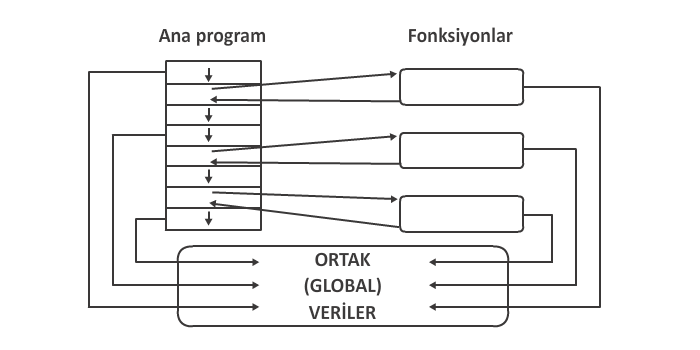
**Nesne Yönelimli Programlama (Object Oriented Programming)**

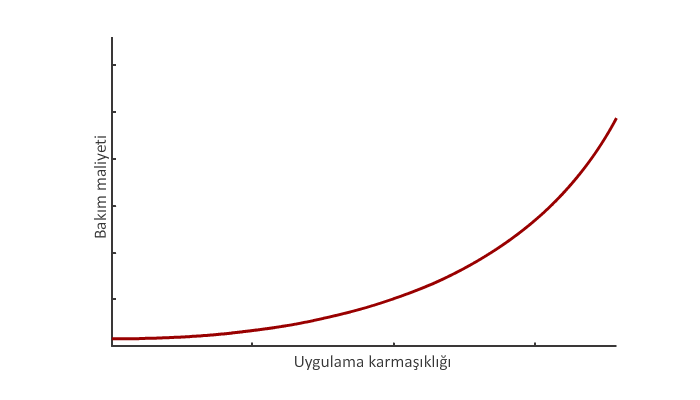
OOP 1960’lardan bugüne yazılım dünyasını etkisine almış bir metodolojidir. Nesne yönelimli programlama C# dışında C++, Java, Objective C, Python vb gibi programlama dillerinde kullanılabilir.**1970** yılından bugüne kadar geliştirilen birçok dil OOP desteğine sahiptir.

OOP temel olarak, ortaya çıktığı güne kadar süregelen programlama mantığını kökten değiştirmiştir. OOP’den önce kullanılan yazılım metodolojisi,***Prosedürel Programlama*** adı ile anılır. Bu metodoloji, belirli bir yönde ilerleyen kodlar ve iş yükünü hafifletmek için ortak işlerin yüklendiği fonksiyonların çağırılması esasına dayalıydı. Geniş ve kapsamlı olan programları küçük parçalara bölerek çözümleme yöntemidir. Her procedure ya da fonksiyonda kullanılan değişkenler, metotlar birbirinden bağımsızdır.



Bu yöntem, yazılım dünyasının uzun süre işini görse de, bazı sıkıntıları da içinde barındırıyordu:

* Geliştirilen uygulama **parçalanamayan bir bütün** halindeydi. Bu yüzden, uygulama üzerinde çalışan her geliştirici; uygulamanın hemen her yapısına hâkim olmalıydı.
* Bu durum nedeniyle, projelere yeni **yazılımcıların katılması** önemli **bir adaptasyon süreci** gerektiriyordu.
* Uygulama tek bir bütün halinde olduğu için**, ufak değişiklikler uygulamanın farklı noktalarında büyük sorunlara** yol açabiliyordu.
* Yıllarla birlikte **müşteri ihtiyaçları ve donanım kabiliyetleri** arttı. Bunun getirisi olarak da, geliştirilen **uygulamaların kapsamları ve boyutları büyüdü**. Bu aşamada, yukarıda bahsedilen problemler gittikçe artmaya başladı. Başlanan projelerin çoğu istenen sürelerde yetiştirilememeye ve geliştirme zorluklarından ötürü iptal olmaya başladı.
* **Uygulama için ayrılan zaman, para** giderek **artmaya** başladı.



