İÇİNDEKİLER

NESNE YÖNELİMLİ PROGRAMLAMAYA GİRİŞ

Nesne yönelimli programlama (Object - Oriented Programming) bir bilgisayar programlama yaklaşımıdır. Günümüzde pek çok çağdaş programlama dili tarafından desteklenmektedir.

1960'lı yılların sonuna doğru ortaya çıkan bu yaklaşım, o dönemin yazılım dünyasında beliren bir bunalımın sonucudur. Yazılımların karmaşıklığı ve boyutları sürekli artıyor, ancak belli bir nitelik düzeyi korumak için gereken bakımın maliyeti zaman ve çaba olarak daha da hızlı artıyordu. OOP'yi bu soruna karşı bir çözüm haline getiren başlıca özelliği, yazılımda birimselliği benimsemesidir. OOP ayrıca, bilgi gizleme, veri soyutlama, çok biçimlilik ve kalıtım gibi yazılımın bakımını ve aynı yazılım üzerinde birden fazla kişinin çalışmasını kolaylaştıran kavramları da yazılım literatürüne kazandırmıştır. Sağladığı bu avantajlardan dolayı, OOP günümüzde geniş çaplı yazılım projelerinde yaygın olarak kullanılmaktadır.

OOP 'nin altında yatan birimselliğin ana fikri, her bilgisayar programının, etkileşim içerisinde olan birimler veya nesneler kümesinden oluştuğu varsayımıdır. Bu nesnelerin her biri, kendi içerisinde veri işleyebilir ve diğer nesneler ile çift yönlü veri alışverişinde bulunabilir. Hâlbuki OOP 'den önce var olan tek yaklaşımda, programlar sadece bir komut dizisi veya birer işlev (fonksiyon) kümesi olarak görülmektedirler.

Günümüzde çok çeşitli nesne tabanlı programlama dilleri olmasıyla beraber, en popüler diller sınıflar üzerine kurulmuşlardır. Bu dillerde nesneler sınıfların birer üyesidir ve nesnelerin tipini de bu sınıflar belirlerler.

Object Orientented Programming Nedir?

Nesneye Yönelimli Programlama, C# tarafında tanımlı olmayan yapıları bizlere tanımlama imkânı sağlar.

Örneğin,
Bir TextBox oluştururken `TextBox txt_giris = new TextBox();` şeklinde bir TextBox tanımlayabiliyoruz. TextBox nesnesini tanımlayabilmemizin nedeni .NET Framework ortamında tanımlı olmasından kaynaklanıyor. Button, TextBox, Random vb. sınıflar Framework ortamında tanımlıdır.

Peki, biz kendi ihtiyaç duyduğumuz yapıları nasıl tanımlayacağız. Kendi ihtiyaç duyduğumuz yapılar .NET Framework ortamında tanımlı olmayacaktır.

Öncelikle, 'Oluşturmak istediğiniz nesne nedir?' Sorusunu kendimize sormamız gerekiyor. Bu sorumuza verdiğimiz cevap bize nesnemizin ismini vereceği için artık nesnemizi oluşturmaya başlayabiliriz.

Örneğin,
Oluşturmak istediğiniz nesne nedir? Sorumuza cevap olarak 'Bilgisayar' cevabını verdiğimizi düşünelim. Artık bilgisayar isimli bir nesne oluşturmaya başlayabiliriz.
Nesnemizi oluştururken şimdi de 'Bilgisayar nesnesinin _____ özelliği vardır.' sorusuna vereceğimiz cevaplarla oluşturacağımız nesnenin özelliklerini bulmaya başlayabiliriz.

- Bilgisayarın Ekranı vardır.
- Bilgisayarın Anakartı vardır.
- Bilgisayarın İşlemcisi vardır.
- Bilgisayarın Belleği(Ram) vardır.
- Bilgisayarın Genişleme Kartı(Ekran kartı vb.) vardır.
- Bilgisayarın Güç kaynağı vardır.
- Bilgisayarın Optik Sürücüsü(DVD-RW,BlueRay vb.) vardır.
- Bilgisayarın Sabit Diski(HDD) vardır.
- Bilgisayarın Klavyesi vardır.
- Bilgisayarın Faresi vardır.

Yukarıda bulduğumuz özellikleri C# programlama dilinde kullanabilmemiz için değişken olarak bu özellikleri tanımlamamız gerekir. Öncelikle her bir özellik için kullanacağımız veri tipimizi belirlememiz gerekmektedir. Her bir

özelliğimiz için veri tipimizi belirledikten sonra bütün özellikleri bir arada tutmak için ne yapacağımızı düşünürsek ilk olarak aklımızı diziler gelecektir.

Bu özellikleri bir arada tutabilmek için bir dizi tanımlarsam dizimin veri tipinin Object olması gerekecektir. Ama Object tipinde bir dizi ile bu işlemi yaparsam verilerin dizi içerisindeki sıralamasını bilmem ve verileri kullanmak için çağırırken de unboxing işlemini yapmam gerekecektir. Dizi ile nesnemizi oluşturmak bizim için zahmetli ve zor olacağından nesnemizi OOP prensiplerine göre hazırlamamız gerekecektir.

