BU DÖKÜMAN *KAAN ASLAN* TARAFINDAN C VE SİSTEM PROGRAMCILARI DERNEĞİNDE VERİLEN *C# İLE NESNE YÖNELİMLİ PROGRAMLAMA* KURSUNDAKİ DERS İÇERİSİNDE TUTULAN NOTLARDAN OLUŞMAKTADIR. NOTLAR ÜZERİNDE HİÇBİR DÜZELTME YAPILMAMIŞTIR

C# Ders Notları

.net nedir?
Microsoft'un yeni kuşak uygulama geliştirme ve çalıştırma ortamıdır.

.net'in ilk resmi duyurusu 2000 yılında yapılmıştır. İlk versiyonu 2002 yılında çıkmıştır. Bu versiyonuna **.net Framework 1.0**'dır. 2003 yılında .net Framework 1.1 çıkmış bunu 2005 yılında **.net Framework 2.0** izlemiştir.

Not: Framework Türkçe karşılık olarak ortam, platform anlamında kullanılabilir.

C# nedir?

Köken olarak programlama dilleri bir birikimden gelirler. Programcılığın temelleri 1950 yıllarına kadar gider. C# köken olarak C ve C++ ailesinden gelir. Burda kullanılan diyez müzikte kullanılan diyezdir. C++ ın inceltilmiş halidir. C # Anders Heljsberg adlı bir programcı ve 4 arkadaşı microsoft için oluşturmuştur. Bu kişi Turbo C ve Delphi(Borland)ı derleyen kişidir. C# .net ortamı dikkate alınarak derlenmiştir. C# Java'ya oldukça benzer Java'nın iyileştirilmiş ve geliştirilmiş hali diyebiliriz. Javayla %80 benzerlik gösterir. Java zaten köken olarak C, C++ kökenlidir. C#, C++ a daha fazla yaklaştırılmıştır. C# public(açık) bir programlama dilidir. Microsoft'un malı değildir. C# ECMA (ECMA 335(European Computer Managment Assocation)) tarafından standardize edilmiştir. C# aynı zamanda ISO tarafından da standardize edilmiştir. C# en son standartları 2006 yılında oluşturulmuştur ve bu C# Language Specification 4.0 olarak adlandırılır.

C# temel olarak .net ortamı gözönünde bulundurularak tasarlanmıştır. C++ yeterli ve zor bir programlama dilidir. C# a bundan dolayı türetilmiş (productive) bir dildir ve hızla öğrenilip uygulamaya geçirilmesi daha kolaydır.

.net Ortamın Temel Özellikleri.

1. Ara Kod Sistemi: .net'te C# programını derlediğimiz zaman elde ettiğimiz kod ne intel'in ne motorola ne de diğer hiçbir işlemcinin ara kodlarını içerir. C# da x.cs olarak yazılan program .net Framework tarafından kullanılan makinanın koduna göre yorumlanıyor.

(C# Derleyicisi)
x.cs ⟶ x.exe

.net'teki en önemli unsur yorumlayıcısıdır. Normal olarak C, C++,Pascal vb diller doğal kod da çalışmaktadır. Yani derleyici kod, program yazılan makinanın işlemcisine uygun kod üretmektedir. Program Makine tarafından doğrudan çalıştırılmaktadır. Fakat .net Framework makine işlemcisine bakmadan ara kod sistemi uygulamaktadır. Bir C# programı derlendiği zaman elde edilen .exe dosyası içinde doğal Makine kodları değil fakat standardize edilmiş ara kodlar (Intermediate Language) bulunur. İçinde ara kod bulunan (ki her işlemciye göre bu arakodlar önceden microsoft tarafından .net framework için yazılmıştır) bir exe dosya çalıştırılmak istendiğinde .net ortamı devreye girer ve içindeki kodları uygulama yapılan işlemci diline yorumlayarak dolaylı bir şekilde çalıştırır.

.net'in içinde arakodu yorumlayarak çalıştıran mantıksal bölüme CLR(Common Language Runtime-Java da karşılığı Java Virtual Machine (JVM)) denir. Şüphesiz yorumlama sistemi doğrudan Makine dilinde çalışma sistemine göre performans olarak düşüklük gösterir. Microsoft'un tespitlerine göre bu performans kaybı % 18'dir. Fakat gelişen teknoloji içinde bu kayıp pek dikkate

alınacak bir kayıp değildir. Performans için sadece hız kritize edilirse bu sorun vardır. Yoksa diğer durumlarda avantajlar da olabilir.
.cs dosyasını Microsoft Intermediate Language(Ara kod) bir JIT(Just in Time) derleyici kullanarak çalıştırılabilir bir koda çevirir. Yani süreç şöyle gelişir. .**net** programı çalıştığı zaman CLR, JIT derleyicisini etkin kılar. JIT derleyicisi programın parçalarının herbirinin ihtiyacını temel olarak MSIL veya IL'i yerel dile dönüştürür.Ara kodun doğal koda çevrilme işlemine JIT işlemi denilmektedir.

Doğal kodlu programlar yalnızca tanımlı olduğu makinalarda çalışırken Arakodlu programlar işletim sisteminden bağımsız olarak her işletim sisteminde ve işlemci de çalışabilir. Buna **"Bir kere yaz her yerde çalıştır"** ilkesi denir. Buradaki konu .exe kodunun her yerde çalışabilirliğidir.

.net'in Linux taki karşılığı "Mono", Unix deki karşılığı "Rotor" dur.

Arakod sisteminde program bir kez derlenir ve .net ortamı oluşturulmuş her ortamda doğrudan çalıştırılabilir.

2. Diller Arası Entegrasyon (Language Interoperability): Bir program geliştirilirken değişik programlama dillerinde yazılabilir. Böyle bir projenin bir araya getirilmesi zor olabilir. Microsoft bu problemi çözmek için COM'u (Component Object Model) geliştirmiştir. Bu zor problem .net ortamında COM'a gerek kalmadan çözülüyor. Zaten .net ortamında bu dillerin ara kod karşılıkları yazılmıştır. Yani .net ortamında birden fazla programlama diliyle yazılmış programları bir arada çalıştırmak mümkündür. Oysaki COM sisteminde COM spesifikasyonu kullanılması zorunluydu. Bu çalışma sisteminde programcı COM belirlemelerine uygun bir şekilde programını yeniden düzenler, diğer kişiler de bu belirlemelere uyarak COM'u kullanırdı. Oysaki .net ortamında bütün dillerin ortak ara kod üretmesiyle diller arası kordinasyon basit ve doğal bir biçimde çözülmüştür.

.net için yazılan derleyiciler şunlardır. C#, VB.net, C++ (İki versiyonu var ISO'nun versiyonu, C++.net diyebileceğimiz .net ortamında kod üreten C++.net veya C++/CLI) Java # (Java Sharp) Microsoft bu dört dil için derleyici yazmıştır. Diğer yazılanlar da vardır.C# .net ortamı için yazılmış tam bir dildir. (Tencere Kapak veya Tornavida vida ilişkisi gibi)

VB .net: Microsoft'un Visual Basic dilinin .net'e uyarlanmış halidir. VB.net'ten önceki versiyonu VB 6.0'dır. Visual Basic 6.0 beynelminel bir dildir.

Microsoft'un 2 C++ derleyicisi vardır. Birincisi ISO C++'dır ve doğal kod üretir. İkincisi Microsoft'un .net ortamına uyumlandırması ile elde ettiği bir dildir. C++'ın .net derleyicisi ara kod üretmektedir.

J# Java'nın .net versiyonudur. Bu da Java Byte Code değilde Microsoft ara kod üretir.

Microsoft'un yazdığı bu derleyiciler dışında .net ortamı için başka dillerde uyumlandırılmaktadır.

Oysaki C# tam bu sistem için bu platform için üretilmiş bir dildir.

Bir dilin .net ortamına uyumlandırılması için o dilin bazı özelliklere minimal düzeyde sahip olması gerekir. Bu ortak özelliklere (ECMA 335 standardının 1. bölümünde belirtilmiştir) Common Language Spacification (CLS) denilmektedir.

Bu programların ara kod ortamında çalışmasında ciddi bir farklılık yoktur. Fakat C++ .net farklılığı hem doğal hem ara kod kodlamasının aynı ortamda üretilmesidir. C# saf ara kod da yazılırken diğerlerinde .net ortamında doğal kod da üretilebilir. Buna program yazılırken çok dikkat etmek

gerekir.

Bir program doğal kodla derlendikten sonra geriye dönmek pek mümkün değildir. C# da yazılan programlarda ne yazık ki bu mümkündür. Microsoft bu sorunu bir karıştırıcı üretip çözmeye çalışmıştır. Tamamen arakodda çalışmayarak ta bu sorunun bir nevi çözümü olabilir.

3. Ortak Sınıf Kütüphanesi: .net ortamında her dilin kullandığı ortak bir sınıf kütüphanesi vardır. Her dilin ayrı bir kütüphanesi yoktur. Önemli olan bu ortak sınıf kütüphanesinin ustaca kullanılmasıdır. % 50 oranında bir programcının iyi olup olmadığını bu kütüphanenin kullanımını belirler.

Microsoft'un bu sınıf kütüphaneleri kendi içinde pek çok bölüme ayrılmaktadır.
Örneğin kütüphanenin veri tabanı işlemlerinin yönelik sınıfların bulunduğu bölüme **ADO .net**, web sayfası oluşturmak için kullanılan sınıfların oluşturduğu bölüme **ASP .net**, pencereler, butonlar yani görsel ögelerin bulunduğu sınıfa **forms** denir. .net ortamı geliştirildikçe bu sınıf sistemi de büyümektedir.

4. Visual Studio IDE'si(Integrated Devolopment Environment): IDE kavramı ile derleyici kavramı sıklıkla birbirine karıştırılır. Ide bir derleyici değildir. Ide görsel bir biçimde angaryaları karşılayan bir yazılımdır. Asıl derleyici (Compiler) farklıdır. Ide derleyici değildir. Bize derleyici dışında her şeyi sunar. Ide derleme işlemini sunmaz ama derleyiciyi çağırır. Microsft'un C# derleyicisi **csc.exe** isimli programdır. Ide den derleyici çağrıldığında (compiler seçeneği) ide bu derleyiciyi çağırarak derleme işlemi yapılır. **csc.exe** siyah ekranda çalışan basit bir derleyicidir. Dünyanın en büyük programı csc.exe derleyicisi ve bir notepad kullanılarak yazılabilir.

Anahtar Notlar: .net denildiğinde framework'ün tamamına deniliyor. .net ürünün ismidir. Selpak Kağıt Mendil ilişkisi gibi. Framework sözcüğü microsoft'un oluşturduğu ve daha sonra ECMA ve ISO tarafından standardize edilen bu sistemin tamamını anlatmak için kullanılır.

Bu kursta .net Framework'ün karşılığı olarak ortam veya platform tabirleri kullanılacaktır. .net kavramı microsoft'un windows sistemleri için oluşturduğu ticari ismidir. Standartlarda .net diye bir söz geçmemektedir. Bu sistemi anlatmak için Standartlarda CLI(Common Language Infrastructer) resmi ismi kullanılmaktadır. Örnek olarak CLI'nin Linux versiyonu Mono BSD yani Unix versiyonu Rotor ticari isimleri ile adlandırılmaktadır. Fakat .net sözcüğü CLI sözcüğünün karşılığı olarak kullanılmaktadır.
Microsoft .net framework 1.0 içinVisual Studio 2002, .net framework 1.1 için Visual Studio 2003, .net framework 2.0 için Visual Studio 2005 idesini tasarlamıştır. Visual Studio ide'sinin henüz resmi sürümü yapılmayan Visual Studio 2008 (Orcas) tasarımı çıkacaktır. Visual Studio ürünü çeşitli ticari modellerle satılmaktadır. En geniş versiyonu Visual Studio Team versiyonudur. Windows Vista için Servise Pack 1 indirilecektir.

5. Güvenlik Mekanizması: . net güvenlik bakımından klasik sistemden daha iyidir. .net ortamı klasik ortamlara göre çeşitli bakımlardan daha güvenilir tasarlanmıştır. Çalışmakta olan bir programın sisteme zarar vermesi minimal düzeye indirgenmiştir. Bu sistemde casus yazılımların barınabilmesi çok zordur.

.net Ortamının Kurulması: .net framework visual studio ide'sini içeren bir kavram değildir. .net için yazılmış bir programın çalışmasını sağlayan tüm ögeleri içermektedir. Windows XP'nin son sürümleri ve Vista sistemlerinin hepsi .net ortamını zaten içermektedir. Microsoft bundan sonraki Windows sistemlerinin default olarak bu ortamı içereceğini beyan etmiştir. Fakat Windows 95, 98, 2000, Milennium ve XP'nin ilk sürümleri bu ortamı içermediği için bu sistemlere .net ortamı bilinçli olarak kurulmalıdır. .net ortamının farklı versiyonları aynı makinaya kurulabilir.

Şu anda .net ortamının kullanılan 2.0 versiyonudur. Tipik ide'si ise Visual Studio 2005'dir.

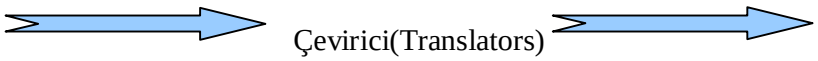
.net Ortamı birkaç biçimde kurulabilir.
1. İde kurulduğunda .net framework ortamı kurulur.
2. Microsoft .net sayfasından Microsoft Framework 2.0 SDK indirilerek kurulur. Bu kurulum yapıldıktan sonra sadece .net ortamı değil derleyici ve debugger gibi araçlarda kurulmaktadır.Windows zaten bedava ide'yi veriyor.
Microsoft web sayfasından Visual Studio ide'sinin daraltılmış bir biçimi olan Visual C# express indirilir.
Minimal düzeyde kurmak için: .net için yazdığımız bir programın .net ortamı olmayan bir bilgisayarda çalışabilmesi için gereken minimal kurulum için redistrubutable Package yüklenir. Bu paketin yaklaşık boyutu 22 MB'dir.

IDE ve Derleyici Kavramları

SDK: Service Devolopment Kit
csc.exe: Microsoft C# derleyicisi
CSC: C Sharp Compiler
Derleyicinin sade olmasının sebebi motor program olarak kullanılmasıdır. csc.exe derleyicisi komut satırından çalıştırılan basit bir ara yüze sahiptir.
IDE(Integrated Devolopment Environment): Uygulama geliştirmeyi kolaylaştıran çeşitli araçları barındıran bir programdır.
Ide'den derleme işlemi yapmak istediğimiz zaman ide csc.exe derleyicisini kullanarak derleme işlemi yapar. Aslında çok büyük projeler dahi bir notepad ve derleyici kullanılrak yazılabilir.

Programlama Dilleri, Gramer, Sentax ve Sematik: Programlama dilleri insanlar tarafından oluşturulmuş formal dillerdir. Bir dilin tüm kurallarına o dilin **grameri** denir. Doğal dilleri dikkate aldığımızda gramerin çeşitli alt birimleri vardır. Fakat programlama dillerinin grameri Sentax ve Semantik olarak incelenir.
Sentax: Bir dilin yazım kurallarıdır. Örneğin if deyiminin nasıl yazılması gerektiği sentax'ın alanı içindedir.
Semantik: Doğru yazılmış şeylerin ne anlam ifade ettiğini belirten kurallardır. Örneğin if deyiminin nasıl yazılması gerektiği sentax'ın konusu içinde iken;If deyiminin ne anlam ifade ettiği ve nasıl çalışacağı sematik bir açıklamadır. Bu kursta genellikle bir dil ögesi ele alınırken önce sentax sonra semantik açıklama yapılacaktır.
Sentax bakımından doğru yazılmamış bir deyim semantik olarak bir analize sokulmamaktadır.
Programlama dillerinin sentax'ını açıklamak için çeşitli notasyonlar kullanılıyor olsada bunların en yaygını BNF(Bockus Nour Form) diye bilinen bir notasyondur. Genellikle programlama dillerinin standartlarında BNF ya da bunun bir türevi kullanılmaktadır.
Biz bu kursta şekilsel notasyonları kullanacağız. Köşeli parantez, açısal parantez, **Açısal parantezler kesinlikle kullanılması gereken ögeleri içerirken köşeli parantezler bulunması zorunlu olmayan ögeleri** içermektedir.

Derleme İşlemi: Bir derleyici yazılan bir programın yapmak istedikleri şeyleri anlayarak onu yapacak kod üretir. Derleyici bir dönüştürücü programdır. Bir programlama dilinde yazılmış bir programı başka bir programa diline dönüştüren programlara çevirici(translators) programlar denilmektedir.



Örneğin Pascal programlama dilinde yazılmış bir program C'ye dönüştüren programlar Çevirici programlardır.

Eğer hedef dil alçak seviyeli bir dil ise yani arakod, sembolik makine dili ya da doğal makine dili ise bu tür çevirici programlara derleyici (Compiler) denir.