İşte tüm bu sıkıntıların aşılmasında OOP etkili olmuştur. Yazılım dünyasında bu çıkmazın aşılması OOP ile sağlanmıştır. 1960’larda OOP fikrini ilk ortaya atan Alan Kay, önerdiği metodolojiyi şu şekilde ifade etmiştir:

* Uygulama, nesneler ve onların ilişkileri çerçevesinde belirli bir iş yapmak için geliştirilebilmelidir.
* Her nesnenin bir sınıfı olmalıdır ve sınıflar nesnelerin ortak davranışlarını ifade etmelidir.
* Nesneler birbirleri ile iletişime geçebilmelidir.

Gerçek dünya modellemesiyle anlatılmak istenen şudur:

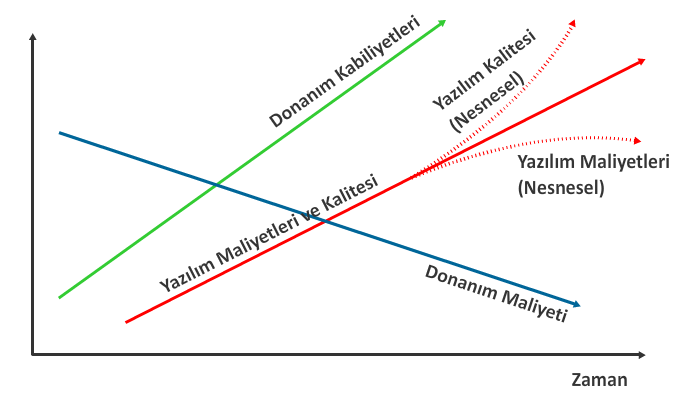
Bir fabrika örneğini ele alalım. Bu fabrikada işçiler, makineler gibi birçok nesne bulunur ve bu nesnelerin ilişkisi çerçevesinde fabrika çeşitli işler yapıp çıktılar üretebilir.

OOP ile programlama mantığında da, bu örnektekine benzer şekilde program kurgulanır. Çeşitli nesneler geliştirilip birbirleriyle ilişkilendirilerek, belirli amaçlara hizmet eden uygulamalar geliştirilir.

Bu yapının önemli getirileri şunlardır:

* Yazacağınız sınıflar birbirinden bağımsız olarak geliştirilebilir. Bu sayede program *böl, parçala, fethet* mantığı çerçevesinde çok kolay bir şekilde parçalanır ve her parça ayrı ayrı ele alınabilir. OOP mantığında gerçek dünya algılayışı temel alındığı için bu, anlaşılması çok daha kolay bir yapıdır.
* OOP, yapısı gereği kod tekrarlarının önüne geçer ve bu durum, özellikle ilk dönemlerde yazılımcıların hızlı bir şekilde OOP yapılarına geçmesinin temel nedenlerinden biri olmuştur.

Yukarıda değinilen getiriler, projelerin yönetilebilirliğini büyük miktarda artırdığı için daha büyük projeler çok daha az çaba ile yönetilebilir hale gelmiştir. Yine aynı getiriler sayesinde, projeler rahat bir şekilde büyütülebilmiştir. Alttaki grafikte noktalı çizgilerle gösterilen bölümler OOP’nin etkileridir. Yani, OOP ile yazılım kalitesi artmış, bunun yanısıra yazılım maliyetleri düşmüştür.



# OOP özellikleri

* Encapsulation (sarmalama/paketleme),
* Inheritance (miras alma)
* Polymorphism’dir (çok biçimlilik).

# Sarmalama / Paketleme (Encapsulation)

Sarmalama / Paketleme (Encapsulation), bir nesnenin özelliklerinin dışarıdan kullanılmasını sınırlamak için kullanılır. Bir insanı örnek verelim. Vücudumuzun yaptığı işler sarmalanmıştır ve biz buna müdahale edemeyiz. Örneğin, biz sadece solumakla ilgileniriz; ciğeri şişirmek, kana oksijen karıştırmak, kirli kanı temizlemek gibi işlemler içeride yapılıyor olsa da, bizim için durum sadece solumaktan ibarettir. Diğer işlemler sarmalanmıştır. Aynı mantıkla, nesnelerimizin de bazı işleri kendi içinde yapması ve bu sayede daha kolay kullanılabilmesi sağlanır böylece veri güvenliği de sağlanmış olur.