Nesne Yönelimli Programlamanın Avantajları

- Reusability(Tekrar Kullanılabilirlik)
- Extensibility(Genişletilebilirlik)
- Maintainability(Sürdürülebilirlik)

Bir Eticaret sitesi projesi hazırladığımızı ve bu eticaret sitemizin sadece Türkiye içine satış yaptığını varsayalım. Sistemimizi kurduktan sonra eticaret sistemimizi büyütüp yurt dışına açıldığımızı düşünelim.

Böyle bir senaryo da kalıtım dediğimiz bir olay ile müşteri sınıfının özelliklerini miras alan yabancı müşteri ve yerli müşteri adında iki class oluştururuz ve iki class(sınıf) içinde tekrar tekrar kod yazmak zorunda kalmayız sadece farklılıkları ekleriz ve önceden yaptığımız çalışmaları kullanabiliriz. Gördüğümüz gibi tekrar kullanılabilirlik açısından oldukça başarılı bir yöntem. Yine bu örnekte görüldüğü gibi projemizi daha geniş bir alana yayma özelliğini de(Genişletilebilirlik) kullanmış oluruz. Bu ikisinin yanında aynı projemizi yeni projemiz içinde devam ettirebiliriz. Yani yabancı müşteriye satış yapmak için yeni bir yazılıma ihtiyaç duymayız.(Sürdürülebilirlik).

Nesne Tabanlı Programlama Dilleri Nelerdir?

Programlama Dilleri tarihinde 1960'lı yıllara gelindiğin de Simula(1962-1967 arası) isimli bir dil geliştirilmiştir. Geliştirilen bu dil ilk nesneye yönelik programlama dilidir.

1970'li yıllara gelindiğinde günümüz programlama dillerinin tamamını etkileyen C dilinin temelleri atılmaya başlanmıştır. 1972 yılın da Dennis Ritchie tarafından Bell Laboratuarlarında C dili geliştirilmiştir.

1979 yılında Bjarne Stroustrup tarafından C++ dili geliştirilmiş ve 1983 yılında C++ dili yayınlanmıştır. C++ dili, C dilini temel alarak geliştirilmiştir ve Nesneye yönelik programlama kavramını kabul etmektedir. 1986 yılında Bertrand Meyer tarafından geliştirilen Eiffel programlama dili C++ gibi nesneye-yönelik bir dildir.

En yaygın OOP dillerinden bazıları, C#, Python, C++, Objective-C, Smalltalk, Delphi, Java, Swift, Perl, Ruby ve PHP' dir.

Nesne Nedir?

Nesne niteliği ve özelliği olan, görülen, hissedilen, duyulan her şeydir.

Object Oriented Prensipleri

Encapsulation

Nesnenin üyelerine yapılan erişimin kontrol altına alınmasına ve bu kontrolün nesnenin kendisi tarafından yapılmasını sağlamaktır. Amaç fiedl'ları private yaparak bu alanlara dışarıdan erişimi önlemek ve get-set metotları ile kontrolü sağlamaktır.

İnheritance

Inheritance (miras alma, kalıtım), bir nesnenin özelliklerinin farklı nesneler tarafından da kullanılabilmesine olanak sağlayan OOP özelliğidir. Yazılan bir sınıf bir başka sınıf tarafından miras alınabilir. Bu işlem yapıldığı zaman temel alınan sınıfın tüm özellikleri yeni sınıfa aktarılır.

Polymorphism

Polymorphism(Çok Biçimlilik) bir metodun farklı sınıflar için farklı etkiler oluşturmasını sağlayan bir yöntemdir.

Abstraction

Abstract Class, ortak özellikli sınıflara Base Class olma görevini üstlenir. Abstract Class'lar, diğer sınıflara base Class olmak için yazılır. Bu nedenle Abstract Class'dan nesne türetilemez. Abstract öğelerin amacı, kendisinden kalıtım alan sınıflarda bir takım özelliklerin kullanılmasını zorunlu kılmaktır.

SINIF KAVRAMI

Sınıf Nedir?

Sınıflar (Class) nesne üretmek için kullandığımız kalıplardır. Bilgisayar ortamında tanımlayacağımız her bir sınıf, bizim nesneler üretmemizi sağlar. Örneğin ben bir insan sınıfı tanımlamak istediğimde oluşturacağım sınıf tüm insanlarda bulunan özellikleri taşır. Saç rengi, göz rengi, boy, kilo ve diğer özellikler sınıf içerisinde tanımlanır ama içerisine değer atanmaz.

Sınıf Nasıl Oluşturulur?

Nesneler sınıfların birer örneğidir. Biraz daha açıklayıcı olmak gerekirse nesneler aslında sınıftaki verilerin anlamlı bir şekilde modellemesidir. Peki, nesnelerin ne işe yaradığını düşünecek olursak, aslında yaptığımız tanım neticesinde nesneler sınıfları bizim için işlem yapılır hale getiriyor diyebiliriz.

Örneğin;

Yukarıda özelliklerini bulduğumuz Bilgisayar sınıfımızın öncelikle özellikleri ile yaratıyoruz.