Interpreter(Yorumlayıcı): Interpreter bir translators değildir. O anda programı yorumlayıp çalıştırır. Örneğin C# derleyicisi söz konusu olduğunda C# programlama dili kaynak dil, Hedef dil ise IL ara koddur.

Derleyicinin derleme işlemi yaparken verdiği uyarılar iki (2) grup altında incelenir.
1. Uyarılar(Warning): Programcı herhangi bir sentax hatası yapmamıştır. Dolayısı ile derleyici kodu yükleyerek işlemi başarı ile tamamlar. Fakat derleyici programcının mantıksal bazı hatalar yapmış olduğundan şüphelenmektedir. Bu amaçla programcıyı uyarmaktadır.
2. Gerçek Hatalar: Dilin gramer kurallarına uyulmamasından kaynaklanan ciddi hatalardır. Derleyici derleme işlemini başarıyla gerçekleştiremez. Hatanın kesinlikle düzeltilmesi gerekir.

Merhaba Dünya Programı:

```
namespace CSD
{
        class App
        {
                public static void Main()
                {
                        System.Console.WriteLine("Merhaba Dünya");
                }
        }
}
```

C# programlarının uzantısı geleneksel olarak .cs'dir.
Programın komut satırından derlenmesi şöyle yapılır.

csc <Dosya İsmi>Enter
Örnek: csc Sample.cs Enter
Bu işlemin sonunda Sample.exe dosyası oluşur.

ATOM(Token): Bir programlama dilinde kendi başına anlamlı olan en küçük birime atom denir. Merhaba Dünya sonucunu çıkaran programını atomlarına şöyle ayırabiliriz.
Namespace, CSD, {, class, App, {, public, static, void, main, (,), {, System, ., Console, WriteLine, (, “Merhaba Dünya”,), ;, },},} ifadelerin hepsi atomdur.

Atomların Sınıflandırılması: Atomlar 6 gruba ayrılır.
1. Anahtar Sözcükler(Keywords/Resolved Words): Dil için özel anlamı olan değişken olarak kullanılması yasaklanmış sözcüklerdir. Örnek programda namespace, class, public, static, void, main...
2. Değişkenler(Variables/Identifiers): Bunlar isimlerini istediğimiz gibi verebileceğimiz atomlardır.
3. Operatörler(Operators): Bir işleme yol açan bu işlemin sonucu bir değere yol açan atomlar operatörlerdir.
4. Sabitler(Constant/Literals): Doğrudan yazılan sayılara sabitler denir.
5. Sözceler(Strings):” ”içindeki yazılar string tipinde atomlardır.
6. Ayıraçlar(Separators): Aç Kapa küme parantezleri, noktalı virgül, yukarıdaki grupların dışında kalan ifadeleri ayırmak için kullanılan atomlardır.
Derleyiciler derleme işlemini yaparken ilk adımda yaptıkları kaynak kodu atomlarına ayırmaktadır.

Boşluk Karekterleri: Klavyeden boşluk oluşturmak için kullanılan karakterlere boşluk karekter denir. Space Bar, Tab, Enter.

C# Kuralları

1. Atomlar arasında istenildiği kadar boşluk karakter bırakılabilir.
2. Anahtar sözcükler ve değişkenler bitişik yazılamaz. Fakat atomlar istenildiği kadar bitişik yazılabilir.

Merhaba Dünya Programının Açıklaması:
Bir C# programı kabaca isim alanlarından oluşur. İsim alanları sınıflardan, sınıflarda fonksiyonlardan(Methods) oluşmaktadır. C# programında fonksiyonların karşılığı metotlardır.

Anahtar Notlar: C# ve Java terminolojisinde fonksiyon yerine metod terimi kullanılır. Ancak kursumuzda biz C ve C++ da kullanılan fonksiyon terimini kullanacağız.
İki küme parantezi arasındaki bölgeye blok diyoruz. {}
Bir isim alanı şöyle oluşturulur.

```
namespace<İsim>
{
        --------
}
```

Bir sınıf alanı şöyle oluşturulur.

```
class<isim>
{
        ---------
}
```

Fonksiyon bildiriminin ise genel biçimi şöyledir.

Public	*static*	*void*	*Main*
[erişim belirleyici]	[Static]	<Geri Dönüşüm türü>	<Fonksiyon ismi>

```
<[Parametre Bildirimi]>
{
        --------
}
```

Main fonksiyonunda erişim belirleyicisi public alınmıştır.
Erişim belirleyicisi public, protected, private, protected internal'lerinden biri olabilir. Erişim belirleyicisi yazılmamışsa private gibi işlem görür. Yani fonksiyon tanımlamada default erişim tipi private'dir.

Anahtar Notlar: Static fonksiyonlara C# terminolojisinde “Static Method” static olmayan fonksiyonlara “instance method” denir. (Örnek=Instance)
Geri dönüş değeri void alınan bir fonksiyonun geri dönüş değeri yoktur denir. Main fonksiyonununda parametre bildirimi yapılmamıştır.

Bir fonksiyonun tanımlanması(defination) fonksiyonun programcı tarafından yazılması anlamına gelir. Bir sınıf sıfır tane veya daha fazla fonksiyon içerebilir. İç içe fonksiyon tanımlanamaz. Fakat fonksiyon tanımlamada sıranın bir önemi yoktur.
Örneğin:

```
class Sample
{
        public static void foo()
        {
                //----- (vs,vs)
        }
        public static void Bar()
        {
                //----
        }
}
```

Sample sınıfı n tane fonksiyondan oluşuyor.

Anahtar Notlar: Foo ve Bar isimleri bilgisayar terminolojisinde öylesine isimler olarak kullanılmaktadır.

Anahtar Notlar: C# da /* */ arasındaki ifadeler dikkate alınmaz. Bunlar programın açıklanmasına yardım eden yorum satırlarıdır. Aynı zamanda // ' den sonra yazılan ifadelerde satır sonuna kadar derleyici tarafından yorum satırı olarak kabul edilir.

Bir C# programı Main fonksiyonundan çalışmaya başlar. Main fonksiyonu herhangi bir sınıfta bulunabilir. Main bitince programda biter. Bir programda Main fonksiyonu en fazla bir tane bulunur.

Fonksiyonlar deyimlerden oluşmaktadır. Örnek programımızda WriteLine adlı fonksiyon çağrılmıştır. WriteLine Fonksiyonu Console sınıfı içindedir. Console sınıfı da System isim alanı içindedir. System isim alanı Console sınıfı ve WriteLine fonksiyonu bizim tarafımızdan yazılmamıştır.

Microsoft tarafından yazılmış olan çeşitli isim alanlarının içinde bulunan yüzlerce, binlerce fonksiyon vardır.

Console sınıfı böyle bir sınıf. Console sınıfı klavyeden ve ekrandan giriş ve çıkışları için oluşturulmuştur.

Anahtar Notlar: Nesne yönelimli Programlama Tekniği (Object Oriented Programming) çok kaba bir tanımla sınıflar kullanılarak programlama tekniğidir. Bir takım anahtar kavramların toplamından oluşmaktadır. Bu kavramlar bütünüyle birbirinden ayrı değildir. İç içedir.

Static Bir Fonksiyonu Çağırmanın genel biçimi şöyledir:

[İsim alanı ismi][.][Sınıf ismi][.]<Fonksiyonismi>([Argüman ismi]);

Console sınıfının WriteLine fonksiyonu parametre olarak belirtilen yazıyı cursor'un bulunduğu yere yazar ve satır başı yapar. Console sınıfının write fonksiyonu ise kursoru yazının sonunda bırakır.

Çağrılacak fonksiyon eğer başka bir isim alanında bulunan sınıfın bir fonksiyonu ise, isim alanı ismi.sınıf ismi.ve fonksiyon ismi yazılmak zorundadır. Eğer fonksiyon aynı isim alanında bulunan başka bir sınıfın içinde ise bu durumda isim alanı isminin belirtilmesine gerek yoktur. Fakat belirtilmesi **arıza(error)** oluşturmaz.

Eğer fonksiyon aynı sınıfın bir fonksiyonu ise yalnızca fonksiyon ismi belirtilerek çağırma yapılabilinir. Fakat isim alanı ismi ve sınıf ismi belirtilmesi arızaya yol açmaz.

C#'ın Temel Veri Türleri(Type): Tür bir değişkenin ya da sabitin bellekte kaç byte yer kapladığını belirten ve onun içine yerleştirilecek değerin niteliğini belirleyen önemli bir bilgidir.

Tür Belirten Anahtar Sözcük	**Uzunluk(Byte)**	**Yapı Karşılığı**	**Sınır Değerler**
int	4	Sytem.Int32	-2.147.483.648 , +2.147.483.647
uint	4	System.Uint32	0, 4.294.967.295
short	2	System.Int16	-32.768, +32.767
ushort	2	System.Uint16	0, +65.535
long	8	System.Int64	-9.223.372.036.854.775.808,+9.223.372.036.854.775.807
ulong	8	System.Uint64	0, 18.446.744.073.709.551.616
byte	1	Sytem.Byte	0, 255
sbyte	1	System.Sbyte	-128, +127
char	2	Sytem.Char	0, 65.535
float	4	System.Single	$3.6x10^{-38}$ – $3.6x10^{38}$
double	8	System.Double	$1.8x10^{-388}$ – $1.8x10^{388}$ (?)
decimal	16	Sytem.Decimal	28 digit noktalı sayı tutuyor
bool	1	System.Bool	True ve false değerini tutuyor

int türü pozitif ve negatif sayıları tutan bir türdür.

uint türü ise işaretsizdir(UnSign=u) Uint türü int türü ile aynı boyutta yer kaplar fakat negatif sayıları içermediği için int türüne göre iki kat fazla pozitif sayı tutar.

short türü iki byte uzunluğundadır. Tıpkı int türü gibi işaretli tam sayı türüdür.

ushort türü short türünün işaretsiz biçimidir.

long 8 byte uzunluğunda işaretli bir tam sayı türüdür.

ulong türü long türünün işaretsiz biçimidir.

byte 1 bytlik işaretsiz bir tam sayı türüdür.

sbyte ise 1 bytlik işaretli bir tam sayı türüdür.

char C# da 2 byte uzunluğundadır. Unicode biçimindedir. Karakter tutar. İşaretsiz tam sayı türü şeklinde değerlendirilir. Char türü Unicode karakterleri tutmak için düşünülmüştür.

float 4 byte uzunluğunda bir gerçek sayı türüdür. Float türünün yuvarlama hatalarına karşı direnci düşüktür.
Yuvarlama Hatası(Rounding Error): Sayının tam olarak ifade edilmeyip ona yakın bir sayı ile

ifade edilmesi şeklinde bir hatadır.

double 8 byte uzunluğunda gerçek sayı türündedir ve yuvarlama hatalarına karşı daha dirençlidir.

Sayılar nokta içinde tutuluyorsa buna fixed point tutulum denir. Kayan floatting point türünde ise sayı olduğu gibi yazılır ve nokta başka bir yerde tutulur.

decimal türde ise tam ve noktalı kısmı ile birlikte en fazla 28 dijitlik sayı tutulabilir. Fakat decimal sayılarda yuvarlama hatası sözkonusu olmaz. Bu tür özellikle finansal hesaplamalarda tercih edilmektedir. Fakat decimal sayı türü doğal bir sayı türü değildir. Derleyici aslında decimal işlemleri arka planda Makine komutlarıyla gerçekleştirmektedir. Halbuki float ve double türler daha etkin türlerdir.

float, double ve decimal türler default olarak işaretli türlerdir. Bunların işaretsiz biçimleri yoktur.

bool türü yalnızca doğru ya da yanlış bilgisini tutmaktadır.

Bildirim Kavramı:
Modern programlama dillerinde bir değişken kullanılmadan derleyiciye tanıtılmak zorundadır. Bu tanıtma işlemine **bildirim(declaration)** denilmektedir. Ayrıca defination yani tanımlama kavramıyla da aynı anlamda kullanılmaktadır.
Bildirimin genel biçimi şöyledir: *<tür> <değişken listesi>;*

Değişken listesi tek bir değişkenden oluşabileceği gibi birden fazla değişkenden de oluşabilir. Eğer birden fazla değişken kullanılacaksa aralarına virgül konulur.

Örnek:
int a;
long x, y, z;
decimal k, l;

Bildirimler fonksiyon içinde yapılabilir. Eğer bildirim fonksiyon içinde yapılırsa buna **yerel değişken(Local variable)** denilir. Eğer bildirim sınıfın içinde yapılmışsa buna **sınıfın veri elemanları(data members)** denir. Değişkenler genel sınıf içinde tanımlanamazlar.

Bildirimler İki Yerde Yapılabilir:
1. Fonksiyonların içlerinde
2. Fonksiyonların dışında fakat sınıfın içinde

Yerel değişkenler fonksiyonun herhangi bir yerinde bildirilebilir. Fakat bildirildikten sonra kullanılabilirler.

Bir değişkene bildirim sırasında değer verilebilir. Bu işleme **ilkdeğer verme (initialisation)** denilmektedir.
Örnek:
int a, b=10, c=20, d; burada a, b ve c'ye ilk değerler verilmiş d'ye verilmemiştir.

C# da bir değişkene henüz bir değer atamadan o değişken içindeki değerin kullanılmak istendiğini bir ifade de kullanılamaz.

Değişkenlerin Faaliyet Alanları(Scope): Bildirilen bir değişken ancak belirli bir program

aralığında kullanılabilir. İşte değişkenin kullanılabildiği bu alana **faaliyet alanı(Scope)** denilmektedir.

Bir fonksiyonun bir ana bloğu olmak zorundadır. Fakat bu bloğun içinde istenildiği kadar çok iç ya da ayrık bloklar olabilir. Bir yerel değişken tanımlanma noktasından sonra tanımlandığı blokta ve o bloğun kapsadığı bloklarda kullanılabilir. C# da değişkenin tanımlandığı blokta veya kapsadığı bloklarda aynı isimde bir değişken tanımlanamaz. Bu işlem geçersizdir.

Örnek:

```
namespace CSD
{
        class App
        {
                public static void main()
                {
                        long a;
                        //----
                }
                int a; ----error
                //---
        }
                public static void foo()
                {
                        //...
                }
        }
}
```

Fakat ayrık bloklarda aynı isimli değişkenler kullanılabilir.

Sabitler: Yalnızca değişkenlerin değil sabitlerin de türleri vardır.

1. 1. Sabitlerin türleri onların yazılış biçimleri ile ilgilidir. Bir sayı nokta içermiyorsa ve sayının sonunda herhangi bir ek yoksa sayı sırasıyla int, uint, long, ulong, türlerinin hangisinin sınırlarına ilk kez giriyorsa sabit o türdendir.
   Sayı ulong sınırlarının da içine sığmıyorsa hata verir.
   Örnek: 0 ----> int 100----->int 3.000.000.000---->uint 5000000000---->long

2. Sayı nokta içermiyorsa ve sonunda küçük harf ya da büyük harf "L" varsa sayı long ve ulong türlerinin sırasıyla hangisinin içine giriyorsa o türdendir.
   Örnek: 0L--->long 100L---->long

3. Sayı nokta içermiyorsa ve sayının sonunda küçük harf ya da büyük harf "U" varsa sayı uint veya ulong türlerinin hangisinin sınırları içine ilk kez giriyorsa o türdendir. Sayı ulong sınırlarının dışında ise error verir.
   Örnek: 100u--->uint 5000000000U----->ulong

4. Sayı nokta içermiyorsa ve sayının sonunda küçük harf ya da büyük harf "UL" Ya da "LU" varsa sabit ulong türündedir. Sınır aşılırsa error verir.

   Anahtar Notlar: C# da en çok kullanılan tam sayı türü int en çok kullanılan gerçek sayı türü ise double dir. Programcı özel bir nedeni yoksa bu sayı türlerini tercih etmelidir.
5. Sayı nokta içeriyorsa ve sayının sonunda hiçbir son ek yoksa sabit double türdendir.

Örnek: 1.2 3.5 gibi

6. Sayı nokta içersin ya da içermesin sayının sonunda küçük veya büyük "d" varsa sayı double türdendir.
   Örnek: 3.2d 4.6D

7. Sayı nokta içersin veya içermesin sayının uçlarında küçk ya da büyük "F" varsa sayı float türdendir. **Örnek:** 3F 4.6f

8. Sayı nokta içersin ya da içermesin sayının sonunda "M" varsa sayı decimal türdendir.
   Örnek: 5M, 7.6M

9. Bir karakter tek tırnak içine alınırsa bu ifade ilgili karekterin unicode tablosundaki sıra numarasını belirten bir sayıyı anlatır.Bu tür sabitler char türdendir. **Örneğin:** 'a' , '?', '.' gibi

Unicode her karakter için 2 bytlik yer tutulur. ASCII de ise 1 bytlik yer tutulur.