Bu işlem yapılırken esasen access modifier’larla işlemlere ne şekillerde erişileceğini belirlemeniz yeterlidir.

İnsan örneğini kod halinde ifade edelim:

**public** **class** **Insan**

{

**private** **void** **KirlikandakiKarbondioksitiAl**()

    {

        //.......

    }

**private** **void** **OksijenikanaVer**()

    {

        //.......

    }

**private** **void** **CigeriSisir**()

    {

        //.......

    }

**private** **void** **CigeriDaralt**()

    {

        //.......

    }

**public** **void** **NefesAl**()

    {

        CigeriSisir();

        OksijenikanaVer();

        KirlikandakiKarbondioksitiAl();

        CigeriDaralt();

    }

}

Bu örnekte de görüldüğü gibi, Insan sınıfının sadece NefesAl metodu **public** olduğu için dışarıya açıktır. Bu nesneyi kullanan yapılar asla diğer metodlara erişemez, istese de değişikliğe sebebiyet veremez.

Aynı durumu daha gerçekçi bir örnekle inceleyelim:

**public** **class** **TarihBilgisi**

{

**private** DateTime tarih;

**public** **string** Gecerlitarih;

    {

**set** { tarih = Convert.ToDateTime(**value**) }

    }

**private** **double** **GunOlaraktarihFarki**(DateTime farkAlinacaktarih)

    {

**return** (farkAlinacaktarih - tarih).TotalDays;

     }

// Yukarıda yapılan işlemi incelediğimizde Gecerlitarih isimli property’e atılan değer dönüştürülerek DateTime tipindeki tarih isimli değişkene atılmaktadır. GunOlaraktarihFarki isimli metod parametre olarak almış olduğu farkAlinacaktarih isimli değişkenden tarih isimli değişkeni çıkarmaktadır. Kullanılan TotalDays özelliği çıkarılan iki tarih arasındaki farkı gün olarak hesaplamaktadır.

**public** **string** **GunFarki**(**string** Yenitarih)

    {

**return** GunOlaraktarihFarki(Convert.ToDateTime(Yenitarih)).ToString() + " gün fark var";

    }

}

Bu örnekte de görüldüğü gibi, tarih alanı (field) ve GunOlaraktarihFarki metodu **private** yapılarak dışarıdan gizlenmiştir. TarihBilgisi nesnesi ile çalışan bir uygulama bu alandan ve metottan haberdar olmayacak ve sadece bizim izin verdiğimiz yapılara erişerek işlerini görecektir. Kısıtlama kötü gibi görünse de, aslında bu sayede gereksiz alanlarla uğraşmasına gerek kalmamış ve değiştirmemesi gereken yerleri değiştirme şansı kalmamıştır.

# Miras Alma / Kalıtım (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.

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.

C# dilinde sadece tek bir sınıftan miras alabilirsiniz.

Örnek:

**public** **class** **Canli**

{

**public** **int** ayak;      // Canlının kaç ayağı olduğunu tutan field

**public** **int** el;        // Canlının kaç eli olduğunu tutan field

**public** **string** tur;    // Ne tür bir canlı olduğunu belirleyeceğimiz field.

}

Şimdi iki adet daha sınıf oluşturarak Canli sınıfından türetelim:

**class** **Insan** : **Canli**

{

**public** **void** **Konus**()

    {

        Console.WriteLine(“Selam benim {0} adet elim ve {1} adet ayağım vardır.”,el,ayak);

        Console.ReadLine();

    }

}

Yukarıda farklı kullanılan bir yapı gözükmektedir. **Console.WriteLine** metodunun daha öncelerde farklı kullanımını görmüştük. Bu kullanımda ise indeks mantığı ile değişkenler atanmıştır. Yani “{0}”**Console.WriteLine** metoduna girilecek metnin ardından gelen parametrelerin ilkini yani index’i sıfır olanı ifade etmektedir. Bu da metodun kullanımında el isimli değişkene denk gelmektedir. Diğeri olan “{1}” ise bir sonraki indeksteki ayak isimli değişkene denk gelmektedir.