Sınıf Oluşturmak

```
public class Bilgisayar
{
   public bool MonitorVarMi { get; set; }
   public string Anakart { get; set; }
   public string Islemci { get; set; }
   public int BellekMiktari { get; set; }
   public string GenislemeKarti { get; set; }
   public bool GunKaynagiVarMi { get; set; }
   public string OptikSurucu { get; set; }
```

```
    public int SabitDiskAlani { get; set; }
    public bool KlavyeVarMi { get; set; }
    public bool MouseVarMi { get; set; }


}
```

Artık Bilgisayar adında gerçek bir nesnemiz oluşmuş durumdadır.

Instance Nedir?

.Net Ortamında bir nesne oluşturma işlemine **instance** almak denir.

RAM üzerinde oluşturduğumuz bir nesneye, nesneyi oluşturduğumuz sınıftaki tüm özellikleri aktarmak olarak tanımlayabiliriz.

Örnek;

Instance Almak

```
Bilgisayar computer = new Bilgisayar();
```

Bir sınıfa ait bir nesne oluşturduğumuzda, o nesne de sınıfta tanımlanan bütün değişkenlerin birer kopyası oluşur. Yani Bilgisayar sınıfından yararlanılarak oluşturduğumuz her nesnede sınıfımızda tanımlı olan özelliklerimizin birer kopyaları bulunmaktadır. (Anakart, Islemci, BellekMiktari vb.) Bu aynı zamanda oluşturduğumuz nesnenin içindeki değişkenlerim farklı değerler alabileceği anlamına da gelmektedir. Bu değişkenlere ulaşmak için nokta(.) operatörü kullanılır.

Instance Alınan Özellikleri Doldurmak

```
computer.MonitorVarMi = true;
computer.Anakart = "P5 Anakart";
computer.Islemci = "i7 4.Nesil";
computer.BellekMiktari = 8;
computer.GenislemeKarti = "GTX960";
computer.GunKaynagiVarMi = true;
computer.OptikSurucu = "Bluray";
computer.SabitDiskAlani = 1000;
computer.KlavyeVarMi = true;
computer.MouseVarMi = false;
```

Field Nedir?

Bizler verileri saklamak için değişkenleri kullanırız. Değişkenler RAM üzerinde verileri saklamak için kullanılan yapılardır. Class'lar içerisinde bir değişken oluşturup bu değişkenlerde veri saklayabiliriz. Field'lar class içerisinde oluşturulan, dışarıya kapalı değişkenlerdir. Field'lar içerisinde değer saklar. Field'lar dışarıya kapalı olduğundan field'a değer atanamaz.

Property Nedir?

Property field'a değer atamak ve field'da ki değeri okumak için kullanılan aracı bir yapıdır. Bizler Field'lara erişebilmek için Propertyleri kullanmak zorundayız çünkü Field – Property yapısı bizim Field'ı mıza doğrudan erişim sağlamamızı engeller ve Property içerisinde veri kontrollerini yapma imkânı tanır. Yani RAM üzerinde tuttuğumuz verinin kontrolünü Property içerisinde yapabiliriz.

Encapsulation

Nesnenin üyelerine yapılan erişimin kontrol altına alınmasına ve bu kontrolün nesnenin kendisi tarafından yapılmasını sağlamaktır. Amaç fiedl'ları private yaparak bu alanlara dışarıdan erişimi önlemek ve get-set metotları ile kontrolü sağlamaktır.

Encapsulation Örneği

```
private string _ad; //_ad değişkenimize dışarıdan erişimi engelledik.


public string Ad //_ad Field'ımıza dışarıdan erişilemeyecek fakat Field dan türetilen Ad isimli Property ile dışarıyla haberleşip Field dan veri alıp dışarıya atacak, dışarıdaki değeri alarak Field'a ulaştıracaktır.
{

    get { return _ad; }//Field daki değeri dışarıya gönderir.
    set { _ad = value; }//Dışarıdan alınan değeri Field'a atar.

}
```

Access Modifiers

Erişim belirteçi anlamına gelen Access Modifiers, classlar, metotlar ve değişken gibi yapıların erişim durumlarını belirler.

Erişim Belirleyici	Açıklama
Private	Bu erişim belirteci ile tanımlanan öğeler sadece tanımlandıkları sınıf (class) ya da yapı (struct) içinde erişilebilirler. En kısıtlayıcı belirteçtir.
Public	Public olarak tanımlanan öğeler aynı projeden ya da kendisini referans olarak gösteren diğer projeler tarafından erişilebilirler.
Protected	Bu şekilde tanımlanan öğeler, tanımlandıkları sınıf (class) ya da yapı (struct) veya bu sınıflardan türetilmiş diğer sınıflar içerisinden erişilebilirler.
Internal	Internal tanımlanan classlar bulundukları proje içerisinde her yerden erişilebilir ancak başka bir proje tarafından kullanılamazlar.
Protected Internal	Bu belirteç uygulanan öğenin Protected ya da Internal olduğunu vurgular. Yani Protected Internal bir öğe aynı proje içindeki tüm sınıflardan erişilebileceği gibi, bulunduğu sınıftan türetilmiş diğer sınıflardan da erişilebilir

Constructor Nedir?