C# programlama dili tamamen Unicode da çalışır. Bazı karakterler var ki ekran(display) karşılığı yok. Bu tür karakterlere **"nonprintable karekterler"** denilir. Bu türk karekterler için tek tırnak içine ters bölü konularak ifade edilirler. Bunlardan bazıları aşağıdaki gibidir.
'\?' bu karakterler eşit karakter sabitlerini belirtir.

Örnek:

'\b' Backspace karekteri	'\a' Alert	'\f' form feed	'\r' Enter
'\\' ters bölü	'\v' vertical tab	'\" tek tırnak	'\n' newline

Anahtar Notlar: \ karekteri çift tırnak içinde de kullanılabilir. Aşağıdaki gibi bir hatayla çok sık karşılaşılır.

System.Console.WriteLine("C:\temp"); Halbuki şöyle yazılmalıdır.
System.Console.WriteLine("C:\\temp");

Anahtar Notlar: C# da char karakteri kesinlikle bir sayı gibi işlem görür ve kullanılabilir.
Örnek: x='a'+2; geçerlidir. Buradan 98 sayısı elde edilir. Aslında bilgisayarın temelinde karekter diye bir şey yoktur herşey aslında bir sayıdır.

10. True ve False bool türden sabitlerdir.

11.C# da byte, sbyte, short ve ushort türden sabit kavramı yoktur.

Yerel Değişkenlerin Ömürleri: Bir değişkenin bellekte tutulduğu zaman aralığına **ömür(duration)** denir. Değişkenler sürekli bellekte kalmazlar. Yaratılırlar belli bir süre yaşarlar ve yok ediliriler.

Örnek:

```
namespace CSD
{
        class App
        {
                public static void Main()
```

```
        {
            int a;
            {
                int b;
                func();
                //---
            }
            int c;
            //---
        }
        public static void func();
        {
            int x;
            {
                //---
                int y;
                //---
            }
            int z;
            //---
        }
    }
}
```

Yerel değişkenler programın akışı sürecinde değişkenlerin tanımlandığı noktaya gelindiğinde yaratılırlar. Akış sürecinde tanımlandıkları bloklardan çıkıldığında otomatik olarak yok edilirler. Yerel değişkenlerin bu biçimde yaratıldıkları yere **"stack"** denilmektedir. Stacklar yaratılma ve yok edilme otomatik bir biçimde çok hızlı yapılmaktadır.
Ömür ile faaliyet alanı konuları birbirini gerektirmektedir. Yani bloğun dışında o bloktaki yerel değişkene erişememizin nedeni aslında o değişkenin kendi bloğunun dışında yaşamamasıdır.

Visula Studio IDE'sini kullanarak C# Programlarının Derlenmesi:
Visual Studio IDE si sadece C# için değil diğer programlama dilleri içinde ortak IDE durumundadır. Visual Studio içinde proje oluşturmak için çeşitli seçenekler vardır. Fakat kursumuzda boş proje oluşturarak işlem yapılacaktır. Proje oluşturulduktan sonra projenin içine C# kaynak dosyası yerleştirilir. Derleme ondan sonra yapılır.
Sırasıyla
File/New/Project seçilir.
Project Type Visual C# Windows yapılır.
Template Empty Project olarak seçilir.
Aslında projeler solution denilen kavramın içinde bulunmaktadır.
Bir solution'a birden fazla proje yerleştirilebilinir. Bir proje yaratıldığında aynı zamanda bir solution da yaratılmış olur.
Normal olarak IDE solution için bir klasör oluşturur. Proje içinde o klasörün içinde bir klasör oluşturulur. Fakat "Create directory for solution" seçenek kutusundaki "tik" kaldırıldığında proje için ayrı bir solution klasörü oluşturulmaz.
New project menüsüdeki location solution, klasörün hangi klasörün altında yaratılacağını belirtir. Şimdi solution içindeki projeye bir dosya eklemek gerekmektedir. Bir solution yaratıldığında IDE bir solution için çeşitli işlemleri görsel olarak yapmakta kullanılır. Solution Explorer adlı bir exe yaratır. Bu pencere kapatılırsa wiew menüsünde tekrar yaratılabilir ya da araç çubuklarından bu pencere çıkartılabilir. Projeye kaynak dosya eklemek için Proje/Add new item seçilebilir. Ya da Aynı menüde solution explorer de projenin üzerine sağ klik yapılarak bağlam menüsünden add/new

item seçilerek te çıkartılabilir. Çıkartılan bu menüde Code Files seçilir. Artık kod dosyanın içine yazılabilir.

Yazılan kodu derlemek için build/build solution seçilir. Bu seçenek solution içindeki tüm projeleri derlemektedir. Halbuki proje ismi x olmak üzere menüde built x seçeneği de vardır. Bu seçenek yalnızca ilgili projeyi derlemektedir.

Programı derlemek için debug/start without debugging (Ctrl + F5) seçilir.
Zaten daha önce derleme yapılmadıysa bu seçenek önce derlemeyi yapıp sonra programı çalıştırmaktadır.

Daha önceki bir solution veya projeyi açmak için File/Open/Project Solution seçilir.
Solution klasörüne gelinir ve .sln uzantılı x projesi seçilir.

Fonksiyonların Geri Dönüş Değerleri: Fonksiyon çağrıldıktan sonra onu çağıran fonksiyona ilettiği değere **geri dönüş değeri(return value)** denilir. Fonksiyonun geri dönüş değerinin türü fonksiyon isminin sonuna yazılır. Geri dönüş değerinin void olması fonksiyonun geri dönüş değerine sahip olmadığı anlamına gelir.

İfade Kavramı(Expression): Değişkenlerin operatörlerin ve sabitlerin her bir bileşimine ifade denir.
Örneğin: a, 10, a+10; a+b+10;
Tek başına bir operatör ifade değildir.

Return Deyimi: Return işleminin genel biçimi şöyledir.
return[ifade];
Return deyimi fonksiyonu sonlandıran ve geri dönüş değerini oluşturur. Fonksiyonun geri dönüş değeri void değilse fonksiyon her olası akışında return deyimi ile karşılaşması zorunluluğu vardır.

Geri dönüş değerinin kullanılması bir geçici değişken yoluyla yapılmaktadır. Return işlemi ile bir geçici değişken yaratılır. Geri dönüş değerinin değeri önce bu geçici (temp) değişkene aktarılır. Sonra oradan alınıp kullanılır. Bu durumda Return işlemi yaratılan bu geçici değişkene atama işlemidir. **Fonksiyonun geri dönüş değerinin türü aslında bu geçici değişkenin türünü belirtir.**
Örneğin: *x=func();* işleminin eş değeri temp=return ifadesi; x=temp;
Yaratılan bu geçici değişken yine derleyici tarafından otomatik olarak silinir.

Bazı Matematiksel Fonksiyonlar:
System isim alanı içinde Math sınıfının double bir değere geri dönen bazı faydalı fonksiyonları vardır.
Örneğin: Sqrt isimli fonksiyon parametre içinde verilen değerin karekökü ile geri döner.

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         double result;
         result = System.Math.Sqrt(10);
         System.Console.WriteLine(result);
      }
   }
```

```
}
```
Math sınıfının iki parametreli Pow isimli fonksiyonu kuvvet alır. Log fonksiyonu logaritma, Sin, Cos, Tan, Atan fonksiyonları trigonometrik dönüşümleri yapar.

Klavyeden Değer Okumak: T bir tür olmak üzere T.Parse(System.Console.ReadLine()); klavyeden bir değer okuyarak onu T türünden geri verir.
Örnekler:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         int val = int.Parse(System.Console.ReadLine());
      }
   }
}
```

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         int a = int.Parse(System.Console.ReadLine());
         int b = int.Parse(System.Console.ReadLine());
         int c;
         c = a + b;

         System.Console.WriteLine(c);
      }
   }
}
```

```
namespace CSD
{
        class App
        {
                public static void Main()
                {
                        System.Console.Write("Lutfen bir sayi giriniz: ");
                        double val = double.Parse(System.Console.ReadLine());
                        System.Console.WriteLine(System.Math.Sqrt(val));
                }
        }
}
```

Fonksiyonların Parametre Değişkenleri:
Fonksiyonlar parametre değişkenlerine sahip olabilir. Parametre değişkenleri parametre parantezlerinin içinde bildirilir.
Örnek: *public static int Add(int a, int b)*

Parametreli bir fonksiyon parametre değişkeni kadar argümanla çağrılır.
x=Add(ifade1, ifade2);

Anahtar Notlar: Fonksiyonun parametre değişkenine parametre(*Parameter*) çağrılırken yazılan ifadelere **argüman (*argument*)** denilmektedir.
Parametreli bir fonksiyon çağrıldığında önce parametre değişkenleri yaratılır. Sonra argümanlardan parametre değişkenlerine karşı bir atama yapılır. Sonra programın akışı fonksiyona geçirilir. Fonksiyon sonlandığında parametre değişkenleri otomatik olarak yok edilir. Görüleceği gibi parametreli bir fonksiyonun çağrılması otomatik bir atama işlemine yol açmaktadır.

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         int x, y, z;
         x = int.Parse(System.Console.ReadLine());
         y = int.Parse(System.Console.ReadLine());
         z = Add(x, y);
         System.Console.WriteLine(z);
      }
      public static int Add(int a, int b)
      {
         return a + b;
      }
   }
}
```

Fonksiyonun parametre değişkenleri yalnızca fonksiyonun içinde kullanabilir.

Operatörler: Bir işleme yol açan işlem sonucunda bir değer üretilmesini sağlayan atomlara **operatör** denir. Operatörlerin işleme soktukları atomlara **operant** denir.
Her operatörün diğerlerine göre bir öncelik durumu vardır.

()	Soldan Sağa	a=b/c-d;
* /	Soldan Sağa	i1)=b/c;
+ -	Soldan Sağa	i2)=i1-d;
Eşittir=	Soldan Sağa	i3)=a=i2;

Operatörlerin birbirlerine göre konumu yukarıdaki tabloda gösterilmiştir. Tablo satırlardan oluşur. Üst satırlardaki operatörler alt satırlardaki operatörlere göre daya yüksek önceliklidir. Aynı satırlardaki operatörler eşit önceliklidir. Eşit öncelikli operatörler ifade içindeki konuma göre soldan sağa ya da sağdan sola yapılır. (Associativity)

değişken	operatör	ifade	operatör	ifade
Func	()	func()	+	func()

ifade

İfadeler 3 biçimde sınıflandırılır:

1. İşlevlerine göre sınıflandırma:
Bu grup operatörler aritmetik operatörler(aritmatical Operator) denir. + - * /
Bir gruba karşılaştırma operatörü denir (Compression Operator, Relational Operator(ilişkisel))
< > <= >=
Bir diğer gruba mantıksal operatörler(Logical Operator) denilir. &&(and) //(or)
Bir diğer grup operatörler bit operatörlerdir. (Bitwise Operator) Bit seviyesinde and ve or kaydırma işlevi yapan operatörlerdir.
Diğer operatörler (Spacial purpose)
2. Operant sayılarına göre yapılan sınıflandırma:
Tek operantlı operatörler(Uniary Op.)
İki operantlı operatörler(binary Op.) a=b+c+d
Üç operantlı operatörler(Ternary Op) a?b:c(1 tane var)

3. Operatörün konumuna göre sınıflandırma:

Önek operatör (Prefix Op)	+ab
Ara ek operatör (Infix Op)	a+b
Son ek operatör (Postfix Op)	ab+

Bir operatörün teknik olarak tanımlanabilmesi için üç sınıflandırma biçiminde de nereye düştüğünün belirtilmesi gerekir.
Örneğin: + operatörü 2 operantlı araek aritmetik bir operatördür.
Ya da ! Operatörü Uniary Prefix Logical bir operatördür.

+, -, x, / operatörleri 2 operantlı araek aritmetik operatörlerdir. Klasik dört işlemi yaparlar.

% Operatörü iki operantlı araek aritmetik bir operatördür. Mod alma operatörüdür.

İşaret + ve İşaret – operatörleri

Bunlar tek operantlı önek operatörlerdir. – operatörü operandının negatif değerini üretir, + operatörü operandıyla aynı değeri üretir. Öncelik tablosunda aritmetik opertörlerden daha önceliklidirler ve sağdan sola gurupta bulunurlar.

++ (Increment) --Decrement Operatörler:
++ operatörüne arttırma (***increment***), -- operatörüne eksiltme (***decrement***) operatörler denir. Hem *prefix* hemde *postfix* kullanılabilir operatörlerdir. ++operanttaki değeri bir arttırır --ise bir eksiltir. Bu öperatörleri *prefix* ve *postfix* kullanımları birbirinden farklıdır. ++ ve – operatörleri öncelik tablosunun ikinci düzeyinde sağdan sola doğrudur.

(,)		Soldan Sağa
++, --, +, -	--->	Sağdan Sola
*, /, %	--->	Soldan Sağa
+, -	--->	Soldan Sağa
=	--->	Sağdan Sola

Artırım yada eksiltim tablodaki öncelikte yapılır. Fakat *prefix* kullanımda artırılmış yada eksiltilmiş değer geri kalan işleme sokulurkan *postfix* kullanımda artırılmamış yada eksiltilmemiş değer işleme sokulur.

Örnek:

```
namespace CSD
{
        class App
        {
      public static void Main()
      {
                int a = 10, b;

                b = a++ * 2;

                System.Console.WriteLine(a);
                System.Console.WriteLine(b);

                }
        }
}
```

Sonuç:

11
20

Şüphesiz bu operatörlerin operantlarının birer değişken olması gerekmektedir. Örneğin ++3 gibi bir ifade anlamsızdır. Yada --5 gibi bir ifade anlamsızdır. Şüphesiz bu operatörler tek başlarına kullanıldığında *prefix* yada *postfix* kullanıldığında bir fark oluşmaz. Yani örneğin ++a ile a++ arasında bir fark yoktur.

Karşılaştırma Operatörleri:

C# da altı tane karşılaştırma operatörü vardır. >, <, >=, <=, ==, !=. Bunların hepsi iki operantlı *infix* operatörlerdir. Karşılaştırma operatörlerinin hepsi aritmetik operatörlerdir.

Mantıksal Operatörler(! *(NOT)*, && *(AND)*, || *(OR)*)

Ünlem operatörü tek operantlı *prefix*, && ve || operatörleri iki operantlı *infix* operatörleridir. Bu operatörlerin operantları mutlaka bool türden olmak zorundadır. Ünlem operatörü ikinci düzeyde sağdan sola gurupta && ve || operatörleri karşılaştırma operatörlerinden düşük önceliklidir.

Anahtar Notlar: C# standartlarında true ve false değerlerinin hangi sayısal değerlerle temsil edileceği hakkında bir şey söylenmemiştir. Zaten bunu programcının bilmesinede gerek yoktur.

&& ve || operatörleri klasik öncelik kuralına uymamaktadır. Bu operatörlerin kısa devre (*short circuit*) özelliği vardır. Bu operatörlerin sağ tarafında ne kadar öncelikli operatör bulunursa bulunsun önce sol tarafları tamamen yapılır bitirilir, sağ tarafları duruma göre hiç yapılmayabilinir. && operatörünün sol tarafı *False* ise sağ tarafı hiç yapılmaz sonuç *False* olarak belirlenir. || operatörünün sol tarafı *True* ise sağ tarafı hiç yapılmaz sonu *True* olarak belirlenir.
C# ta & ve | operatörleri bit operatörleridir. Bunlar sayının karşılıklı bitlerini *and* ve *or* işlemine sokarlar. Fakat bu operatörlerinin operantları *bool* türdende olabilmektedir. Bu durumda bunlar mantıksal operatör gibi de davranmaktadır fakat bir farkla: bunların kısa devre özelliği yoktur. O halde programcı kısa devre özelliği olmayan *and* ve *or* işlemleri kullanabilir.

Atama Operatörü:

Atama operatörü iki operantlı *infix* bir operatördür. Atama operatörüde bir değer üretir. Atama operatörünün ürettiği değer işlemlere sokulabilir, atama operatörü sol taraftaki atanmış olan değişkenin değerini üretir.

Örnek:
a=b=1;
i1=b=1 ⟹ *1*
i2=a=i1 ⟹ *1*

Örnek:
b = a = 3 + 2;

Yukarıdaki iki işlem arasında farklılık vardır. Aşağıdaki işlemde geçerlidir.

int a = 10;
Func (a = 10);

Noktalı virgülün (;) işlevi:

Noktalı virgül bir ifadeyi sonlandırarak o ifadenin ayrı bir biçimde ele alınmasını sağlar. Bu görevdeki atomlara sonlandırıcı (*terminator*) denilmektedir. Eğer noktalı virgülü unutursak derleyici önceki ifade ile sonraki ifadeyi tek bir ifade olarak ele alır ve hata olur. Basic gibi bazı dillerde *Enter* karakteri bu amaçla kullanılmaktadır.

Etkisiz Kodlar:

Aşşağıdaki gibi bir ifadede bir gariplik varmıdır;
3 + 2;

Bu ifadenin hiçbir etkisi yoktur. Ve C# bu gibi ifadeleri kabul etmez ve error verir. Halbuki *C* ve *C++* da bu hata vermez.
++a;

Bir etkisi vardır C# hata vermez. Benzer şekilde
Func (); Hata vermez.