**class** **Kopek** : **Canli**

​{

**public** **void** **Havla**()

    {

        Console.WriteLine(“Hav”);

        Console.ReadLine();

    }

}

Bu sınıfların kullanım örneği aşağıda verilmiştir:

**class** **CanliDemo**

{

**public** **static** **void** **Main**()

    {

        Insan ins=**new** Insan();

        Kopek kop=**new** Kopek();

        ins.el=2;

        ins.ayak=2;

        kop.ayak=4;

        ins.Konus();

        kop.Havla();

    }

}

Bu uygulamayı çalıştırdığınızda ekranda aşağıdaki çıktı görülür:

     Selam benim 2 adet elim ve 2 adet ayağım vardır.

     Hav

Görüldüğü gibi el ve ayak gibi özellikler canlıyı kalıtım alan sınıflarda bir daha yazılmamıştır. Insan ve Kopek sınıfları CanliDemo sınıfından türediği için bu özellikleri doğrudan edinmiştir.

**Çokbiçimlilik (Polymorphism)**

Çokbiçimlilik (Polymorphism*)*bir nesnenin farklı amaçlar için de kullanılabileceği anlamına gelir ve statik ve dinamik olmak üzere ikiye ayrılır.

* **Statik çokbiçimlilik**; metot ve operatörlerin aşırı yüklenmesi (overload) olarak belirtilir.
* **Dinamik çokbiçimlilik**; özet sınıflardan miras alma yoluyla işlemlerin gerçekleştirilmesi işlemine verilen isimdir.

Çokbiçimlilik konusunun daha iyi anlaşılması için, statik çokbiçimlilik türünden, metotların aşırı yüklenmesine yönelik bir çalışma yapacağız.

Öncelikle iki adet **enum** oluşturalım: Hareket ve CanliTip.

enumCanliTip

{

    Insan=0,

    Hayvan=1

}

enumHareket

{

    Havla=0,

    KuyrukSalla=1,

    PatiUzat=2

}

Ardından; üç adet **static void** metot oluşturalım.

**static** **void** **Konus**(CanliTip e,**string** ifade)

{

    Console.WriteLine("{0}der ki: '{1}'",e,ifade);

}

**static** **void** **Konus**(**string** ifade)

{

    Console.WriteLine(ifade);

}

**static** **void** **Konus**(CanliTip e , Hareket f)

{

    Console.WriteLine("{0}şunu yaptı: {1}",e,f);

}

Şimdi de, bu metotlarımızı projede kullanalım:

**static** **void** **Main**(**string**[] args)

{

    Konus(CanliTip.Insan,"ben bir insanım");

    Konus(CanliTip.Hayvan , Hareket.Pati);

    Konus("Hav");

    Console.ReadLine();

}

Uygulama çalıştırıldığında aşağıdaki gibi bir çıktı alınır:

* İnsan der ki:'ben bir insanım'
* Hayvan şunu yaptı: PatiUzat
* Hav

Sonuç olarak, çokbiçimlilik bir nesnenin bir işlemi farklı şekillerde yapabileceğini gösteren bir kavramdır. Birbirine benzeyen nesneleri ortak özellikleriyle ele alabilmenizi ya da bir nesnenin aynı işi farklı şekillerde yapabilmesini sağlar. Çokbiçimlilik diğer konulara kıyasla daha soyut kaldığı için örnekleri artırmakta fayda var. Aşağıdaki örnekte farklı bir statik çokbiçimlilik yapısını göreceğiz. Bu örnekte yazıcıdan farklı değerlerin çıktısını almak için mantıklı olarak tek bir metot aracılığıyla işlem gerçekleştireceğiz.

// Yazıcı sınıfı ile çıktısını almak istediğimiz değerleri tek bir metot ile farklı formatlarda çıkarttırabiliriz.