Constructor, yapılandırıcı metot olarak kullanılır ve görevi oluşturulan nesneyi ilk kullanıma hazırlamasıdır. Bir class'dan instance alındığında ilk tetiklenecek olan metottur. Geriye dönüş tipi yoktur.

Constructor'ın bilinen temel özellikleri:

- Kendi sınıfı ile aynı isme sahip olması,
- Açık bir dönüş tipi olmaması,
- Başka sınıflar tarafından kullanılabilmesi için erişimin public olmasıdır.

Programci.cs

```
class Programci
{
        int yasi;
        string adi;
        string soyadi;
        string kullandigiDil;

        // Hic parametre almayan bir constructors..
        public Programci()
        {
            this.adi = null;
            this.yasi = 0;
            this.soyadi = null;
            this.kullandigiDil = null;
        }

        // İsmini ve yasini alan bir constructors..
        public Programci(int yasi, string adi)
        {
            this.adi = adi;
            this.yasi = yasi;
            this.soyadi = null;
            this.kullandigiDil = null;
        }




        // İsmini, soyismini ve yasini alan bir constructors..
        public Programci(int yasi, string adi, string soyadi)
        {
            this.adi = adi;
            this.yasi = yasi;
            this.soyadi = soyadi;
            this.kullandigiDil = null;
        }

        // İsmini, soyismini kullandığı dili ve yasini alan bir constructors..
        public Programci(int yasi, string adi, string soyadi, string kullandigiDil)
        {
```

```
            this.adi = adi;
            this.yasi = yasi;
            this.soyadi = soyadi;
            this.kullandigiDil = kullandigiDil;
        }
}
```

Yukarıda oluşturduğumuz Class'ımızı instance alarak kullanmak istediğimizde aşağıdaki gibi farklı şekillerde kullanabiliriz.

Oluşturduğumuz sınıftan instance almak

```
Programci kul1 = new Programci();
Programci kul2 = new Programci(23, "Mert");
Programci kul3 = new Programci(27, "Mert", "Kurt");
Programci kul4 = new Programci(30, "Mert", "Kurt", "C#");
```

Referance Types ve Value Types kavramları

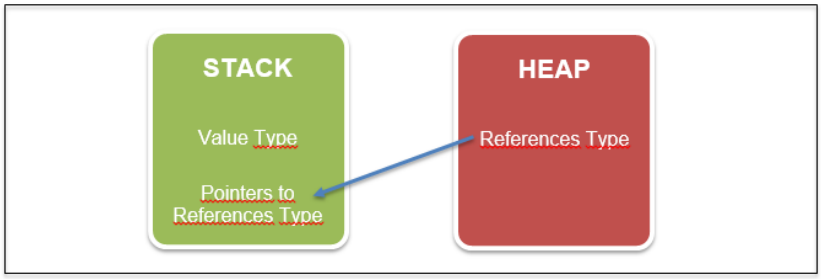
Value Type (değer tipleri), RAM üzerinde ki STACK alanında yaşarlar. STACK alanında açılan yerde doğrudan değişken içerisindeki bilgi tutulur. Int,bool, double, float vb. değerler STACK alanında saklanmaktadır.

Örnek

```
decimal sonuc = 5.45M;
```

İfadesinde, STACK bölümünde sonuc isminde bir alan açıyoruz. Bu alan içerisinde 5.45 sayısını tutuyoruz. Value Type değişkenlerde bilginin kendisi tutulur.

Referans Type ise değişken RAM'in HEAP alanında yaşar. String, diziler ve classlar RAM'in Referans Tipleri arasındadır. Referans tiplerinin değerleri HEAP alanında tutulurken, STACK alanında, değeri işaret eden bir adres bulunur. Biz veriyi okumak istediğimizde, STACK üzerinde değeri işaret eden referans adresinden, HEAP'de verinin tutulduğu alana giderek veriye ulaşabiliriz.



Struct nedir?

Aralarında mantıksal bir ilişki bulunan farklı türden bilgilerin tanımlandığı ve üstlerinde işlemlerin yapıldığı yapılara denilmektedir.

Struct ve Class Arasındaki Farklar

STRUCT	CLASS
Value Type	References Type
Inheritance alır, kendisi inheritance vermez	Inheritance hem alır hem verir
Access modifier'lardan Public yapısını kullanır.	Access Modifier'lardan Public ve İnternal yapısını kullanır.
16 KB den daha küçük veriler için uygundur.	16 KB den daha büyük veriler için uygundur.

Struct Kullanımı

Öğrenci isimli bir struct oluşturmak istersek öncelikle hangi özellikleri kullanacağımızı belirlememiz gereklidir. Biz öğrenci olarak tanımlayacağımız bu struct için Ad,Soyad,Sınıf ve No bilgilerini tutan bir yapı oluşturalım.

Struct Örneği

```
struct Ogrenci
{
    public string Ad;
    public string Soyad;
    public char Sinif;
    public int No;
}
```

Oluşturmuş olduğumuz bu struct' ı kullanmak için instance almamız gerekiyor.