İşlemli Atama Operatörleri

C# da +=, -=, *=, /=, %= gibi bir gurup işlemli operatörlerdir.
a += b; ⟹ *a = a + b;*

Yukarıdaki iki komutda aynı işi yapmaktadır. İşlemli atama operatörleri, atama operatörleri ile sağdan sola aynı gurupta bulunur.
a = 5;
*a *= 3 + 2;*
Yukarıdaki işlemin sonucu 5 dir.

Deyimler (statements):

İki cismin aynı cinsten olması birbirlerini içermeyeceği anlamına gelmez. Örneğin bir büyük kutu ve iki küçük kutuya dışarıdan baktığımızda üç kutu görürüz. Bu iki küçük kutuyu büyük kutunun içine koyup dışarıdan baktığımızda bir kutu görürüz. Büyük kutunun içerisini açıp baktığımızda iki kutu görürüz.

Deyim (*Statements*) programlamadaki çalıştırma birimleridir. Bir C# programı kabaca isim alanlarından (*namespace*), isim alanları sınıflardan(*class*), sınıflar fonksiyonlardan (*Method*), fonksiyonlarda deyimlerden (*Statements*) oluşur. Bir fonksiyonun çağırıldığında çalıştırılması demek o fonksiyonu içindeki deyimlerin tek tek çalıştırılması demektir. Örneğin program *Main* fonksiyonunda çalışmaya başlar, *Main* in çalışması, M*ain* içerisindeki deyimlerin çalıştırılmasıdır. *Main* de bir fonksiyon çağırılmışsa o fonksiyonun içerisindeki deyimler çalıştırılır. Yani aslında hep deyimler çalışmaktadır.

Deyimler 5 Kısma Ayrılırlar:

1. Basit Deyimler(Simple Statement): Bir ifadenin sonuna noktalı virgül (;) getirilirse buna basit deyim denir

x=11; gibi

2. Bileşik Deyimler(Compound Statement): Bir blok içerisine sıfır tane ya da daha fazla deyim yerleştirilirse bloğun tamamına bileşik deyim denir.

Anahtar Notlar: Bir fonksiyon deyimlerin çalıştırılması ile çalıştırılır. Basit bir deyimin çalıştırılması demek ilgili ifadenin yapılması demektir. Bileşik deyimlerin çalıştırılması demek içindeki deyimlerin tek tek çalıştırılması demektir.

3. Bildirim Deyimleri (Declaration Statement): Bildirim yapmakta kullanılan deyimlerdir.
Örneğin: *int a,b; long x;*

4. Kontrol Deyimleri: Programın akışını kontrol etmeye yarayan deyimlerdir. if, while, for gibi

5. Boş Deyim(Null Statement): Başında bir ifade bulunmayan noktalı virgüllere boş deyim denir.
x=10;; gibi *x=10;* basit deyim *; ise boş deyim*

Kontrol Deyimleri:

if Deyimi: Genel biçimi şöyledir.

```
if(<bool türden bir ifade>)
        <deyim>
else
        <deyim>
```

Derleyici if anahtar sözcüğünden sonra parantezler içinde bool türden bir ifade bekler. if deyiminin doğruysa ya da yalnışsa bölümünde tek bir ifade bulunmak zorundadır. Birden fazla deyim kullanılacaksa bloğa alınıp öyle kullanılmalıdır.

Örnek:

```
if (ifade)
else
        ifade;
if(ifade)
{
        ifade1;
        ifade2;
}
else
        ifade3;
```

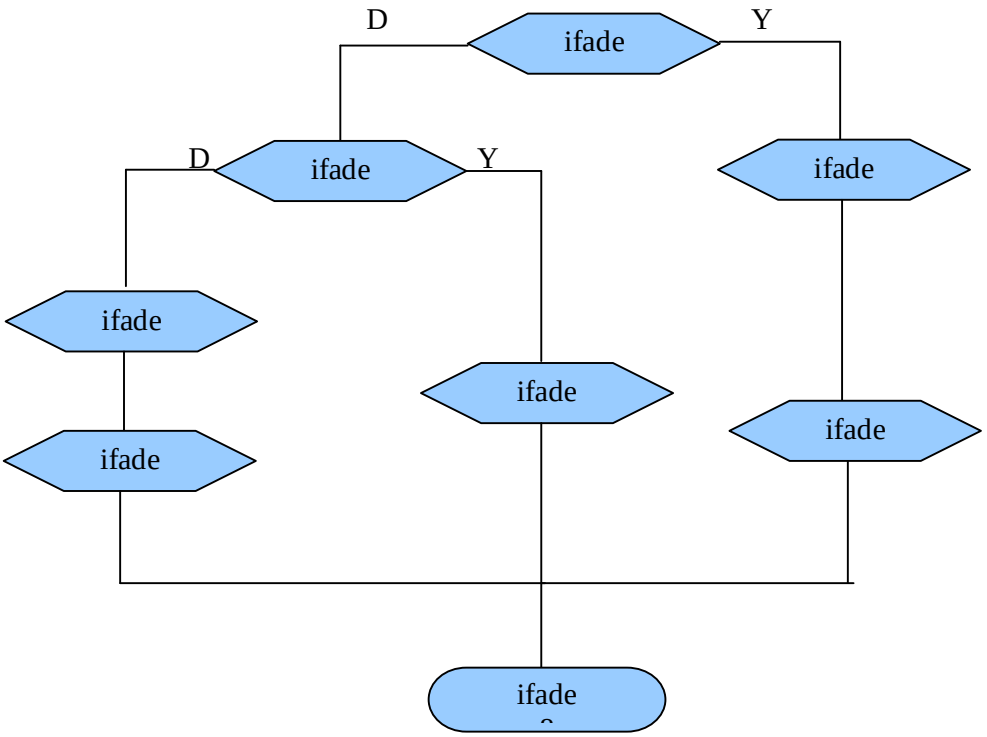
if deyimi şu şekilde çalışır. Derleyici if parantezi içindeki ifadenin değerini hesaplar. Bu değer true ise doğru bölümü çalıştırılır. Else kısmını atlar. Yanlışsa else kısmı çalıştırılır.

```
namespace CSD
{
        class App
        {
                public static void Main()
                {
                        int a;
                        System.Console.WriteLine("Bir sayi giriniz :");
                        a=int.Parse(System.Console.ReadLine());

                        if(a>100)
                                System.Console.WriteLine("Evet");
                        else
                                System.Console.WriteLine("Hayir");
                }
        }
}
```

if deyiminin else kısmı olmak zorunda değildir. Eğer derleyici if deyiminin doğruysa kısmından sonra else anahtar kısmını görmezse if deyiminin bittiğini düşünür.

Sıklıkla if deyiminin ifade kısmı ; ile kapatılmaktadır. Bu durumda if deyimi sonlandırılmış olup diğer deyimlere geçer ve bu ifadeler if deyiminin dışındadır.

if deyiminin tamamı tek bir deyimdir. Örneğin if deyiminin başka bir kısmında başka bir if deyimi bulunabilir.



```
If (ifade1)
        if(ifade2)
        {
                ifade3;
                ifade4;
        }
        else
                ifade5;
else
{
        ifade6;
        ifade7;
}
ifade8;
```

if deyimine karşılık bir else varsa derleyici else yi içteki if in else si olarak anlar ve işlem yapar. Eğer else nin ilk if in elsesi olması isteniyorsa ***bilinçli bloklama*** yapmak gerekir.

```
if(ifade1)
{
        if(ifade2)
                ifade3;
}
else
        ifade4;
```

şeklinde.

Bir koşul doğruyken diğerinin doğru olma olasılığı yoksa bu iki koşula ayrık koşullar denir.

Örneğin:

a==3	a>10
a==5ayrık koşular	a>20 ayrık koşullar

Ayrık koşulların ayrı if lerle sorgulanması kötü bir tekniktir. Bunlar için ***else if*** yapısının kullanılması gerekir.

Örneğin:

```
if(a>10)
        ifade1
else
        if(a<0)
                ifade2;
```

else if durumları bir merdiven oluşturabilir.

```
if(a==1)
        ifade1;
else if(a==2)
        ifade2;
else if(a==3)
        ifade3;
else
        ifade4;
```

Burda her if deyimi diğerinin else kısmındadır. if deyiminin doğruysa kısmı bulunmak zorundadır. Yalnızca yalnışsa kısmı bulunan bir if söz konusu olamaz. Fakat bu durum suni olarak sağlanabilir.

```
if(ifade1)
        ;
else
        ifade2;
```

Örnek: 2.derece denklemin köklerini bulan program:

```
namespace CSD
{
        class App
        {
                public static void Main()
                {
                        double a, b, c;
                        System.Console.WriteLine("Lütfen katsayıları giriniz: ");
                        a=double.Parse(System.Console.ReadLine());
                        b=double.Parse(System.Console.ReadLine());
                        c=double.Parse(System.Console.ReadLine());

                        GetRoots(a, b, c);
                }
                public static void GetRoots(double a, double b, double c)
                {
                        double delta;
                        delta=(b*b)-(4*a)*c;

                        if(delta<0)
                        {
                                System.Console.WriteLine("Gerçek kök yok");
                          }
                        else
                        {
                                double x1=(-b+System.Math.Sqrt(delta)/2*a);
                                double x2=(-b-System.Math.Sqrt(delta)/2*a);

                                System.Console.WriteLine(x1);
                                System.Console.WriteLine(x1);
                        }
                 }
        }
}
```

Döngüler: C# da 3 tür döngü vardır.

1. **while Döngüleri(While Loop)**
2. **for Döngüleri**
3. **for each döngüleri**

while döngüleri kendi içinde kontrolün başta ve sonda yapıldığı döngüler olmak üzere 2 kısma ayrılırlar.

Kontrolün başta yapıldığı while döngüsü: while döngüleri bir koşulun sağlandığı sürece yinelenen döngülerdir.

```
while(<bool türden ifade>)
        <deyim>
```

Döngü deyimi herhangi bir deyim olabilir. while döngüsü şöyle çalışır. Parantez içindeki ifadenin değeri hesaplanır. Bu değer true ise döngü deyimi çalıştırılır ve başa dönülür. False ise programın akışı döngünün dışındaki ilk deyimle devam eder.

Örneğin:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         int i = 0;
         while (i < 10)
         {
            System.Console.WriteLine(i);
            ++i;
         }
      }
   }
}
```

Döngünün yanlışlıkla boş deyimle kapatılması durumuyla sık karşılaşılmaktadır.

Örneğin:

```
while(ifade1);
{
        ifade2;
        ifade3;
}
```

Burada döngünün içinde boş deyim vardır yani bileşik deyim döngünün dışında kalmıştır.
while parantezinin içinde son ek bir arttırım veya eksiltim varsa döngü kararının verilmesinde arttırılmamış veya eksiltilmemiş değer kullanılır.

```
int i=0;
while(i++<10) //++i değil
        System.Console.WriteLine(i);
```

Burada 1 den 10 kadar sayılar ekranda gözükür.
n kez yinelenen bir döngü oluşturmakiçin pratik kalıp şöyledir.

```
int n=10;
while(n-->0)
{
        //----
}
```

n defa döngü kalıbı

Bir değeri önce atayıp sonra karşılaştırmak istiyorsak atama işlemine öncelik vermek için parantez kullanmalıyız.

Örneğin:

```
namespace CSD
{
        class App
        {
                public static void Main()
                {
```

```
                int a;
                while((a=int.Parse(System.Console.ReadLine()))!=0)
                {
                        //---(System.Console.WriteLine(a);
                }
        }
    }
}
```

Kontrolün Sonda Yapıldığı While Döngüleri:
Bu tür while döngülerinde döngü en az bir kez yinelenir. Genel yazım biçimi şöyledir.

do
    <deyim> //basit ya da bileşik deyim olabilir.
while (<Bool türden ifade>);

sondaki noktalı virgül mutlaka kullanılmak zorundadır. Burdaki noktalı virgül boş bir deyim değildir. Sentaks'ın bir parçasıdır.
Kontrolün sonda yapıldığı do while döngüleri ile seyrek karşılaşılmaktadır.

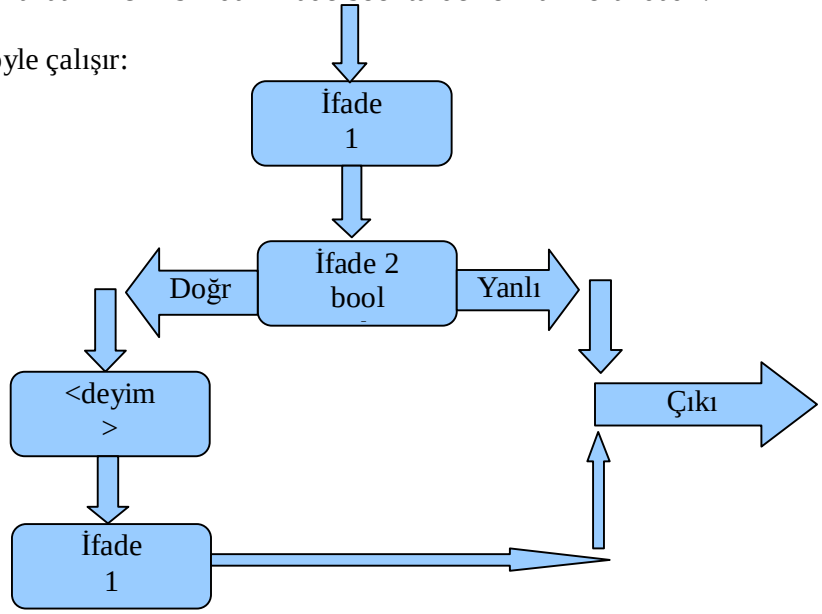
For Döngüleri:

For döngüleri while döngü deyimini de kapsamaktadır. For döngüsünün genel yazım biçimi şöyledir.

for([ifade1]; [bool türden bir ifade2]; [ifade3])
    <deyim>;

for anahtar sözcüğünden sonra parantezler içinde 2 tane noktalı virgül bulunmak zorundadır. Bu iki noktalı virgül for döngüsünü üç kısma ayırır. Birinci ve üçüncü kısımdaki ifadeler herhangi bir türden olabilir. Fakat ikinci kısımdaki ifade bool türden olmak zorundadır.

For döngüsü şöyle çalışır:



For döngüsü en çok şu kalıpla kullanılır.

```
for(ilk değer; koşul; arttırım)
{
        //...
}
```

Örnek:

```
for(i=0; i<10; ++i)
        System.Console.WriteLine(i);
```

for döngüsünün birinci kısmındaki ifade döngüye girişte bir defa yapılır. Bir daha yapılmaz. Döngü, ikinci kısımdaki ifade true olduğu sürece yinelenir. Üçüncü kısımdaki ifade her yinelemede döngü deyimi çalıştırıldıktan sonra gerçekleştirilir.

Mademki döngüye girişte birinci kısmdaki ifade bir defa çalıştırıldıktan sonra çalışmıyorsa bu ifadeyi döngünün yukarısına yazarsak sonuç farketmez.

Mademki üçüncü kısımdaki ifade döngü deyiminden sonra yapılmaktadır o halde aşağıdaki döngü de eşdeğerdir.

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         int i;

         i = 0;
         for (; i <= 10; )
         {
           System.Console.WriteLine(i);
           ++i;
         }
      }
   }
}
```

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         int i;

         i = 0;
         for (; ; )
         {
            if (i >= 10)
               break;
```

```
            System.Console.WriteLine(i);
            ++i;
        }
    }
  }
}
```

böylede yazılabilir.

For döngüsünün bölümlerine istediğimiz gibi ifade yerleştirebiliriz. Yeterki bizim için anlamlı olsun.

for döngüsünün ikinci kısmındaki ifade yazılmaz ise bu sonsuz döngü olması demektir.

for döngüsünün birinci ifadesinde bildirim yapılabilir. Bu durumda bildirilen bu değişken döngü deyimi içinde kullanılabilir. Dışarıda kullanılamaz.**(Yerel değişken)**
Kullanım alanı:

```
for(int i=0; i<10; ++i){
        //... Burada kullanılabilir
}
```

Ayrık birimlerde aynı değişken kullanılabilirken iç içe birimlerde aynı değişken ifade kullanılamaz. Standartlara göre:

```
for(T(tür) i= ; <ilk değer>; ifade2; ifade 3)
{
        //...
}
```

deyimin eşdeğeri

```
{
        T i;
        for(i= ; <ilk değer>; ifade2; ifade 3)
        {
                //...
        }
}
```

biçimindedir.

For döngüleri yanlışlıkla boş deyim kullanılarak kapatılmaktadır.