// Nedenle içine Yazdir isimli iki metot tanımlıyor ve aldıkları parametreleri değiştiriyoruz.

**public** **class** **Yazici**

{

**public** **void** **Yazdir**(**int** i)

{

Console.WriteLine("Sayısal değer yazdırılıyor - {0}", i);

}

**public** **void** **Yazdir**(**string** s)

{

Console.WriteLine("Metinsel değer yazdırılıyor - {0}", s);

}

}

**static** **void** **Main**(**string**[] args)

{

Yazici yazici = **new** Yazici();

yazici.Yazdir("Turkcell Geleceği Yazanlar");

yazici.Yazdir(1);

Console.WriteLine("İşlem tamamlanmıştır");

Console.ReadKey();

}

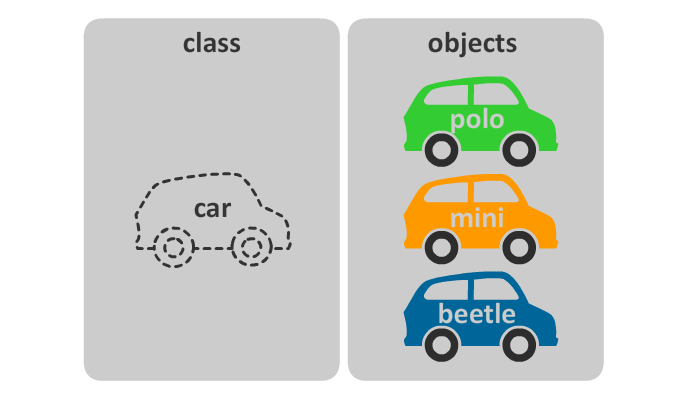
Yukarıdaki örnek gibi birçok farklı örnek çıkartabiliriz. Örneğin, okuma metodu olan bir Okuyucu sınıfı. Klavyeden, veritabanından veya farklı cihazlardan gelen bilgileri farklı parametrelere sahip aynı isimli metot ile okuyabiliriz.

# Sınıflar (Classes)

Sınıflar ve nesneler birbiriyle bağlantılıdır. Bir nesne bir sınıfın RAM’da yer teşkil etmiş halidir. Yani sınıf bir şablondur. (Bluesprint) Ondan new anahtar kelimesini kullarak instance aldığımızda Ram’de yerini alır.

OOP temelde nesneler ve onların ilişkisi üzerine kurulu bir yöntem olarak tanımlanmıştır.

* Nesneler sınıflardan türetilir ve yetenekleriyle yapabilecekleri sınıflarla belirtilir. Nesne-Sınıf ilişkisi için şöyle bir örnek verilebilir:



Programlarda da bu mantıktan hareket edilir. Bir düğme yapıp bunu kullanmak istiyorsanız, önce düğme sınıfını tanımlarsınız. Daha sonra da istediğiniz yerde düğme sınıfından örnekler alırsınız (instance almak). Mesela bir buton oluştururken Button btn = new Button(); şeklinde buton tanımlayabiliyoruz. Buton tanımlayabilmemizin nedeni, .NET Framework içinde tanımlanmış bir Button sınıfı bulunmasıdır.

İsterseniz, tanım yapmak yerine, doğrudan kütüphane içindeki sınıftan da nesne türetebilirsiniz (instance alabilirsiniz).

Alttaki örnekte, önce instance alınmış ve daha sonra, Text özelliğinin değeri belirtilmiştir. Sonra da, Click olayı olduğunda yapılacak işleri belirtmek için **yenibuton.Click** metodu oluşturulmuş ve işaret edilmiştir.