Strunct'dan instance almak

```
Ogrenci ogr = new Ogrenci();
```

İnstance alma işleminden sonra oluşturmuş olduğumuz değişken isminin sonuna nokta(.) operatörü koyarak struct içinde tanımlamış olduğumuz özelliklere erişip eklemeler ve düzenlemeler yapabiliriz.

İnstance Alınan struct'ımızı dolduruyoruz

```
ogr.Ad = "Mert";
ogr.Soyad = "KURT";
ogr.No = 1;
ogr.Sinif = 'A';
```

Static Class, Static Member

Static olarak işaretlenen bir class veya classın içerdiği bir elemanı proje çalıştırıldığında doğrudan RAM bellek üzerine çıkartılır. Static olarak işaretlenmiş bir yapıyı kullanmak için instance alma işlemeni yapmak zorunda kalmayız.

Instance alma işlemi kullanılacak yapının RAM bellek üzerine çıkartılmasını sağlar. Fakat static olarak işaretlenmiş bir yapı proje çalıştığında zaten RAM bellek üzerine çıkartılacağı için instance alma işlemine gerek kalmamaktadır.

Static olarak işaretlenmiş ve .NET Framework ortamında tanımlı olan class'lar da vardır. Bu class'lara örnek vermek gerekirse;

- MessageBox Sınıfı
- Math Sınıfı

Inheritance

Inheritance (miras alma, kalıtım), bir nesnenin özelliklerinin farklı nesneler tarafından da kullanılabilmesine olanak sağlayan OOP özelliğidir. Yazılan bir sınıf bir başka sınıf tarafından miras alınabilir. Bu işlem yapıldığı zaman temel alınan sınıfın tüm özellikleri yeni sınıfa aktarılır.

Örneğin; İnsan – memeli ilişkisinde, insanın memeli sınıfını miras aldığı söylenebilir. Bu sayede insan sınıfını yazarken memelilerin özelliklerini tekrar yazmamıza gerek kalmaz. Elinizde bir taşıt sınıfı varsa; otomobil, kamyon, motosiklet gibi alt sınıfları üretmek çok daha az çaba gerektirir.

Önemli Not

C# dilinde sadece tek bir sınıftan miras alabilirsiniz.

Inheritance'ı bir örnekle açıklayacak olursak, örnek olarak bir uçak oluşturmayı düşünelim. Bizim uçakların ana özelliklerini barındıracak bir class'a ihtiyacımız olacaktır. Bu class'ı aşağıdaki gibi oluşturabiliriz.

Nesnemizi oluşturuyoruz

```
public class Ucak
{
    public double Agirlik { get; set; }
    public double Uzunluk { get; set; }
    public double Yukseklik { get; set; }

}
```

Daha sonra ise oluşturduğumuz bu class tan yararlanarak uçak markalarımızı oluşturabiliriz.

İnheritance Örneği

```
//Ucak Class'ımızdan inheritance(kalıtım) alıyoruz
public class Airbus:Ucak
{
    public string Model { get; set; }
}

//Ucak Class'ımızdan inheritance(kalıtım) alıyoruz
public class Boing:Ucak
{
   public int Model { get; set; }
}
```

Bu şekilde markalarımızı da oluşturduktan sonra her iki sınıfımızdan instance aldığımızda kalıtım aldığımız Ucak class'ımızın propertylerine(özelliklerine) ulaşabiliriz.

Nesnelerimizin özelliklerini dolduruyoruz

```
//Airbus Class'ımızdan instance alarak özelliklerini dolduruyoruz
Airbus marka1 = new Airbus();
marka1.Agirlik = 1200;
marka1.Uzunluk = 19.20;
marka1.Yukseklik = 9.35;
marka1.Model = "A380";


//Boing Class'ımızdan instance alarak özelliklerini dolduruyoruz
Boing marka2 = new Boing();
marka2.Agirlik = 1200;
marka2.Uzunluk = 19.20;
marka2.Yukseklik = 9.35;
marka2.Model = 747;
```

Polymorphism

Polymorphism, bir metodun farklı sınıflar için farklı etkiler oluşturmasını sağlayan yöntemdir.

Örneğin; bir firma için hazırladığımız programda;

- Müşteriler, firmaya sipariş verebilir.

- Firma, tedarikçilere sipariş verebilir.

Bu iki işlem için oluşturulacak metotlar aynı işi, farklı yöntemler ile yapmaktadır. Bir tanesi bizim veri tabanımıza müşteriden alınan siparişi kayıt ederken, diğer metot stoğumuz azaldığında tedarikçiye bilgi göndererek ürün siparişi verir.

Örneğimizde bir sipariş class'ımız bulunmaktadır. Sipariş işlemini hem müşteri, hem de firma yapabilir. Biz sipariş class'ımız da oluşturduğumuz SiparisVer() metodunu farklı iki class da kullanacağız.

