646, 715, 718
short 42, 47, 53, 56, 67, 250, 287, 289, 290
Singleton 179, 218, 219, 220, 221
Singleton (Tek) Nesneler 179, 218
sizeof 473, 475, 480
Sınıf Aşırı Yükleme 521, 532
sınıf bildirimi 26, 211, 243, 244, 384, 386, 503, 526, 531, 533, 535, 695
Sınıf Kütüphanesi Oluşturma 347, 352
sınıflar 406, 407, 412, 422, 423, 443, 466, 473, 476, 490, 496, 497, 499, 516, 641, 642, 656, 658, 676
Sınıflara Metot Ekleme 179, 187
sınıf nesnesi 419, 424, 432
Sınıf Tabanlı Programlama 350
SortedList 559, 561, 573, 574, 575, 588
Split() metodu 331, 332, 346
stack 464, 470, 476, 480, 482, 483
stackalloc 478, 480, 482
Stack Bölgesi 30, 187
Stack Pointer 30
Stack Sınıfı 559, 582
Statik Değişkenler 179, 210
statik diller 510
statik metotlar 208-210, 280, 287, 288, 294
statik sınıflar 214
statik yapıcı metotlar 208, 211, 221
StreamReader Sınıfı 291, 317
StreamWriter Sınıfı 291, 319
String 412, 490, 491, 495
String.Format() 291, 334, 336
String İşlemleri 291, 333
String Metotları 291, 326
String Tanımlama 291, 325
Structs 179, 243
switch 95, 96, 100-104, 119, 120, 121, 171
switch Deyimi 95, 102, 104
System.Array sınıfı 478
System.Collections İsim Alanı 559
System.Convert Sınıfı 51, 66
System.Data 18, 597, 599, 600, 601, 603, 604, 610, 618, 732
System.Enum Sınıfı 251, 252
System isim alanı 17, 18, 27, 66, 261
System.Math 173, 174
System.Math Sınıfı 173, 174
System.Net 18
System.Xml 18, 597, 619, 622, 625, 629, 632, 744, 745, 746

T

takma isim 266, 269-274
Tarih ve Saat Biçimlendirme 291, 337
Tarih ve Zaman İşlemleri 281, 282
Tek Boyutlu İndeksleyici 179, 238
Temel ve Türeyen Sınıf Nesneleri 347, 369
Temsilciler 405, 430, 431, 436
Ternary Operatörler 70, 71
this 428
throw 407, 408, 414, 415, 416, 418, 424
tip güvenliği 512
ToString() 412, 425
true ve false operatörleri 229
true ve false Operatörlerinin Aşırı Yüklenmesi 179, 229
try blokları 410, 415, 417
Tuple 521, 551
Tuple Literal 554, 556
tür dönüştürme operatörü 46, 58, 59, 64, 83, 92, 234, 235, 526, 549
tür dönüşümü 48, 51-59, 65, 148, 161, 234, 235, 236, 251, 423, 471, 475, 514, 548
Türetmenin Yapılması 347, 355
Type Inference 521, 543
type safety 57, 369, 522
Type Sınıfı 405, 489

U-Ü

uint 43, 47, 67, 250, 275, 279, 287, 472
ulong 43, 47, 67, 249, 250, 275, 279, 287
Unary Operatörler 70, 71, 74
Unboxing 51, 64, 65
unchecked 51, 60, 61, 69, 71, 72, 94
UNICODE 177, 277
unsafe code 469
ushort 42, 47, 67, 250, 275, 287
using 17, 28, 29, 128, 254, 257, 260-274, 354, 636, 706

V

ValueTuple 551
Value Types 521, 546
var 50
varsayılan yapıcı metot 200, 202, 538, 698
VB.NET 8
veri kaybı 52, 53, 58, 160, 237, 287
veri türleri 15, 17, 29, 38, 54, 62, 64, 95, 125, 154, 182, 221, 233, 237, 245, 279, 321, 335, 373, 483, 573, 599

Visual Studio.NET 12, 19, 269, 270, 272, 463, 464, 469, 515, 529, 633, 634, 635, 636, 637, 639, 641, 643, 645, 648, 649, 650, 653, 654, 655, 660, 661, 668, 681, 682, 693, 696, 741
Visual Studio.NET Projeleri 633, 636
void 147, 148, 149, 150, 200, 228, 402, 431, 432, 445, 446, 447, 473, 474, 567, 568, 569, 572, 573, 574, 575, 580, 709, 717

W

Web Servisleri 620, 638
while 33, 95, 108, 115, 116, 117, 118, 120, 2296

X

XML 9, 12, 18, 214, 597, 598, 609, 619, 620, 621, 622, 623, 626, 627, 628, 629, 630, 631, 632, 638, 643, 683, 684, 703, 744, 745, 746, 761
XML İşlemleri ve System.Xml 597, 619
XOR 74, 87, 176, 177, 314, 577

Y

Yansıma 405, 486, 489, 505
yapıcı metotlar 201, 206, 208, 211, 212, 213, 215, 221, 279, 280, 495, 628
Yapıcı Metotların Kalıtımdaki Rolü 359
yapılar 154, 155, 180, 221, 248, 274, 276, 281, 282, 367, 444, 473, 483, 598, 642, 688, 705, 709
yapı türünden göstericiler 466
Yerel metotlar 169, 170
yield 559, 591, 592, 595
yıkıcı metotlar 206, 207, 248, 249, 392