Button yenibuton = **new** Button();

yenibuton.Text = "Tıkla";

yenibuton.Click += yenibuton\_Click;

**private** **void** **yenibuton\_Click**(object sender, EventArgs e)

{

    Console.WriteLine("Merhaba");

}

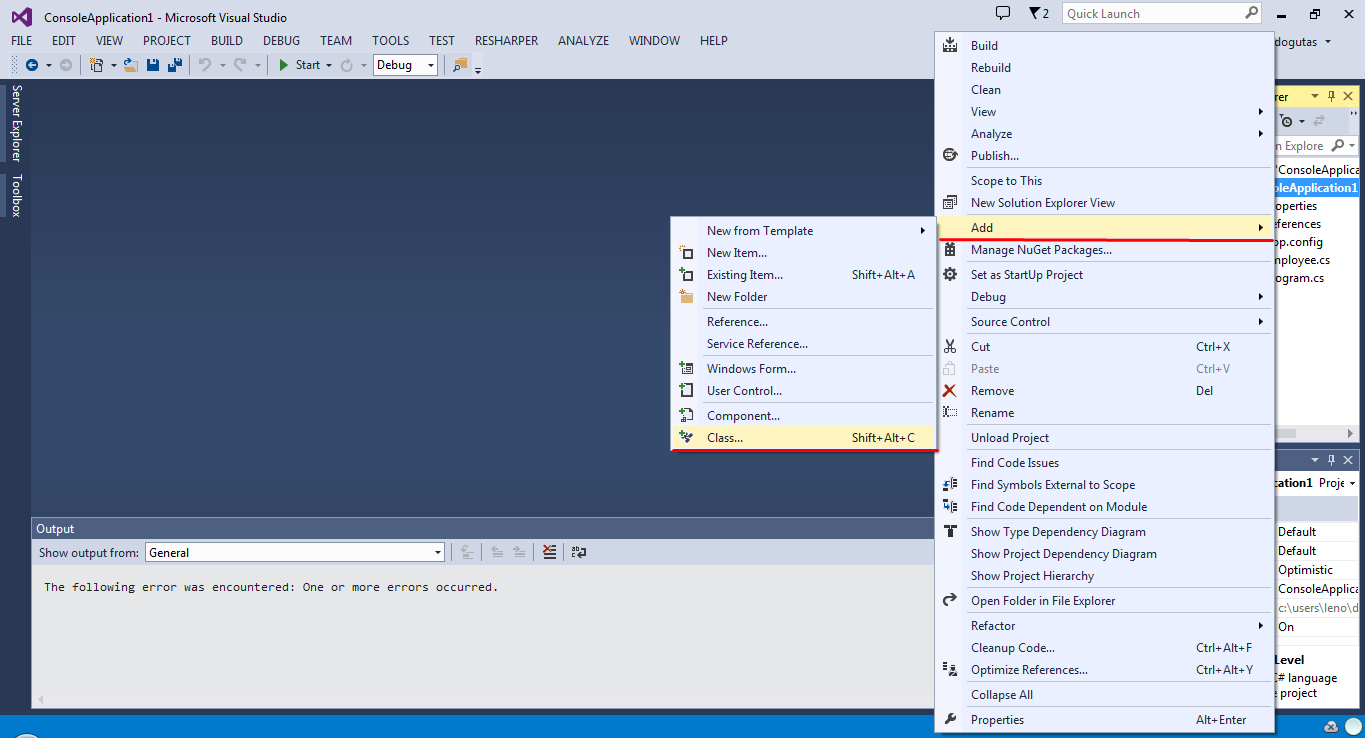
Birinci satırda, **Button** sınıfından bir örnek alınmıştır. İkinci satırda, oluşturulan yenibuton isimli **Button**nesnesinin **Text** özelliğine “Tıkla” yazısı atanır. Üçüncü satırda, **yenibuton.Click +=** yazılır ve klavyeden Tab tuşuna basıldığında eşittirin karşısına**yenibuton\_Click** yazılır. Bir kere daha Tab tuşuna basıldığında aşağıdaki butona Click edildiğinde çalışacak kod bloğunu oluşturacaktır.

Bu kod bloğunda parametre buton object, click işleminin argümanları **EventArgs**'e olarak taşınır.