Polymorphism Örneği

```
public class Siparis
{
      public virtual string SiparisVer()
      {
          return "Sipariş verme işlemi için SiparisVer metodunu tanımlamanız gerekmektedir.";
      }
}

public class Musteri : Siparis //Siparis Class'ın dan SiparisVer metodunu miras alıyoruz.
{

    public override string SiparisVer()
    {
        return string.Format("Müşteri, firmamıza başarıyla sipariş verdi.");
    }

}

public class Firma : Siparis //Siparis Class'ın dan SiparisVer metodunu miras alıyoruz.
{

    public override string SiparisVer()
    {
        return string.Format("Firma, tedarikçiye başarıyla sipariş verdi.");
     }

}
```

Önemli Not

Miras verme işleminden sonra, Siparis classı içerisinde tanımlanan SiparisVer() metodu virtual olarak işaretlenir. Virtual olarak işaretlenen metotlar miras aldığı classlar da ezilebilir olur. Biz SiparisVer() metodunu ezerek istediğimiz işlemleri yapabiliriz.

Yukarıdaki kodu çalıştırdığımızda farklı classlar için SiparisVer() metodunu çağırdığımızda çağırdığımız class içerisindeki metot çalışacaktır.

Abstract Class

Abstract Class, ortak özellikli Class'lara Base Class(Ana Sınıf) olma görevini üstlenir. Bu nedenle Abstract Class'dan nesne türetilemez.

Abstract öğelerin amacı, kendisinden kalıtım alan sınıflarda bir takım özelliklerin kullanılmasını zorunlu kılmaktır. Eğer bir abstract içerisinde abstract bir öğe tanımladıysanız o öğeye bir kod gövdesi belirtemezsiniz.

Önemli Not

- Değişkenleri abstract olarak işaretleyemeyiz.
- Bir abstract öğeyi bir sınıfa yerleştirebilmek için o sınıfın mutlaka abstract olması gerekmektedir.

ENUM

Enum Nedir?

Program içersinde kullanılan sabitlerin anlamdırılması ile sabitlere isim vererek bir grup altında toplamamızı sağlar. Bu gruplara enum (enumeration - numaralandırma) denir.

Örnek ile açıklamamız gerekirse;

Enum Örneği

```
//Gün isimlerini barındıran bir enum tanımlıyoruz
public enum Gunler
{

    Pazartesi,
    Salı,
    Çarşamba,
    Perşembe,
    Cuma,
    Cumartesi,
    Pazar

 }
```

Tanımlamış olduğumuz bu Enum'ı kullanmak istediğimizde ise aşağıdaki gibi bir yol izleyebiliriz.

Enum Kullanımı

```
//Sonuç olarak bize MessageBox içerisinde Çarşamba sonucu gelecektir.
Gunler gunAdi;
gunAdi = (Gunler)2;
MessageBox.Show(gunAdi.ToString());
```

INTERFACE

Interfacelere Türkçe karşılığı olarak arayüz dememiz yanlış olur, Interfaceler C# tarafında yetenek olarak tanımlanabilir.

Interface Nedir?

Interfaceler, class veya struct gibi türler için oluşturulmuş modellerdir. Bir sınıfın temelde hangi üyelerden oluşacağını belirleyen şablon yapılarıdır. Bu sayede oluşturulacak sınıflara öncülük edilir ve içermeleri gereken üyelerin ne olacağının standardı belirlenir. Kullanılması zorunlu olan metotlar belirlenir fakat metot içerikleri doldurulmaz. Simülasyon metot olarak hazırlandıktan sonra, yetenek olarak kazandırıldıkları Class'a bu metotlar implament edilir. Metot içerikleri class içerisinde doldurulur.

Interface yapılarını interface anahtar sözcüğü ile tanımlarız. Genellikle interface isimleri "I" harfi ile başlar, tabi bu bir zorunluluk değildir. Fakat bu şekilde kullanılması standart olarak kabul edilir. Burada ki 'I' harfi Interface'in ilk harfidir.

Inheritance yapısında bir class'ın bir base class'ı bulunmaktaydı. Bir class'a birden fazla classdan kalıtım bırakmak istediğimizde hiyerarşik bir yapı oluşturuyorduk. Interfaceler de bu durum geçerli değildir. Bir class'a birden fazla interface'den yetenek kazandırabiliriz.

Örnek olarak kendimize bir Taşıt interface i yaratıp bu interface'i yetenek olarak kullanan sınıflar oluşturabiliriz.

Interface Oluşturuyoruz

```
public interface ITasit
{

    string Marka { get; set; }
    string Model { get; set; }
    int MotorGucu();
```

```
        string YakitTuketimi();
        byte KapiSayisi();

}
```

Interface Kullanımı

```
public class Mercedes : ITasit
{

        public string Marka { get; set; }

        public string Model { get; set; }

        public byte KapiSayisi()
        {
            return 4;
        }
        public int MotorGucu()
        {
            return 2000;
        }
        public string YakitTuketimi()
        {
            return "5.3 Lt";
        }
}


public class Ford : ITasit
{

        public string Marka { get; set; }

        public string Model { get; set; }

        public byte KapiSayisi()
        {
            return 4;
        }
        public int MotorGucu()
        {
            return 1600;
        }
        public string YakitTuketimi()
        {
            return "4.5 Lt";
        }

}


public class Porsche : ITasit
{

        public string Marka { get; set; }
```

```
        public string Model { get; set; }

        public byte KapiSayisi()
        {
            return 2;
        }
        public int MotorGucu()
        {
            return 4500;
        }
        public string YakitTuketimi()
        {
            return "6.7 Lt";
        }

}
```

DELEGATE VE EVENT

Delegate Nedir?