```
for(i=0; i<10; ++i);
        System.Console.WriteLine(i);
```

Yukarıdaki gibi bir durumla karşılaşılırsa döngü sonucu 10 değeri çıkar. Döngü çıkış değerini alır.

Break Deyimi: Break deyimi ya döngüler içinde ya da switch ifadesi içinde kullanılır. Yazım biçimi şöyledir

break;

Program akış çizgisi içinde break deyimini gördüğünde akış döngünün dışındaki ilk ifade ile devam eder.

Continue Deyimi: Continue sadece döngü içinde kullanılır. Kullanımı şöyledir. Programın akışı continue deyimini gördüğünde sanki döngü deyiminin çalıştırılması bitmiş gibi yeni bir yinelenmeye geçer.

Continue genellikle döngü içindeki uzun if bloklarından kurtulmak için kullanılır.

```
for(; ;)
{
        val = int.Parse(System.Console.Readline ());
        if (!Test (val))
                continue
                //...
}
```

Burada Test fonksiyonu true değer döndürünce döngüden çıkıyor. Continue olmasaydı bu döngü şöyle ifade edilirdi.

```
for(; ;)
{
        val = int.Parse(System.Console.Readline ());
        if (Test (val))
        {
                //...
        }
}
```

Go to deyimi: go to deyimi programın akışını koşulsuz bir biçimde belirli bir noktaya aktarmak için kullanılır. Genel kullanımı şöyledir.

```
goto <etiket>;
        //...
<etiket>:
```

Goto deyiminde fonksiyonlar arası yolculuk yoktur. Programın akışı goto anahtar sözcüğünü gördüğünde **<etiket>:** ile başlayan yere gider.
Etiket isimlendirme kurallarına uygun herhangi bir isim olabilir. Büyük harflerle yazılması tercih nedenidir. go to ile başka bir fonksiyona aktarma yapılamaz. Aynı fonksiyon içinde başka bir yere aktarılır. C# da blok içine goto yapmak yasaklanmıştır. Dışarıdan içeriye yasak fakat içerden dışarıya goto yapmak yasak değildir. Go to deyimi iç ve dış döngülerde tek hamlede döngüden çıkmak için kullanılır.

Switch Deyimi: Yalnızca sabitler ve operatörlerden oluşan ifadelere sabit ifadesi constant expression denir. Sabit ifadelerinin sayısal değeri derleme aşamasında belirlenir.
Switch deyimi bir ifadenin çeşitli sayısal değerleri için farklı işlemlerin yapılması amacıyla kullanılır. Genel yazım biçimi şöyledir.

```
Switch (<ifade>)
{
        case<sabit ifadesi>:
                //...
                break;
        case<sabit ifadesi>:
                //...
                break;
        default:
                //...
                break;
```

Derleyici switch anahtar sözcüğünden sonra parantezler içinde bir ifade bekler. Bu ifade tamsayı türlerine ilişkin olmak zorundadır. Switch deyimi “case” bölümlerinden oluşur. Case anahtar sözcüğünden sonra bir sabit ifadesi ve “:” iki nokta üst üste işareti gelmek zorundadır. Buradaki sabit ifadeleri tam sayı türü olmak zorundadır.

Switch deyiminin default bölümü olmak zorunda değildir. Case bölümlerinin sıralı olması ya da default bölümünün sonda olması zorunlu değildir.
Case bölümleri switch deyimlerinin içinde bulunmak zorundadır. C ve C++ da olduğu gibi başka bölümlerin içinde bulunamaz.
Switch deyimi şöyle çalışır: Derleyici switch deyimi parantezinin içindeki ifadenin sayısal değerini hesaplar. Sonra tüm case ifadelerini tam eşitlik bakımından kontrol eder. Eğer eşit bir case ifadesi bulursa bir go to işlemi gibi akışı oraya yönlendirir. Eğer tam eşit bir case ifadesi bulamazsa ve default ifadesi varsa akışı default a yönlendirir. Eğer uygun bir case ifadesi yoksa ve defaultta yoksa akış switch dışındaki ilk deyimle devam eder.

Akış case bölümüne aktarıldıktan sonra artık switch işlemi bitmiştir. Akış break anahtar sözcüğünü gördüğünde switch dışına çıkar.

C # da C/C++ da olduğu gibi **akışın aşağıya düşmesi yasaklanmıştır.** C# satandartlarına göre akış bir case bölümünden diğerine geçememektedir. Eğer geçiyorsa derleme zamanında hata(error) verir. Bu geçişin engellenmesi için tipik olarak break deyimi kullanılmaktadır. Fakat geçişin engellenmesi için break yerine return ya da go to gibi anahtar sözcükler de kullanabilir. Fakat istisna olarak bir case bölümüne hiçbir çıkış deyimi yerleştirilmemişse yani onu hemen başka bir case bölümü izliyorsa aşağıya düşmeme kuralı geçerli değildir. Birikimli işlem yapmak için “ goto case”ve “goto default” gibi iki deyim eklenmiştir. Bu deyimler switch içinde kullanılabilirler.

```
Switch (<ifade>)
{
        case1<sabit ifadesi/si>:
                //...
                goto case2
        case2<si>:
                //...
                break;
        default:
                //...
                break;
```

Write ve WriteLine Fonksiyonlarının Parametrik Kullanımı:
Console sınıfının Write ve WriteLine fonksiyonları özel bir biçimde kullanılabilir. Fonksiyonlar iki tırnak içindeki karakterleri tek tek yazdırır. Fakat küme parantezi {n} n gibi bir ifadeyi yazdıramaz. Küme parantezi n {n} bir yer tutucudur. İki tırnak **“”** tan sonraki argümanların 0'dan itibaren birer indeks numarası vardır. {n} n. indeks numarasındaki argümanın değeri

```
int a=10,b=24;
        //...
        System.Console.WriteLine(“a={0} b={1}”, a, b)
```

çıkış değeri a=10 b=24 tür.

Not: Sayı duyarlılığı ({0:2d} {0:2f})

Farklı Türlerin Birbirine Atanması:
C# da atama anlamına gelen 3 durumla karşılaştık.

1. Eşittir “=” operatörü ile yapılan açık atama
2. Return işlemi sırasında geçici değişkene yapılan atamalar.

3. Fonksiyon çağrılırken parametre aktarımı sırasında yapılan atamalar.

C# kurallarına göre atama işlemleri sırasında sağ tarafın değeri sol tarafın değerine otomatik olarak (implicit) dönüştürülmektedir. Eğer sağ taraf türünden sol taraf türüne otomatik dönüştürme mevcut ise atama geçerlidir. Değilse error verir.

Farklı türlerin atamasıyla farklı türler arasındaki otomatik dönüştürme işlemi aynıdır.
C# da özet olarak küçük türden büyük türe doğrudan dönüştürme yani atama geçerlidir. Fakat tersi error verir. Başka bir deyişle bilgi kaybına yol açacak atamalara izin verilmemektedir. Büyük türün içindeki değer küçük türün içindeki değere sığıyor olsa bile atama geçerli değildir. Hatta aşağıdaki tür atamalar bile yasaktır.
long x=10
int y;
y=x

int a;
a=10L error ile sonuçlanır.

Yukarıdaki özet kuralın bazı ayrıntıları vardır.

1. Gerçek sayı türlerinden tamsayı türlerine atama yapmak yasaktır. Örneğin float doğrudan long değerine atanamaz.
2. Fakat bir tamsayı türünden gerçek sayı türüne doğrudan dönüşüm mevcuttur.

Anahtar Notlar: int ya da long türünden float türüne atama yapabiliriz. Bu durumda bilgi kaybı sözkonusu olabilir. Neyseki bu bir basamak kaybı değil mantis kaybıdır. Bu durum ciddi bir kayıp olarak değerlendirilmemiştir.

3. Küçük işaretli türden büyük işaretsiz türe doğrudan dönüştürme yoktur. Çünkü işaretsiz tür yalnızca pozitif sayıları tutar.
   Örnek: short türünü uint türüne atayamayız.

Fakat bunun tersi küçük işaretsiz türün büyük işaretli türe atanması bilgi kaybına yol açmaz.
Aynı türün işaretli ve işaretsiz türleri arasında doğrudan dönüştürme yoktur. Yani int türünü uint türüne atayamayız. Tersi de doğrudur.

4. Float ve Double türünden decimal türüne bir atama yoktur. Tersi de yoktur.
5. Bool türünden hiçbir türe dönüşüm yoktur.
6. Int türden bir sabit ifadesi belirttiği değer hedef türün sınırları içinde kalıyorsa sbyte, byte, short, ushort, uint, ulong türlerine doğrudan dönüştürülebilir. Bu sayede int türünden küçük türlere atama yapma işlemi sağlanmıştır.

Örneğin:

byte b;	*byte b;*	*long a;*
a=100;	*b=-100 geçersiz.*	*a=100; geçerli*

int x=100;
byte b
b=x; geçersiz ifade sabit değil

7. Long türden bir sabit ifadesi belirtilen sayı hedef türün sınırları içinde kalıyorsa ulong türüne doğrudan dönüştürülebilir.
   Örneğin: ulong a;

a=100.000.000.000 geçerli ifade

Burada 100 milyar long türünden bir sabit ifadesidir ve atama geçerlidir.

Yukarıda anlatılanların sonucu olarak hangi türün hangi türe dönüştüreleceği aşağıdaki tabloda anlatılmıştır.

sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
char	ushort, uint, int, long, ulong, float, double, decimal
float	double

İşlem Öncesi Otomatik Tür Dönüştürme

C# derleyicisi bir operatörle karşılaştığında önce operandların türlerini araştırır. Eğer operandlar aynı türdense işlem hemen yapılır. Sonuç da aynı türden çıkar. Eğer operandlar farklı türdense önce aynı türe dönüştürülür sonra işlem yapılır. İşlem sonucunda elde edilen değer bu ortak türden olur. Tür dönüştürmesi aslında küçük türün büyük türe dönüştürülmesi şeklinde gerçekleşir. Örnek olarak int ile long türleri işleme girdiğinde int türü long türüne dönüştürülür. Sonuç long türden çıkar. Küçük türün büyük türe dönüştürülmesi sırasında büyük tür türünde geçici bir değişken yaratılır. Sonra küçük tür o geçici değişkene atanır. İşleme o geçici değişken sokulur. Son olarak bu değişken yok edilir.

```
namespace CSD
{
   class App
   {
      public static void Main()
                {
                        double result;
                         result = 10 / 3 * 2;        /*10.0/3*2*/
                        System.Console.WriteLine(result);
                }
   }
}
```

C# da iki tamsayı türünün birbirine böldüğümüzde sonucun tamsayı çıkması için bir gerekçe yoktur.

Küçük türün büyük türe dönüştürülmesi kuralının bazı ayrıntıları vardır.

1. int türünden küçük olan iki tür kendi aralarında işleme sokulduğunda önce her iki operand int türüne dönüştürülür sonucu int türünden elde edilir.

```
byte a=10, b=20, c;
 c=a+b; ---->error!
```

2. Küçük işaretli tür ile büyük işaretsiz tür işleme sokulamaz. Fakat istisna olarak long ile ulong türü işleme sokulabilmektedir. Yani operandlardan biri ulong türünden ise ve diğer operand tamsayı türünden ise bu durumda işleme sokulabilir.

short a=10;
ushort b=20;
long c;
c=a+b; ------->geçerli

Fakat işlem öncesinde int türünden küçük olan türler otomatik int türüne dönüştürüldüğü için short türü ile ushort türünü işleme sokabiliriz.

3. Bool türü hiçbir türe dönüştürülemez.

4. Float ve double türleri decimal türle işleme sokulamaz.

5. Tam sayı türleri ile gerçek sayı türleri işleme sokulduğunda dönüştürme her zaman gerçek sayı türüne dönüştürülür.

Örneğin: *long ile float türleri işleme sokulduğunda dönüştürme long türünden float türüne doğru olur.*

Tür Dönüştüme Operatörü:
Aslında bool türü dışında biz bilinçli olarak tür dönüştürme operatörü ile bilgi kaybını göze alarak her temel türü diğerine dönüştürebiliriz.
Tür dönüştürme operatörünün genel biçimi şöyledir.
(tür) operand
Tür dönüştürme operatörü öncelik tablosunda 2. düzeyde sağdan sola öncelik önemine sahiptir.
Örnek: *c = (long) a*b;*
int a=10; b=3;
double c;
c = (double) a/b;

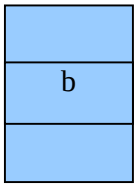
Tür dönüştürme işlemi sırasında dönüştürülecek tür türünden geçici bir değişken yaratılır. Dönüştürülecek değer o değişkene atanır. İşleme bu geçici değişken sokulur. İşlem bittiğinde bu geçici değişken yok edilir.

Tür dönüştürme operatörü ile dönüştürme sırasında bilgi kayıpları şu biçimde oluşur.

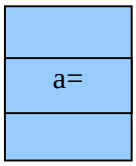
1. Eğer dönüştürülecek değer hedef türün sınırları içinde kalıyorsa bilgi kaybı sözkonusu olmaz.
2. Büyük tamsayı türünden küçük tamsayı türüne dönüşümde sayı küçük türün sınırları içine sığmıyorsa sayının yüksek anlamlı bitleri atılıyor.
3. Aynı türün işaretli ve işaretsiz türleri arasında dönüştürme yapılırken sayının bit kalıbı değişmez. Sadece işaret bitinin anlamı değişir.
4. Gerçek sayı türlerinden tamsayı türlerine dönüştürme yapıldığında sayının noktadan sonraki kısmı tamamen atılır. Eğer bunun sonucunda bile sayı hedef türün sınırları içine sığmıyorsa ***Exception*** oluşur.
5. Küçük işaretli türden büyük işaretsiz türe dönüştürme yapılırken dönüştürme 2 aşamada gerçekleştirilir. Önce büyük türün işaretli biçimine devamında büyük türün işaretsiz biçimine dönüştürülür.

ADRES Kavramı: Bellekteki her bir byte ilk byte "0" olmak üzere artan sayı karşılığında bir sayıya karşılık getirilmiştir. Her değişkenin yaşam süresi içinde bir adresi vardır.

Örneğin:



10010138 bir byte dan uzun olan değişkenlerin adresleri için onların en küçük adres değeri kullanılır.



10000180 ---->a nın adresi
111111111
000111111

Değer Türleri ve Referans Türleri: C# da kategori olarak türler **değer türleri (value types)** ve **referans türleri (reference types)** olmak üzere 2'ye ayrılır. Eğer bir değişkenin türü kategori olarak değer türlerine ilişkinse, değişken doğrudan değerin kendisini tutar. Eğer değişkenin türü kategori olarak referans türlerine ilişkinse bu durumda değişken bir adres tutar. Asıl değer, tuttuğu adresteki bellek bölümündedir. Başka bir deyişle değişken değerin bulunduğu bellek bölgesinin adresini tutmaktadır. Bugüne kadar gördüğümüz C#'ın temel türlerinin hepsi kategori olarak değer türlerine ilişkindir.

Sınıfların Veri Elemanları: Bidirimleri fonksiyonların dışında fakat sınıfların içinde yapılan değişkenlere sınıfın veri elemanları denir. Sınıfın veri elemanları tıpkı fonksiyonlar gibi statik olabilirler ya da olmayabilirler. Onlarda erişim belirleyicilerine sahiptirler. İlgili konu açıklanana kadar public değeri alacaklardır.

Bir sınıf türünden değişkenler tanımlanabilir.
Örneğin:

```
namespace CSD
{
        class sample
        {
                public int a;
                public int b;
                public static int c;
                public void foo()
                {
                        //-----
                }
                public static void bar()
                {
                        //---
                }
        }
        //---
        sample x;
}
```

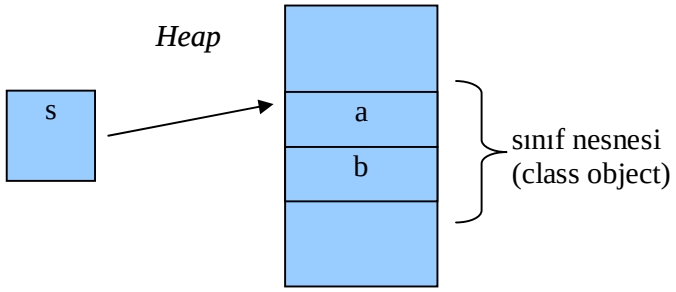
C# da sınıflar kategori olarak referans türlerine ilişkindir. Yani sınıf türünden değişkenlerin içine adres yerleştirilmelidir. Sınıf türünden değişken tanımladığımızda biz yalnızca adres tutan bir

değişken yaratmış oluruz. Ayrıca bu değişkenin gösterdiği yerin de tahsis edilmesi gerekir. Bu işlem "new" operatörü ile yapılır.
"new" operatörünün kullanım şekli şöyledir.

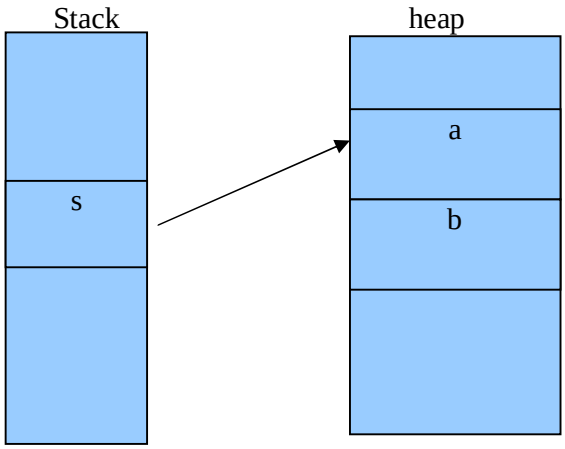
new <sınıf ismi> <[argüman listesi]>

Örneğin:
sample s;
s=new sample ();

new operatörü belleğin heap (yığın) denilen bölümünde sınıfın statik olmayan veri elemanları için ardışıl bir blok tahsis eder. Tahsis ettiği bloğun adresi ile geri döner.
Örneğin:
sample s;
s = new sample();



Görüldüğü gibi new ile tahsisat yapıldığında sınıfın statik veri elemanları için yer ayrılmaz. Fonksiyonlar içinde yer ayrılmaz. Yalnızca sınıfın statik olmayan veri elemanları için yer ayrılır. New ile yer ayırdığımız alana sınıf nesnesi denilmektedir. Sınıf nesnesi bir grup değişkenden oluşan bileşik bir nesnedir. Statik olmayan veri elemanları bu nesnelerin parçalarını oluşturmaktadır.
Bir değişken ister değer türlerine ilişkin olsun isterse referans türlerine ilişkin olsun eğer bir fonksiyonun içinde tanımlanmışsa yerel bir değişkendir ve yerel değişkenler bellekte **"stack"** denilen yerde yaratılırlar. Oysaki new ile yaratılan tahsisatta "heap" denilen bölgede gerçekleştirilmektedir. Yani stacktaki referans heap deki nesnenin adını tutmaktadır.

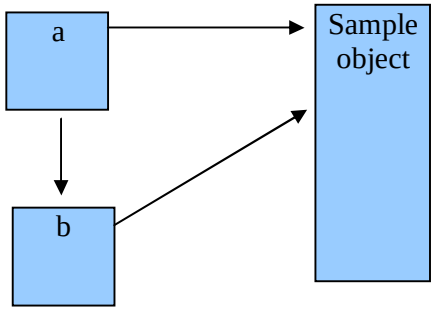


Her new işlemi heap de yeni bir sınıf nesnesinin yaratılmasına yol açmaktadır.

Aynı türden iki referans birbirlerine atanabilir. Böylece iki referans aynı nesneyi gösteriyor konuma gelirler.

Örneğin:

sample a=new sample();
sample b;
b=a;



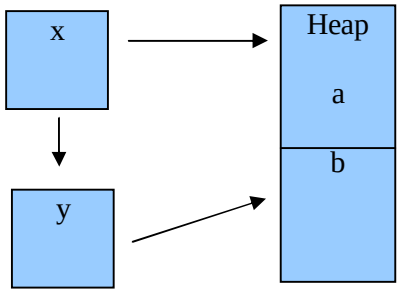

```
class sample
{
        public int a;
        public int b;
        //...
}
//---
sample x=new sample ();
```

x.a burda nokta nesne operatörüdür.

"." nokta operatörünün sonunda bir sınıf türünden referans bulunabilir. Bu durumda sağında ilgili sınıfın bir elemanı bulunmak zorundadır. Operatör sol taraftaki referansın gösterdiği yerdeki nesnenin sağ tarafta belirtilen nesnenin elemanına erişmekte kullanılır.
Nokta operatörü öncelik tablosunun en yüksek düzeyindedir. Dolayısıyla ++x.a gibi bir işlemde önce x.a yapılır. Sonra x.a arttırılır.

```
namespace CSD
{
        class App
        {
                public static void Main()
                {
                        sample r = new sample();
                        r.a = 10;
                        r.b = 20;

                        System.Console.WriteLine(r.a);
                        System.Console.WriteLine(r.b);
                }
        }
        class sample
        {
                public int a;
                public int b;
                public static int c;
                //...
        }
}
```

```
class sample
        {
                public int a;
                public int b;
                //...
        }
//...
sample x = new sample();
x.a = 10;
x.b = 24;
sample y;
y = x;

System.Console.WriteLine(y.a);
System.Console.WriteLine(y.b);
```



Yukarıda da söz edildiği gibi aynı sınıf türünden iki referansı birbirine atatığımızda referansların içindeki adresler atanır. Bu durumda iki referans aynı nesneyi gösterir. Nesnelerin elemanlarına artık hangi referansın erişitiğinin bir önemi yoktur.

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         sample x = new sample();
         sample y = new sample();

         x.a = 10;
         x.b = 20;

         y.a = 30;
         y.b = 40;

         System.Console.WriteLine("x.a = {0} x.b = {1}", x.a, x.b);
         System.Console.WriteLine("y.a = {0} y.b = {1}", y.a, y.b);
      }
   }
   class sample
   {
      public int a;
      public int b;
      public static int c;
      //...
   }
}
```

Sınıfın Statik Veri Elemanları:
Sınıfın statik veri elemanlarının toplamda tek bir kopyası vardır. Bu kopya program çalıştırıldığında hazır bir biçimde bulundurulmaktadır. Yani statik veri elemanlarını kullanmak için bir new işlemi yapmaya gerek yoktur. Statik veri elemanlarına sınıf ismiyle erişilir. Referansla erişilmez. r bir

referans olmak üzere r.a ifadesi referansın gösterdiği yerdeki nesnenin a parçası anlamına gelmektedir. Oysaki statik elemanlar referansın gösterdiği yerde değildir. Farklı sınıfların aynı isimde statik veri elemanları olabilir. Bu nedenle erişimde sınıf ismi gerekmektedir.

```
namespace CSD
{
	class App
	{
		public static void Main()
		{
			Sample.c=10;
			Test.c=20;

			System.Console.WriteLine(Test.c);
			System.Console.WriteLine(Sample.c);
		}
	}

	class Sample
	{
		public int a;
		public int b;
		public static int c;
		//...
	}
	class Test
	{
		public int a;
		public int b;
		public static int c;
		//...
	}
}
```

Sınıfın Statik Olmayan Fonksiyonları:
Sınıfın statik olmayan fonksiyonları sınıfın statik olmayan veri elemanlarını doğrudan kullanabilir. Fakat sınıfın statik fonksiyonları sınıfın statik olmayan veri elemanlarını kullanamaz.
Sınıfın statik fonksiyonları sınıfın ismiyle çağrılırlar. Statik olamayan fonksiyonları ise nesneyle çağrılırılar.
s bir sınıf türünden referans func ise bu sınıfın statik olmayan fonksiyonu olsun çağırma işlemi s.func(---) biçiminde olur.
Statik olmayan fonksiyonlar içinde statik olmayan veri elemanları o fonksiyon hangi referansla çağrılmışsa o referansın gösterdiği yerdeki nesnenin veri elemanlarıdır.

```
namespace CSD
{
	class App
	{
		public static void Main()
		{
			Sample s= new Sample ();
			s.Set(10,20);
```

```
                Sample k= new Sample ();
                k.Set(30,40);
                s.Disp();
                k.Disp();
            }
        }

        class sample
        {
            public int a;
            public int b;
        }
        public void Set(int x, int y)
        {
            a=x;
            b=y;
        }
        public void Disp()
        {
            System.Console.WriteLine(x);
            System.Console.WriteLine(y);
        }
}
```

Sınıfın statik fonksiyonları statik veri elemanlarını doğrudan kullanabilir. Statik veri elemanlarının bir referansa gereksinimi yoktur. Sınıfın statik olmayan fonksiyonları sınıfın statik veri elemanlarını doğrudan kullanabilir. Program bu elemanları sınıf ismiyle kullanıyormuş varsayar. Özetle sınıfın statik olmayan fonksiyonları hem statik olmayan veri elemanlarını hem de statik veri elemanlarını doğrudan kullanabilir. Fakat sınıfın statik fonksiyonları yalnızca sınıfın statik veri elemanlarını kullanabilirler.

Sınıfın statik bir fonksiyonu sınıfın başka bir statik fonksiyonunu doğrudan çağırabilir. Bu durumda bu statik fonksiyonun sınıf ismiyle çağrılmış olduğunu düşünebiliriz.
Fakat sınıfın statik bir fonksiyonu statik olmayan bir fonksiyonunu doğrudan çağıramaz. Sınıfın statik olmayan bir fonksiyonu sınıfın statik bir fonksiyonunu çağırabilir.

Eleman kullanımına ilişkin yukarıda açıklanan kurallar şöyledir.
1. Sınıfın statik olmayan bir fonksiyonu sınıfın statik olmayan veri elemanlarını doğrudan kullanabilir. Sınıfın statik veri elemanlarını da doğrudan kullanabilir. Sınıfın statik olmayan fonksiyonlarını çağırabilir. Sınıfın statik fonksiyonlarını da çağırabilir.

2. Sınıfın statik fonksiyonları sınıfın statik olmayan veri elemanlarını doğrudan kullanamaz. Fakat sınıfın statik veri elemanlarını doğrudan kullanabilir. Sınıfın statik olmayan fonksiyonlarını doğrudan çağıramaz.Fakat sınıfın statik fonksiyonlarını doğrudan çağırır.

Anahtar Notlar: Sınıfın elemanı demekle hem veri elemanlarını hem de fonksiyonlarını kastederiz.

Yukarıdaki açıkladığımız erişim kurallarını daha da özetlersek şunları söyleyebiliriz: Sınıfın statik olmayan fonksiyonları sınıfın hem statik hem de statik olmayan elemanlarını doğrudan kullanabilir. Sınıfın statik fonksiyonları ise yalnızca sınıfın statik elemanlarını doğrudan kullanabilir.

Bir sınıf için foo ()gibi bir fonksiyon yazacak olalım bu fonksiyonu statik mi yapmalıyız. İşte foo fonksiyonu sınıfın statik olmayan veri elemanlarını kullanıyorsa biz bu fonksiyonu statik yapamayız. Fakat kullanmıyorsa onu statik yapabiliriz. Şüphesiz sınıfın statik olmayan veri elemanlarını kullanmadığı halde yinede biz foo foksiyonunu statik olmayan bir fonksiyon biçiminde yazabiliriz. Fakat bu durumda onu referansa mahkum etmiş oluruz.
Örnek: *r.Foo();*

Foo sınıfın statik olmayan veri elemanlarını kullanmadığı halde sanki yukarıdaki ifade de r referansının gösterdiği yerdeki elemanları kullanıyormuş gibi bir izlenim oluşturmaktadır.

Rastgele Sayı Üretilmesi: Rastgele sayı üretimi için system isim alanındaki Random sınıfı kullanılır. Random sınıfının int parametreli statik olmayan next isimli fonksiyonu 0 dan parametresiyle belirtilen sayı – 1 'e kadar sayı arasında rastgele bir sayı üretir.

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         System.Random r = new System.Random();

         for (int i = 0; i < 10; ++i)
                  System.Console.WriteLine(r.Next(100));
         }
   }
}

namespace CSD
{
   class App
   {
      public static void Main()
      {
         System.Random r = new System.Random();

         long tail = 0, head = 0;

         for (int i = 0; i < 1000000; ++i)
         {
            if (r.Next(2) == 0)
               ++head;
            else
               ++tail;
         }
         double headRatio = head / 1000000D;

          System.Console.WriteLine(headRatio);
         }
   }
}
```

Çöp Toplayıcısı: Bir yerel değişken ister temel türlere ilişkin olsun isterse referans türlerine ilşkin olsun stack denilen bölümde yer alır. Anımsanacağı gibi yerel değişkenler programın akışı tanımlanma bölgesine geldiğinde yaratılıp akış onların tanımlandığı bloktan çıktığında değişkenler yok edilmektedir. Bu yaratılma ve yok edilme çok hızlı olmaktadır. Heap teki sınıf nesneleri new operatörü tarfından akış new ifadesine geldiğinde yaratılır. New operatörü ile tahsisat göreli biçimde yavaş bir işlemdir.
Stack teki bir referans heap teki bir nesneyi gösteriyor olsun. Akış referansın tanımlandığı bloktan çıktığında referans otomatik yok edilir. Peki heap tek new operatörü ile tahsis ettiğimiz alan ne olacak. C ve C++ da nasıl new gibi bir operatör varsa delete gibi de bir operatör vardır. C ve C++ da new ile tahsis edilmiş alanların silinmesi programcının sorumluluğundadır. Halbuki Java ve C# da new ile yaratılmışi nesneler arkaplanda **çöp toplayıcı(Garbage Collector)** denilen bir mekanizma sayesinde silinir.
Çöp toplayıcı mekanizma heapteki nesne hiçbir referans tarafından gösterilmiyor duruma geldiğinde silmeye çalışır. Heap teki bir nesneyi belirli bir anda belirli sayıdaki bir referans gösteriyor durumdadır. Buna nesnenin referans sayı denilmektedir. Programın akışı sırasında nesneyi gösteren hiçbir referans kalmamışsa nesnenin referans sayacı sıfıra düşmüş denilmektedir. Bu duruma nesnenin çöp toplayıcı için **seçilebilir (eligible)** duruma gelmesi denilmektedir. Bir nesne seçilebilir duruma geldikten ne kadar sonra çöp toplayıcı onu siler? Işte gerek C# standartlarında gerekce CLI ile standartlarında bu konuda belirleyici bir şey söylenmemiştir. Çöp toplayıcı mekanizma programcının çıkarına olacak biçimde uygun gördüğü bir zamanda devreye girerek seçilebilir durumdaki nesneleri heap ten siler. Şüphesiz iyi bir biçimde yazılmış CLR sistemi silme işlemini çokta geç kalmadan yapacaktır. Bazan çöp toplayıcı nesneyi silen program sonlanmış olabilir. Özetle çöp toplayıcının seçilebilir duruma gelmiş olan nesneleri silmesi tam olarak planmış bir deterministik biçimde yapılmamaktadır.

System isim alanı içindeki GC isimli sınıf çöp toplayıcı faaliyetleri ile ilgilidir. GC sınıfının statik collect fonksiyonu çöp toplayıcıyı göreve davet ederek çöp toplama işlemini o anda gerçekleştirir.

Referansların Fonksiyonlara Parametre Olarak Geçirilmesi: Fonksiyonun parametre değişkeni bir sınıf türünden referans olabilir. Fonksiyon da bir sınıf referansı ile çağrılabilir. Bu durumda fonksiyon içinden ilgili nesne kullanılabilir.

```
using System;

   class circle
   {
      public static void Main()
      {
         Sample r = new Sample();

         r.a = 10;
         r.b = 20;

         Test(r);
      }
      public static void Test (Sample s)
      {
         System.Console.WriteLine(s.a);
         System.Console.WriteLine(s.b);
      }
   }
   class Sample
```

```
    {
        public int a;
        public int b;
    }
```

Referansa Geri Dönen Fonksiyonlar: Fonksiyonların geri dönüş değerleri referans olabilir. Yani fonksiyon içinde new operatörü ile bir nesne tahsis edilip onun adresiyle geri dönülebilir.

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s;

            s = Test();

            System.Console.WriteLine(s.a);
            System.Console.WriteLine(s.b);
        }
        //...
    }
    public static Sample Test()
    {
        Sample s = new Sample();

        s.a = 10;
        s.b = 20;

        return s;
    }
}
class Sample
{
    public int a;
    public int b;
}
}
```

Null Referans Kavramı : Bir referansa henüz değer atamadan onu kullanamayız. Kullanmaya çalışırsak derleme zamanında error oluşur. Null anahtar sözcüğü herhangi bir referansa atanabilir. Null anahtar sözcüğü değer türlerine atanamaz. Bir referansın içinde null varsa ona değer atanma gereksinimi karşılanmıştır. Fakat içinde null bulunan bir referans kullanılırsa ve program çalışırken akış referansın kullanıldığı noktaya geldiğinde Exception oluşarak program çöker.
Referansın içinde null değeri olup olmadığı == ve != operatörleri ile karşılaştırılabilir.
Örneğin:

```
if (s == null)
{
        //---
}
```

Örneğin bir referansa null atayarak o referansın gösterdiği yerdeki nesnenin referans sayacını düşürebiliriz.

```
Sample s=new Sample ();
//...
s=null;
```

Burada nesnenin referans değeri sıfıra düşürülmüş ve çöp toplayıcı için seçilebilir duruma getirmiştir.