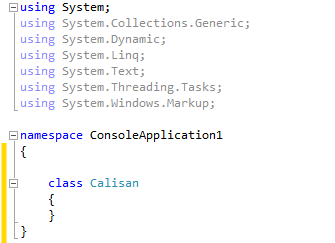
Bu sayede düğmenin geliştirilmesine olan gereksinim ve bu geliştirmenin uygulama üzerindeki etkileri ortadan kaldırılmış, projenin daha rahat bir şekilde yönetilmesi sağlanmıştır.

# Class Tanımlama:

Mesela C#’da kullandığımız Formlar da bir sınıftır. Her yeni ekran oluşturduğumuzda new ile nesne haline getirdik. Her sınıf .cs uzantılı dosyalarda tutulur.



Console uygulaması açıp Solution Explorer kısmından projenin üstünde Add-New Item sekmesinden Class seçeneğini seçiyoruz. Sonrasında dosyayı oluşturup isim veriyoruz. İsim verdikten sonra class’larımızı yani şablonlarımızı oluşturabiliriz.

Yukarısında belirtilen namespacelerin birçoğunu kullanmayacağız. Bizim için gerekli olan sadece System namespace’idir.

class Calisan

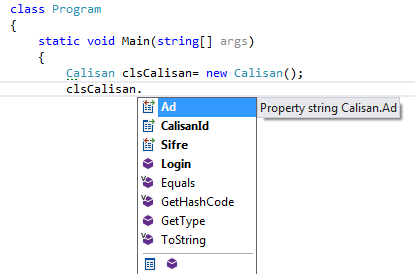
{

string Ad;

string Soyad;

string Bolum;  
}

Sınıfa ait birkaç özellik tanımladık. Nesneyi kullanabilmek için program.cs’de classta instance alalım.



Calisan clsCalisan= new Calisan();

clsCalisan.Ad = "Burcu";

clsCalisan.Sifre = "be";

Artık Ram’da yer kaplamaktaaynı zamanda clsCalisan isminde nersne oluşturur.

Visual studio’nun Intellisence özelliği sayesinde noktaişaretine bastığınızda sadece erişilebilecek üyeleri göreceksiniz.

**Sınıf İçindeki Yapılar**

**Class**

**Fields:** Değer tutan yapılardır.

**Properties:** Nesnenin özellikleri

**Methods:** Nesnenin yapabildikleri

**Events:** Nesnenin gerçekleştireceği olaylar

**Construction:** Nesne CLR'da ortaya çıkarken gerçekleşecek işlemler

**FIELDS:**

* Genellikle, nesnenin özellikleri (property) için değer saklama alanıdır. Varsayılan durumda private yapılardır.
* Field,  global alanda tanımlanmış bir değişken olarak da düşünülebilir. Yani class içerisindeki tüm metotlardan erişilebilecek bir değerdir.

**PROPERTIES:**

* Nesnenin özellikleridir.
* Kendi içinde iki metot barındırabilir:
  + **Set** metodu: Bir property’ye yeni bir değer atamak için kullanılır. Set metodu olmayan property’ler read-only durumundadır.
  + **Get** metodu: Bir property’nin değerini okumak için kullanılır

Genelde bir field’da bulunan değeri değiştirmek / okumak için kullanılır.

**METHODS:**

* Nesnenin yapabildiği işlerdir.
* **Yalın Methodlar:** Parametre almayan ve parametre geri döndürmeyen methotlardır.
* **Geri Dönüşlü Metotlar:** Herhangi bir veri tipini, herhangi bir tipten diziyi yada nesneyi döndürebilir. Void anahtar kelimesi geri dönüş değeri olmadığını yansıtır. Eğer geri döndürmek istersek geri döndürülecek tipi void yerine yazmamız gerekir.
* **Parametre alan Methotlar**
* Static olarak oluşturulabilir. Bu sayede, metot nesnenin değil sınıfın metotu haline gelir.

Static metotlar o metoda başka sınıflardan instance alınmadan ulaşılmasına olanak sağlar.