Delegate'ler bir olay gerçekleşirken birden fazla metotun çağırılmasını sağlayan yapılarımızdır. Örneğin kullanıcı bir düğmeye tıkladığında birden fazla method'un otomatik olarak çağrılmasını istiyorsunuz. Bunun için delegate kullanabilirsiniz.

Deletegate Oluştururken izlenecek adımlar;

- 1. Tanımlama-Declare
- 2. Örneklenme-Instance
- 3. Çağırma - Invoke

Delegateler başka fonksiyonlara aracılık eden özel fonksiyonlardır.

Delegate olarak bir method tanımlanır. Bu method aslında temsilcidir. Bir olay olduğunda siz delegate method'unu çağrırsınız. Delegate method'una kendisini ekleyen method'lar otamatik olarak çağrılırlar. Aşağıda basit bir delegate methodu yaratıyoruz.

Delegate Oluşturmak

```
public delegate int MyDelegate(int Sayi1, int Sayi2);
    //Delegate içerisinde kullanılacak metot.
static int Carp(int sayi1, int sayi2)
{
```

```
    return sayi1 * sayi2;
}
```

Delegate Kullanımı

```
//Delegate' in instance' ını oluşturalım.
MyDelegate delege = new MyDelegate(Carp);

//Deleagate Çağıralım ( invoke)
int iCarpim = delege(3, 4);

MessageBox.Show(string.Format("Çarpım Sonucu :{0}", iCarpim));
```

Önemli Not

Delegatelerin asıl kullanım amacı Frameworklerde kullanılıyor olmasıdır.

Event Nedir?

Events (Olaylar) kullanıcının fare ile tıklaması, klavyeden bir tuşa basma gibi işletim sistemi üzerinden gerçekleştirdiği eylemlerdir.

Bir bileşen üzerinde meydana gelen olayları takip eden ve bunları yakalayan mekanizma event handler olarak adlandırılır. Event Handler ilgili olay gerçekleştiği zaman tetiklenir. Olay gerçekleştikten sonra hangi işlemlerin yapılacağı olay yöneticileri için delegateler ile temsil edilir. Bir kullanıcının fare ile tıklaması örneğin bir Click olayıdır, bir tuşa basma olayı bir Keypress olayıdır ve bu olayların sonucunda nelerin yapılması gerektiğini biz kodlarımızla belirtebiliriz.

C#'da en çok kullanılan eventler ve açıklamaları aşağıdadır.

KONTROL	EVENT	AÇIKLAMA
Tüm Kontroller	Click	Kullanıcının kontrole tıkladığı zaman aralığında tetiklenen eventtir.
Tüm Kontroller	DoubleClick	Kullanıcının kontrole çift tıklığı zaman aralığında tetiklenen eventtir.
Tüm Kontroller	MouseClick	Kullanıcının kontrole mouse ile tıkladığı zaman aralığında tetiklenen eventtir.
Tüm Kontroller	MouseDoubleClick	Kullanıcının kontrole mouse ile çift tıkladığı zaman aralığında tetiklenen eventtir.
Tüm Kontroller	MouseMove	Kullanıcının ilgili kontrol üzerinde fare imlecini hareket ettirdiğinde tetiklenen eventtir.
Tüm Kontroller	MouseEnter	Kullanıcının ilgili kontrol üzerine mouse ile geldiğinde tetiklenen eventtir.
Tüm Kontroller	MouseLeave	Kullanıcının kontrol üzerinden ayrılındığında tetiklenen eventtir.
Tüm Kontroller	MouseDown	Mouse tuşuna basıldığında tetiklenir. Click olayından önce çalışan bir olaydır.
ComboBox, Listbox, ListView	SelectedIndexChanged	Seçili index değiştiğinde tetiklenen eventtir.

COLLECTİON KULLANIMI

List Kullanımı

List sınıfı ArrayList sınıfı gibi içerisinde object tipinde veri saklayabilen bir yapıdır. Aralarındaki en büyük fark ArrayList sınıfı içerisine eklenen her türlü veriyi object tipinde saklarken List yapısına saklayacağı verilerin tipi bilgi olarak verilebilir. Bu şekilde List sınıfı içerisindeki verilerin tipleri bilindiği için her hangi bir tip dönüştürme işlemi kulanılmadan direk işleme sokulabilir.

List Tanımlama Yöntemi

```
//Kullanımı;

  List <T> = new List<T>

//T => istenilen veri tipinin yazılacağı bölüm.
```

Bir örnekle açıklamak istersek öncelikle Araç isimli bir Class(sınıf) yaratıyoruz.

Nesne Oluşturuyoruz

```
class Arac
{
        public string Marka { get; set; }
        public string Model { get; set; }
        public int UretimYili { get; set; }
        public string YakitTuketimi { get; set; }
}
```

Daha sonra oluşturduğumuz Arac Class'ı tipinde bir liste tanımlıyoruz.

List Oluşturuyoruz

```
List<Arac> Araclar = new List<Arac>();
```

Arac Class'ımızdan instance alarak birkaç adet yeni araçlar yaratıyoruz.