String Sınıfı: Bir yazı bir grup karakterden oluşan bir kümedir C# da yazılar System isim alanı içindeki String isimli sınıf nesnesinin içinde tutulur. Bir String nesnesinin içinde yalnızca yazı değil onun karakter uzunluğu da tutulmaktadır. System.String sınıfı çok kullanıldığı için "String" anahtar sözcüğü ile ifade edilmektedir. String nesneleri programcı tarafından new operatörü ile yaratılmaz. Derleyici ne zaman iki tırnak içinde bir yazı görse kendisi new operatörü ile heap te bir string nesnesi yaratır. Yazıyı ve uzunluğunu bu nesnenin içine yerleştirir. Bu işlemden nesnenin referansını üretir. Bu durumda biz iki tırnak ifadesini bir string referansına atamalıyız.

```
String s;
s = "Ankara";
```

Yine String nesneleri de sıradan nesnelerdir. Bunları da hiç bir referans göstermezse Garbage Collector siler.
Console sınıfının Write ve WriteLine fonksiyonlarına string referansı verilirse bu fonksiyonlar referansın gösterdiği nesnenin içindeki yazıyı ekrana yazarlar. Yazılarıda fonksiyonlara string sınıfı yoluyla geçirebiliriz. String sınıfı çeşitli operatör fonksiyonlarını içermektedir. Örneğin iki string referansını == ve != operatörleriyle karşılaştırma işlemine sokabiliriz. Bu durumda referanslar içindeki adresler değil referansların gösterdiği nesnenin gösterdiği yazılar karşılaştırılmaktadır.

Örneğin:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         string psswd="maviay";
         string s;

         System.Console.WriteLine("Enter password: ");
         s = System.Console.ReadLine();

         if (psswd==s)
            System.Console.WriteLine("ok...\n");
         else
            System.Console.WriteLine("invalid password!...");
      }
   }
}
```

String sınıfının tuttuğu yazının herhangi bir karakterine köşeli parantez operatörü ile erişilebilir. Çünkü string sınıfının bir indeksleyicisi vardır. Yazının ilk karakteri 0. cı indekstedir. Köşeli parantez içindeki sayı negatif ya da yazının limitleri dışında oluşursa exception oluşur.
String sınıfının statik olmayan length isimli property elemanıyla yazının uzunluğu elde edilebilir.

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         string s = "Ankara";

         for (int i = 0; i<s.Length; ++i)
            System.Console.Write(s[i]);

         System.Console.WriteLine();
      }
   }
}
```

Fakat yazının herhangi bir karakterini değiştiremeyiz. String sınıfının tuttuğu yazını karakterleri değiştirilemez. Fakat değişmiş yeni bir yazı elde edilebilir. (String sınıfına benzeyen String Builder adlı başka bir sınıf da vardır. Bu sınıfın en önemli farkı karakterlerin değiştirilebilimesidir.
String sınıfının statik olmayan **ToLower ToUpper** fonksiyonları *public string ToUpper ()* yazıyı büyük harf ve küçük harfe dönüştürür. Bu fonksiyonlar mevcut yazıyı değiştirmezler, değiştirilmiş bir yazı verirler.

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         string s = "Ankara";
         string t;

         t = s.ToUpper();
         System.Console.WriteLine(t);
      }

   }
}
```

Bir yazının belirli bir indeksinden belirli bir miktarda karakteri alarak bir string nesnesi biçiminde veren String sınıfının statik olmayan bir **Substring** fonksiyonu vardır.

```
Public string Substring(
        int startindex,
```

```
        int lenght
)
```

Eğer indeks ve uzunluk sınır dışına taşmaya yol açarsa Exception oluşur.

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         string s = "Ankara";
         string t;

         t = s.Substring (2, 4);
         System.Console.WriteLine(t);
      }

   }
}
```

C# da switch parantezinin içindeki ifadeler ve case ifadeleri string türünden olabilir.

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         string cmd;

         for (; ; )
         {
            System.Console.Write("CSD>");
            cmd = System.Console.ReadLine();
            if (cmd == "quit")
               break;
            switch (cmd)
            {
               case "dir":
                  DirCmd();
                  break;
               case "del":
                  DelCmd();
                  break;
               default:
                  System.Console.WriteLine("'{0}' iç yada dış komut çalıştırılabilir bir program veya
dış komut değil", cmd);
                  break;
            }
```

```
            }
        }
        public static void DirCmd()
        {
            System.Console.WriteLine("Dir");
        }
        public static void DelCmd()
        {
            System.Console.WriteLine("Del");
        }
    }
}
```

String sınıfının **Trim** isimli fonksiyonu yazının başındaki ve sonundaki boşluk karakterlerini atarak yeni bir yazı verir.

Public string Trim()

Örnek:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string cmd;

            for (; ; )
            {
                System.Console.Write("CSD>");
                cmd = System.Console.ReadLine();
                cmd = cmd.Trim();
                if (cmd == "quit")
                    break;
                switch (cmd)
                {
                    case "dir":
                        DirCmd();
                        break;
                    case "del":
                        DelCmd();
                        break;
                    default:
                        System.Console.WriteLine("'{0}' ic yada dis komut calistirilabilir program ve toplu
is dosyasi olarak taninmiyor.", cmd);
                        break;
                }

            }
        }
```

```
        public static void DirCmd()
        {
            System.Console.WriteLine("Dir");
        }
        public static void DelCmd()
        {
            System.Console.WriteLine("Del");
        }
    }
}

namespace CSD
{
    class App
    {
        public static void Main()
        {
            string cmd;

            for (; ; )
            {
                System.Console.Write("CSD>");
                cmd = System.Console.ReadLine();
                cmd = cmd.Trim();
                if (cmd.Length == 0)
                    continue;
                if (cmd == "quit")
                    break;
                switch (cmd)
                {
                    case "dir":
                        DirCmd();
                        break;
                    case "del":
                        DelCmd();
                        break;
                    default:
                        System.Console.WriteLine("'{0}' ic yada dis komut calistirilabilir program ve toplu
is dosyasi olarak taninmiyor.", cmd);
                        break;
                }

            }
        }
        public static void DirCmd()
        {
            System.Console.WriteLine("Dir");
        }
        public static void DelCmd()
        {
            System.Console.WriteLine("Del");
        }
```

```
    }
}

namespace CSD
{
    class App
    {
        public static void Main()
        {
            string cmd;

            for (; ; )
            {
                System.Console.Write("CSD>");
                cmd = System.Console.ReadLine();
                cmd = cmd.Trim();
                if (cmd.Length == 0)
                    continue;
                if (cmd == "quit")
                    break;
                switch (cmd)
                {
                    case "dir":
                        DirCmd();
                        break;
                    case "del":
                        DelCmd();
                        break;
                    default:
                        System.Console.WriteLine("'{0}' ic yada dis komut calistirilabilir program ve toplu
is dosyasi olarak taninmiyor.", cmd);
                        break;
                }

            }
        }
        public static void DirCmd()
        {
            System.Console.WriteLine("Dir");
        }
        public static void DelCmd()
        {
            System.Console.WriteLine("Del");
        }
    }
}
```

String sınıfının **Insert** isimli statik olmayan fonksiyonu mevcut yazının belirli bir indeksinden itibaren başka bir yazı insert eder. Tabi ki fonksiyon insert edilmiş yazıyla dönmektedir.

```
public string Insert(
    int startIndex,
```

```
    string value
)

namespace CSD
{
   class App
   {
      public static void Main()
      {
         string s = "EskiŞehir";
         string t;

         t = s.Insert (4, "Yeni");

         System.Console.WriteLine(t);
      }
   }
}
```

String sınıfının çeşitli statik fonksiyonları da vardır. Örneğin **Format** fonksiyonu tamamen Write ve WriteLine fonksiyonlarındaki mantıkla çalışır. Fakat oluşturulan yazı ekrana yazmazda bunu bir string biçiminde verir.

Örneğin:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         int a = 10, b = 20;
         string s;

         s = string.Format("a = {0} b = {1}", a, b);

         System.Console.WriteLine(s);
      }
   }
}
```

String sınıfının **+ operatör fonksiyonu** yeni bir string nesnesinin yaratılmasına yol açar. İki string referansını topladığımızda derleyici tarafından yeni bir string yaratılır. Bu yeni yaratılan string'in içindeki yazı iki yazının birleşiminden oluşur. Böylece toplama işleminden yeni yaratılan string nesnesinin adresi elde edilir.

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         string a = "Ankara",  b = "İzmir",  c;

         c = a + b;

         System.Console.WriteLine(c);
      }
   }
}
```

Standartlara göre aynı assembly (exe ve dll dosyaları kastediliyor) içinde t**amamen özdeş stringler için yeniden yer ayrılmaz.** Tek bir string nesnesi yaratılır. Aynı nesneye ilişkin aynı referans değerleri elde edilir.

Anahtar Notlar: System isim alanındaki Object sınıfının ReferenceEquals isimli statik fonksiyonu iki referans alarak bu referansların aynı nesneyi gösterip göstermediğini belirler. Yani referansların içindeki adreslerin aynı olup olmadığını tespit eder. Bu fonksiyonun geri dönüş değeri bool türden bir değerdir.

Bir string in soluna yapışık olarak **@ karakteri** getirilirse böyle stringlere **dayanıklı stringler** (verbatim strings) denir.

Dayanıklı stringler içindeki ters bölü karakterleri gerçekten ters bölü karakterleridir. Böylece yol ifadeleri gibi ters bölü içeren yazılar daha kolay ifade edilebilir.

Örneğin:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         string path = @"\temp\ali";

         System.Console.WriteLine(path);
      }
   }
}
```

Ayrıca dayanıklı stringler birden çok satıra bölünebilmektedir.

Örneğin:

```
namespace CSD
{
```

```
    class App
    {
        public static void Main()
        {
            string path = @"Ankara
                izmir";

            System.Console.WriteLine(path);
        }
    }
}
```

İki tırnak açılır açılmaz kapatılabilir. Bu durumda derleyici yine bir string nesnesi yaratır fakat içine hiçbir karakter yerleştirmez. Dolayısıyla uzunluğu sıfır olacaktır.

Örneğin:

s = “”

Console sınıfının ReadLine isimli fonksiyonu klavyeden girilen yazıdan bir string oluşturur ve o string referansıyla geri döner.

string ReadLine();

Örnek:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string s;

            s = System.Console.ReadLine();
            System.Console.WriteLine(s);
        }
    }
}
```

Diziler: Elemanları aynı türden olan ve hızlı bir biçimde erişilebilen veri yapılarına dizi denir. Diziler C# da kategori olarak referans türlerine ilişkindir. Yani bir dizi aslında bir sınıftır. Dizi elemanları sınıfın içinde bulunmaktadır. Dizinin uzunluğuda sınıfın içinde saklanır. C# da T bir tür olmak üzere her zaman T[] biçiminde bir tür de vardır. Buna T türünden dizi türü denir. T ister değer türlerine ister referans türlerine ilişkin olsun T[] kategori olarak referans türlerine ilişkindir.

Örneğin:

```
int [] a;
```

Burada a yalnızca bir referanstır. Dizinin elemanları bu referansın gösterdiği yerdeki dizi nesnesinin içinde olacaktır. Dizi tahsis etmenin genel biçimi şöyledir.

new <tür> <[ifade]>

Örnek:

new string[20]

Köşeli parantez içinde tam sayı türlerine ilişkin bir ifade bulunmak zorundadır. Bu dizinin uzunluğunu belirtir. Uzunluk belirten ifade sabit ifadesi olmak zorunda değildir.

Örneğin:
length = int.Parse(System.Console.ReadLine());
int [ ] a = new int[length];
Dizinin herhangi bir elemanına köşeli parantez operatörü ile erişilir. Köşeli parantez içinde tam sayı türlerine ilişkin bir ifade bulunmak zorundadır.
Örneğin: a[i] a referansını gösterdiği yerdeki dizinin i. ci elemanı anlamına gelir. Dizinin ilk elemanı 0. cı indisli elemanıdır. Bu durumda son elemanı n-1 indisli elemanı olur.
Dizinin olmayan bir elemanına erişmeye çalışmak Exception oluşmasına neden olur.

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         int[] a;
         a = new int[10];

         for (int i = 0; i < 10; ++i)
            a[i] = i * i;

         for (int i = 0; i < 10; ++i)
            System.Console.WriteLine(a[i]);
      }
   }
}
```

Bir dizinin uzunluğu dizi nesnesinin içinde tutulmaktadır. Dizilerin Length isimli property elemanları dizinin uzunluğunu verir.

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
```

```
        int[] a;
        a = new int[10];

        for (int i = 0; i < 10; ++i)
            a[i] = i * i;

        for (int i = 0; i < a.Length; ++i)
            System.Console.WriteLine(a[i]);
    }
  }
}
```

Aynı türden iki dizi referansı birbirine atanabilir. Bu durumda iki dizi referansı aynı dizi nesnesini gösteriyor durumda olur.

Örnek:

```
namespace CSD
{
  class App
  {
    public static void Main()
    {
        int[] a;
        a = new int[10];

        for (int i = 0; i < 10; ++i)
            a[i] = i * i;

        int[] b = a;

        for (int i = 0; i < b.Length; ++i)
            System.Console.WriteLine(b[i]);
    }
  }
}
```

Diziler de Heap te yaratıldığına göre onların silinmesi de çöp toplayıcı sistem tarafından gerçekleştirilir.

Dizilerin fonksiyonlara geçirilmesi oldukça kolaydır. Tek yapılacak şey fonksiyonun parametre değişkeni olarak bir dizi referansı almaktır.

Örnek:

```
public static void Disp(int [ ] a)
{
          for (int i = 0; i < a.Length; ++i)
            System.Console.WriteLine(a[i]);
}
```

Fonksiyonun geri dönüş değeri bir dizi türünden olabilir.

Örneğin:

```
public static int[ ] GetNumbers()
{
        //---
}
```

Programcı fonksiyon içine diziyi tahsis ederek bunun referansı ile geri dönebilir.

Örnek:

```
public static int[ ] GetNumbers()
{
        int [] a = new int [10];

          for (int i = 0; i < a.Length; ++i)
           a[i] = i;

        return a;
}
```

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         int[] x;
         x = GetNumbers();

         for (int i = 0; i <x.Length; ++i)
            System.Console.WriteLine(x[i]);
      }
      public static int[] GetNumbers()
      {
         int[] a = new int[10];
         for (int i = 0; i < a.Length; ++i)
            a[i] = i;
         return a;
      }
   }
}
```

new operatörü ile dizi tahsis edildiğinde hemen tahsis edilmiş dizi elemanlarına ilk değerler verilebilir.

Örneğin:

```
a = new int [5] {1, 2, 3, 4, 5}
```

görüldüğü gibi köşeli parantezlerden sonra değer listesi yazılmıştır. Eğer küme parantezleri ile değer ataması yapılmışsa köşeli parantezler içindeki uzunluk ifadesinin sabit ifadesi olması

zorunludur.

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         int[] a;

         a = Foo();

         for (int i = 0; i <a.Length; ++i)
            System.Console.WriteLine(a[i]);
      }
      public static int[] Foo()
      {
         return new int[3] { 1, 2, 3 };
      }
   }
}
```

Anahtar Notlar: C# ta bir kaç durumda sondaki fazladan virgül sintaks hatasına yol açmamaktadır. Örneğin dizilerde ilk değer verirken sondaki fazladan virgül error a yol açmaz.

Örnek: *a = new int[3] {1, 2, 3, };*

Küme parantezleriyle tahsisata değer atarken köşeli parantezin içi boş bırakılabilir. Bu durumda derleyici küme parantezi içindeki değerleri sayar ve dizinin o uzunlukta açılmış olduğunu kabul eder. Örnek yukarıda...
Burada köşeli parantezin içine üç yazmakla yazmamak aynı anlamdadır.

Örnek: *a = new int[] {1, 2, 3, };*
Fakat dizi sayısını yazmak programı kontrol eden için faydalıdır.

Küme parantezleri içindeki değerler sabit ifadesi olmak zorunda değildir.

Örneğin:

```
int x = 10, y = 20;
int[ ] a = new int [ ] {x, y};
```

Aynı türden iki dizi referansı birbirine atanabilir. Fakat farklı türler den iki dizi referansı birbirine atanamaz.

Örnek:

```
int [ ] a;
long [ ] b;
b = a; error
```

Bir dizi referansına doğrudan küme parantezleri içinde ilk değer verilebilir.

Örneğin:
int [ ] a = {1, 2, 3};

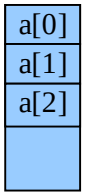
Bu durumda derleyici new işlemini kendisi yaparak verilen ilk değerleri dizi elemanlarına yerleştirir. Yani bu işlemin aşağıdaki işlemden hiç bir farkı yoktur.
int [ ] a = new int [ ] {1, 2, 3};

Fakat bu kısaltma yalnızca ilk değer verilirken söz konusudur.
int [ ] a ;
a = {1, 2, 3} error

Referans Dizileri:
Sınıflar türünden de diziler oluşturabilir. Bu durumda dizinin her elemanı bir referans olur.