//Biz programımızda sonradan kullanabileceğimiz bir şablon tanımlıyoruz. Bu şablon sayesinde programa dahil ettiğimiz her çalışan için ayrı kod oluşturmayacağız. Bu şablondan örnek yaratıp kullanacağız.

class Calisan

{

//Auto propertyler çok fazla tercih edilmiyor. Tercih edildiği durumlar:

//İçerisinde işlem yaptığın durumlar: Mesela şifrelere kısıt koyma, yada kdv hesaplama

//Sadece okuma yapılacağı veya sadece yazma yapabileceği durumlarda kullanılır. Mesela empID sadece okunabilir bir property'dir

//Sadece okuma yapılacağı veya sadece yazma yapabileceği durumlarda kullanılır. Mesela empID sadece okunabilir bir property'dir

private int \_calisanId;

public int CalisanId

{

get { return \_calisanId; }

}

//Property

private string \_ad;

public string Ad

{

get { return \_ad; }

set { \_ad = value; }

}

//Fields

private string \_Soyad;

private string \_Bolum;

private string \_Giris;

//Property

private string \_sifre;

public string Sifre

{

get { return \_sifre; }

set

{

if (value.Length >= 5)

{

\_sifre = value;

}

else

{

throw new Exception("Şifre en az 5 karakter olmalıdır.");

}

}

}

//Metot

//Metot kullanıldığında geriye Evet ya da Hayır döndürecek. Metoda 2 parametre gönderiyoruz. İlk asama için karşılaştıracağımız değerleri biz girdik.

public string FullIsım( )

{

return Ad +" "+ Soyad;

}

public Boolean Login(string kullaniciAdi, string sifre)

{

if (kullaniciAdi == "burcu.engin" & sifre == "be")

{

\_calisanId = 1;

\_Bolum = "Yazılım Birimi";

Ad = "Burcu Engin";

return true;

}

else

{

return false;

}

}

}

}

**EVENTS**

Olaylar nesnelerin tepkileridir. Delegate’ler ile birlikte kullanılır. Düğmenin tıklanması bir event’tir. Daha önce düğmeyi tıklayınca bir tepki vereceği öngörülmüştür. Aşağıdaki örnekte, Kettle sınıfına SuKaynama Eventi eklenmiş ve su derecesi 100’ü geçerse bu event’in tetikleneceği belirtilmiştir. Artık Kettle nesnesi ile çalışırken su kaynayınca ne iş yapması gerektiğini söyleyebiliriz:

public delegate void SuKaynamaDelege();

public class Kettle

{

public event SuKaynamaDelege SuKaynamaEventi;

private byte \_susicakligi;

public byte Susicakligi

{

get

{

return \_susicakligi;

}

set

{

\_susicakligi = value;

//Katte'in içerisindeki su sıcaklığı 100 derece olduğunda su kaynatma eventini tetikler.

if (\_susicakligi >=100)

{

if (SuKaynamaEventi != null)

{

SuKaynamaEventi();

}

}

}

}

}

**Struct**:

Aynı classlar gibi kullanılabilirler. Yani Sınıflardan farkı belleğin heap bölgesinde değil stack’te tutulmasıdır. Bu yüzden classlara göre daha hızlıdırlar. İçlerinde Constructor,properties,field tanımlanabilinir.Kalıtım alamazlar. 16byte yer kaplar.

Örnek tanımlama:

struct calisan

{

public string adi;

public string soyadi;

public string adres;

}

Instance çağırma:

|  |
| --- |
| calisan kisi = new calisan(); |

Elemanların bilgilerini aktarma:

|  |
| --- |
| kisi.adi = "Burcu";  kisi.soyadi = "Engin";  kisi.adres = "İstanbul"; |

**Struct Özellikleri**

* Tanımlanan struct, tüm programda yani başka class ve formlarda kullanılmak istenirse public olarak tanımlanmalıdır.
* Struct içinde virtual metod tanımlanamaz.
* Constructor’ları parametresiz tanımlayamayız.  Constructor’ın aldığı parametre değerleri struct içindeki field’lara atanmalıdır. Parametre değerlerini property’lere atayamayız.
* Struct’lar herhangi bir class’tan kalıtım alamazlar. Sadece interface’den kalıtım alabilirler. Struct’lar aynı zamanda herhangi bir sınıfa kalıtım veremezler.