Nesnemizden Instance alarak özelliklerini dolduruyoruz

```
Arac araba1 = new Arac();
araba1.Marka = "Mercedes";
araba1.Model = "CLS AMG";
araba1.UretimYili = 2017;
araba1.YakitTuketimi = "6,75 Lt";


Arac araba2 = new Arac();
araba2.Marka = "Ford";
araba2.Model = "Focus";
araba2.UretimYili = 2017;
araba2.YakitTuketimi = "4,50 Lt";


Arac araba3 = new Arac();
araba3.Marka = "Renault";
araba3.Model = "Megane";
araba3.UretimYili = 2017;
araba3.YakitTuketimi = "4,20 Lt";
```

Yaratmış olduğumuz bu araçları .Add() metotunu kullanarak tanımlamış olduğumuz listeye ekliyoruz.

Oluşturduğumuz nesneleri list'imize ekliyoruz

```
Araclar.Add(araba1);
Araclar.Add(araba2);
Araclar.Add(araba3);
```

Dictionary

List yapısına benzer bir yapı kullanmaktadır. Tek fark Key ve Value değerlerinin generic olmasıdır. Her bir veri için iki adet boxing işleminden kurtulmak demektir. Bu da bize çok fazla performans artışı sağlamaktadır.

Küçük bir örnekle açıklayacak olursak,

```
Dictionary<int, string> iller = new Dictionary<int, string>();

iller.Add(1,"Adana");
iller.Add(6, "Ankara");
iller.Add(34, "İstanbul");
iller.Add(35, "İzmir");
iller.Add(55,"Samsun");

// foreach ile Dictionary olarak tanımladığımız listenin value larında dönerek illerin isimlerini tek tek
MessageBox'ımıza yazdırabiliriz.
foreach (string gelenVeri in iller.Values.ToList())
{

    MessageBox.Show(gelenVeri);

}
```

IEnumabrable

Bilmemiz gereken en önemli arayüzdür ve collection isim alanının en üstündedir. IEnumerable yapısı foreach yapısının temelidir.

CUSTOM EXCEPTİON

Heralde yazmış olduğumuz uygulamada bir exception meydana gelmesi en istemediğimiz şeydir. Ama bazen bilinçli olarak uygulamamızda exception almak(fırlatmak) isteyebiliriz.

Exception uygulama çalışırken runtime da meydana gelen hatadır. Exception handling ise runtime da meydana gelen bu exception'ları yazılım tarafında ele alma tekniğidir diyebiliriz. Uygulamanızda bir exception meydana geldiğinde .NET Framework kütüphanesinin direkt olarak fırlattığı stack trace açıklaması veya tuhaf exception mesajını kulanıcıya(client'a) göndermek istemeyiz. Bu gibi durumlar için Custom Exception kullanımı devreye giriyor. Custom Exception kullanarak daha yönetilebilir ve daha anlamlı hata mesajları tanımlayabilir client'a dönebiliriz veya Log işlemlerinde handle ederken exception türüne göre daha farklı loglama işlemleri yapmak isteyebiliriz.

.Net tarafında bütün exception'ların base'i System.Exception class'ı dır ve custom tanımlanan exception class'larıda direkt veya dolaylı olarak bu class'tan inherit(Inheritance) olmaktadır.

Custom Exception Nesneleri Oluşturmak

Login işlemlerinde hata exception fırlatmak için kullanılacak LoginException adında bir custom exception tanımlayalım.

Custom Exception Nesnesi Tanımlıyoruz

```
public class LoginException : System.Exception
{
    //işlemler
}
```

Şimdi uygulama içerisinde kullanımı için gerekli constructor tanımlamalarını yapalım.

Custom Exception Nesnesi Oluşturuyoruz

```
public class LoginException : System.Exception
{

public LoginException()
: base()
{ }

public LoginException(String message)
: base(message)
{ }

public LoginException(String message, Exception innerException)
: base(message, innerException)
{ }

protected LoginException(SerializationInfo info, StreamingContext context)
: base(info, context)
{ }

}
```

Yukarıda görüldüğü üzre LoginException'nın constructor'larını tanımaldık ve System.Exception class'ı için yapılacak constructor yönlendirmelerini yaptık, böylece LoginException class'ını kullanıma hazır hale getirdik.

Custom Exception Kullanmak

Yazmış olduğumuz LoginException'ı kullanırken aşağıdaki gibi try/catch içerisinde handle edip kullanabiliriz.

Custom Exception Kullanımı

```
void Login(string userName, string password)
{

        try
        {
            if (userName != "Mert" && password != "123456")
                throw new LoginException("Oturum Açma İşlemi Başarısız");
        }
        catch (LoginException loginException)
        {
            MessageBox.Show(loginException.Message);
        }


}
```

İhtiyaç dahilinde çok daha farklı case'lerde kullanılmak üzre Custom Exception'lar tanımlayabilirsiniz. Exception fırlatmak düz mantıkla düşünüldüğünde tuhaf gelebilir ancak çoğu projelerde hayat kurtarır ve yazmış olduğunuz kodları tek bir yerden yönetebilmek için tercih edebilirsiniz ancak bu exception'ları handle etmeyi unutmamalıyız.