Örneğin:

Sample [ ] a = new Sample [3];



Burada dizinin her elemanı bir referanstır fakat bu referanslar için yer tahsis edilmemiştir. Ayrıca referansların gördüğü yerdeki nesnelerin de tahsis edilmesi gerekir.
```
for(int i = 0; i<a.length; ++i)
        a[i]=new Sample();
```

Aynı işlemler şöylede yapılabilirdi.

```
Sample [ ] a;
a = new Sample [ ] {new Sample(), new Sample(), new Sample(), };
```

Bir dizi new operatörü ile tahsis edildiğinde eğer dizi değer türlerine ilişkinse tüm elemanlarda sıfır bulunur. Eğer referans türlerine ilişkinse tüm elemanlarına null yerleştirilir. Sıfırlama new operatörü tarafında yapılır.
String referanslarından oluşan bir dizi aşağıdaki gibi yaratılabilir.

```
String [] names;
names = new string[] {"Ali", "Veli", "Selami", "Ayşe", "Fatma"};
```

Örnek:
```
namespace CSD
{
   class App
   {
      public static void Main()
```

```
    {
        string[] names;
        names = new string[] { "Ali", "Veli", "Selami", "Ayse", "Fatma" };

        for (int i = 0; i < names.Length; ++i)
            System.Console.WriteLine(names[i]);
    }
  }
}
```

Aynı işlem ilk değer verme biçiminde de yapılabilir.

```
namespace CSD
{
  class App
  {
    public static void Main()
    {
        string[] names = { "Ali", "Veli", "Selami", "Ayse", "Fatma" };

        for (int i = 0; i < names.Length; ++i)
            System.Console.WriteLine(names[i]);
    }
  }
}
```

Çok Boyutlu Diziler: En yaygın kullanılan çok boyutlu diziler iki boyutlu matrislerdir. Çok boyutlu dizi referansları köşeli parantez içine virgül koyularak tanımlanır. O halde dizi referansları köşeli parantez içindeki virgül sayısından bir fazla boyut içerir. Çok boyutlu dizileri tahsis ederken new operatörü ile köşeli parantez içerisine boyut uzunluğu girilmelidir.

Örneğin:

```
int [,] a = new int [5, 10];
```

Burada 5 satırlı 10 sütünlu bir matris söz konusudur.

Erişim uygulanırken a[i, k] biçiminde ifade kullanılır.

Çok boyutlu diziler **Length property** si uygulandığında dizinin toplam eleman uzunluğu elde edilir. Dizilerin **Rank Property l**eri boyut uzunluğunu vermektedir.

Örneğin:

```
namespace CSD
{
  class App
  {
    public static void Main()
    {
        int[,] a = new int[10, 10];
```

```
            System.Console.WriteLine(a.Rank);
        }
    }
}
```

Her dizi sınıfının ayrıca int parametreli bir **GetLength** isimli bir int parametreli bir fonksiyonu vardır. Bu fonksiyona boyut indeksi parametre olarak verilirse biz o boyutun uzunluğunu verir.

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int[,] a = new int[10, 5];

            System.Console.WriteLine(a.GetLength(0));
        }
    }
}
```

0, 1, 2... parametre içine yazılarak ilk sıradan başlayarak boyutlarını yazar.

Çok boyutlu dizilere iç içe küme parantezleriyle değer atanabilir.

Int [,] a = new int [2, 3] {{1, 2, 3}, {4, 5, 6}};

Aynı işlem şöyle de yapılabilirdi.
Int [,] a = {{1, 2, 3}, {4, 5, 6}};

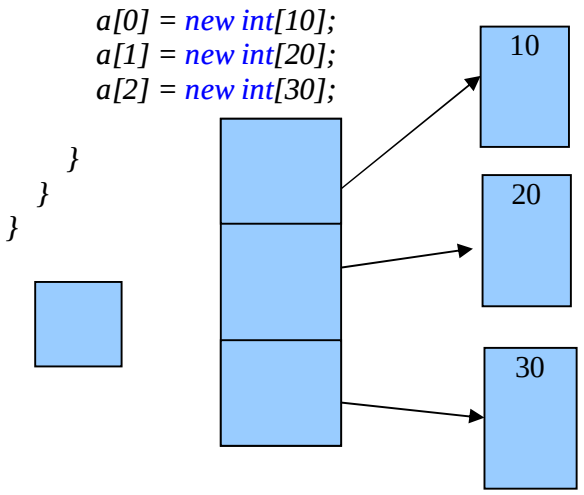
Dizi Dizileri: Diziler de birer referans olduğuna göre dizi referanslarını içeren dizi olabilir. Yani bir dizinin her elemanı başka bir dizinin referansını tutuyor olabilir. Bildirimleri birden fazla köşeli parantez ile yapılır.

int a [ ] [ ] a;

Burada a her elemanı int türden dizi olan bir dizidir.
Dizi dizileri new ile tahsis edilirken birden fazla köşeli pantez kullanılır. Örneğin:
int [ ] [ ] a = new int [10] [ ];

Burada dikkat edilmesi gereken ilk köşeli parantezin dizinin uzunluğunu verdiği ilk köşeli parantez elle kapatıldıktan sonra geri kalanlarının dizinin türüne ilişkin olduğudur. Başlangıçta dizinin her elemanında null değeri vardır.

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int[] [] a = new int[3] [];
```



Burada tek boyutlu bir dizi söz konusudur. Yani a.Length 3 değerini verir. Burada a[0] ifadesinin tür int [] dir. a[0] length ifadesi 10 değerini verir. Buna **Jagger array** deniyor c# terminolojisinde.
Bu durumda a[i][k] ifadesi, dizinin i.ci elemanı ile söz konusu edilen dizinin k.cı elemanını belirtir.

Burada a'nin türü int [] [] dir.
a[i] nin türü int []dir.
a[i][k]'nın türü int türdendir.

1. Örnek: *int [] [,] a;*
Burada a'nın rankı birdir. Yani a tek boyutlu bir dizidir. Dizinin her elemanı iki boyutlu bir dizinin adresini tutmaktadır.

2. Örnek: *int [,] [ ] a;*

Burada a'nın rankı 2 dir iki boyutlu bir dizidir. Matrisin her elemanı bir boyutlu bir dizi tutmaktadır. Yani birinci örnek her elemanı matris olan bir diziyi ikinci örnek her elemanı dizi olan bir matrisi tutmaktadır.

foreach Döngüsü: foreach döngüsü sadece dizilerde değil enumerable arayüzünü destekleyen her türlü sınıf ve yapıyla kullanılabilir.
foreach döngüsünün genel biçimi şöyledir.

foreach (<tür> <değişken> in <dizilim>
<deyim>

Burada dizilim enumerable arayüzünü destekleyen herhangi sınıf ya da yapı anlamında kullanılmaktadır. Her yineleme de dizilimin sonraki elemanı döngü değişkenine taşınır. Dizilim bitince döngü de biter.

```
int [ ] a = {1, 2, 3, 4, 5};
foreach (int x in a)
        System.Console.WriteLine(x);
```

Burada ilk yinelemede x'in içinde önce 1 sonra 2 en son 5 olması gerekir. Şüphesiz en normal durum döngü değişkenin türünün dizilimin türünden olmasıdır.

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         int[] a = { 1, 2, 3, 4, 5 };

         foreach (int x in a)
            System.Console.WriteLine(x);
      }
   }
}
```

Örneğin:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         string [] names = new string [] {"Ali", "Veli", "Selami", "Ayse", "Fatma"};

         foreach (string name in names)
            System.Console.WriteLine(name);
      }
   }
}
```

Burada dizinin her elemanı bir string referansıdır. Dolayısıyla döngü değişkeni de bir stringdir.

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         int[][] a = new int[][] { new int[] { 1, 2 }, new int[] { 3, 4, 5 } };

         foreach (int[] x in a)
         {
            foreach (int y in x)
               System.Console.Write("{0} ", y);
            System.Console.WriteLine();
         }
      }
   }
```

```
}
```

Döngü değişkeni read only dir. Biz döngü değişkenini döngü içinde kullanabiliriz fakat atama yapamayız.

```
foreach (int x in a)
        x = x + " ----> error
```

Her yinelemede dizilimin sonraki elemanının tür dönüştürme operatörü ile dönüştürülüp döngü değişkenine atandığı kabul edilir.

Örneğin:

```
long [] a ;= {1, 2, 3, 4, 5};
foreach (int x  in a) ---->geçerli
{
        //----
}
```
Bu döngü tamamen aşağıdaki gibidir.
```
long [] a ;= {1, 2, 3, 4, 5};
for (int i = 0; i<a.Length; ++i)
{
        int  x = (int)a[i];
        //----
}
```

Görüldüğü gibi foreach döngüsü bir dizinin elemanlarını tek tek gözden geçirmek için pratik bir kullanım sunmaktadır.

Bir Klasör İçindeki Dosyaların Elde Edilmesi: System isim alanının içindeki **IO isim alanının** içinde bulunan **Directory sınıfının GetFiles isimli bir statik fonksiyonu** bir yol ifadesini parametre olarak alır, ilgili klasördeki tüm dosyaların isimlerini yol ifadeleri ile birlikte bir string dizisine yerleştirir ve o string dizisinin referansıyla geri döner.

```
public static string[] GetFiles(){
        string.Format
}

namespace CSD
{
   class App
   {
      public static void Main()
      {
         string[ ] files;

         files = System.IO.Directory.GetFiles(@"c:\windows");

         foreach (string file in files)
            System.Console.WriteLine(file);

      }
```

```
    }
}

namespace CSD
{
    class App
    {
        public static void Main()
        {
            string[] files;

            files = System.IO.Directory.GetFiles(@"c:\windows");

            foreach (string file in files)
            {
                string lowerfile = file.ToLower();

                if (lowerfile.EndsWith(".exe"))
                    System.Console.WriteLine(file);
            }

        }
    }
}
```

Örneğin System.IO isim alanındaki File isimli sınıfın **Copy** isimli statik fonksiyonu dosya kopyalamak için kullanılabilir.

```
public static void Copy (
        string sourceFilename;,
        string destFileName;
        bool overWrite;
)
```

Fonksiyonun 1. parametresi kaynak dosyanın yol ifadesi 2. parametresi hedef dosyanın yol ifadesi 3. parametre true ise ve hedef dosya varsa üstüne yazma yapar. False ise ise fonksiyon başarısız olur. Fonksiyon herhangi bir biçimde başarısız olursa Excepiton oluşur.

System.IO isim alanındaki **path** isimli sınıfın çeşitli statik fonksiyonları parametre olarak bir yol ifadesi alır ve o yol ifadesini ayrıştırır.

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string[] files;

            files = System.IO.Directory.GetFiles(@"c:\windows");
```

```
            foreach (string file in files)
            {
                System.Console.WriteLine(System.IO.Path.GetFileName(file));
            }

        }
    }
}
```

Sınıf Çalışması:
Path simili GetFileName isimli fonksiyonunu kendi sınıfınızda yazınız.

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string[] files;

            files = System.IO.Directory.GetFiles(@"c:\windows");

            foreach (string file in files)
            {
                System.Console.WriteLine(GetFileName(file));
            }
        }
        public static string GetFileName(string path)
        {
            int index = path.LastIndexOf('\\');

            string result = path.Substring(index + 1, path.Length - index - 1);

            return result;
        }
    }

}
```

Sınıf Çalışması:
Belirli bir klasördeki dosyaları başka bir klasöre kopyalayınız.

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string[] files;
            string sourceDir = @ “C:\Dev-Cpp\Lang”;
```

```
        string sourceDir = @ "E:\cs-saper\temp";

          files = System.IO.Directory.GetFiles(sourceDir);

          foreach (string file in files)
        {
           System.IO.File.Copy(file, destDir + @ "\" + System.IO.Path.GetFileName(file)
        }
      }
   }
}
```

Anahtar Notlar: String sınıfının aşağıdaki string isimli fonksiyonu bir yazıyı belirtilen ayıraçlardan parçalara ayırır.

```
Public string [] split (
        char[] separator
)
```

Örneğin: "Ali, Veli, Selami"
Aslında split fonksiyonunun stringli versiyonuda vardır.

Params Dizi Parametreleri: Bir fonksiyonun parametre değişkeni başında params anahtar sözcüğü olacak şekilde bir dizi olabilir.

Örneğin:

```
public static  void Func [params int [] a)
{
        //...
}
```

Params anahtar sözcüğü yalnızca dizi fonksiyon paramtrelerinde kullanılır.
Params dizi parametreli bir fonksiyon normal bir biçimde bir dizi argümanıyla çağrılabilir.

```
public static void Func (params int[] a)
{
        foreach (int x in a)
                System.Console.WriteLine(x);
}
//...
func(new int [] {1, 2, 3, 4})
```

Params dizi parametreli fonksiyonlar ayrıca bir argüman listesiylede çağrılabilir. Bu durumda derleyici uygun türden kendisi bir dizi yaratır girilen argümanları diziye yerleştirir ve fonksiyona diziyi yollar.

```
Public static void Func (params int[] a)
{
        foreach (int x in a)
                System.Console.WriteLine(x);
}
//...
func(1, 2, 3, 4) Bu bir dizidir argümanla çağrılıyor.
```

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         System.Console.WriteLine(Add(1, 2, 3, 4));
      }
      public static int Add(params int[] a)
      {
         int total = 0;

         foreach (int x in a)
            total += x;
         return total;
      }
   }

}
```

Eğer fonksiyonun başka parametreleri de varsa params dizi parametresinin sonda bulunması zorunludur. Yani fonksiyonun params dizi parametresi sonda olmak zorundadır.

Anahtar Notlar: Her ne kadar stringler bir char dizisine benziyorsa da char dizisi ile string aynı anlama gelmez. String sınıfının statik olmayan parametresiz ToCharArray isimli fonksiyonu yazının içindeki karakterleri char türden bir diziye dönüştürmektedir.

Örneğin:

```
string s = "Ali";
char a;
a = s.ToCharArray();
```

Farklı Parametrik Yapılara İlişkin Aynı İsimli Fonksiyonlar: Parametrik yapıları farklı olmak koşuluyla bir sınıflta aynı isimli fonksiyonlar bulunabilir. Parametrik yapıların farklı olması demek parametrelerin türce ve/veya sayıca farklı olması demektir. Parametre değişken isimlerinin önemi yoktur.
Örneğin aşağıdaki sınıfta üç func fonksiyonu da aynı anda bulunabilir.

```
Class Sample
{
        public static void Func() //Parametresiz Func fonksiyonu
        {
                //...
        }
        public static void Func(int a) // int parametreli Func fonksiyonu
        {
                //...
        }
        public static void Func(long a) // long parametreli Func fonksiyonu
        {
                //...
        }
```

Aynı isimli fonksiyonların aynı sınıfta bulunabilmesi durumuna nesne yönelimli programlama

tekniğinde **"function overload"** denilmektedir. Şüphesiz farklı sınıflarda tamamen aynı isimli ve aynı parametrik yapıya sahip fonksiyonlar bulunabilir.

Geri dönüş değeri farklılığı dikkate alınmamaktadır. Yani aynı isimli ve aynı parametrik yapıya sahip fakat geri dönüş değerleri türleri farklı olan fonksiyonlarda aynı sınıfta bulunamaz. Benzer biçimde statik olup olmamanında bir etkisi yoktur. Fonksiyonların erişim belirleyicileri de etkili olamaz.

Bir fonksiyonun ismi ve sırasıyla parametre türlerinin oluşturtuğu kümeye fonksiyonun imzası **"signature"** denilmektedir. İmzası aynı olan fonksiyonlar aynı sınıf içinde bulunamaz.

Örneğin:

```
class Sample
{
        public static void Func(int a, long b) //imzası Func, int, long
        {
                //...
        }
        public static void Func(double a, double b) // imzası Func, double, double
        {
                //...
        }
```

O halde özetle sınıf içinde aynı imzaya sahip birden fazla fonksiyon bulunamaz.
İmza da parametre türlerinin sırası da etkilidir.
Örneğin:
***Func, int, long** imzası ile **Func, long, int** imzaları farklıdır.*

Overload Resolution İşlemi: Aynı isimli bir fonksiyon çağrıldığında derleyicinin hangi fonksiyonun çağrılmış olduğunu belirleyebilmesi gerekir. Bu işlem **overload resolution** denilmektedir. Overload resolution işleminin özet kuralı şöyledir:
Çağrılma ifadesindeki argümanların türlerine bakılır bu türlerle birebir uyuşan parametre yapısına sahip fonksiyonun çağrıldığı kabul edilir. Peki ya tam uyuşan bir fonksiyon varsa? İşte overload resolution işleminin bazı ayrıntıları burda devreye girer.

Aynı isimli bir fonksiyon çagırıldıgında hangi fonksiyonun çağrılmış olduğu üç aşamada belirlenir;
Aday fonksiyonlar belirlenir
Aday fonksiyonlardan bazıları atılarak uygun fonksiyon kümesi elde edilir.
Uygun fonksiyonlar yarışa sokularak en uygun fonksiyon belirlenir.
Eger en uygun fonksiyon varsa çağrılır yoksa çagırma errorle sonuçlanır.
Çağrılma ifadesindeki isimle, aynı isimde olan sınıfın tüm fonksiyonları aday fonksiyonlardır

```
    class Sample
        {
                public static void Func(int a long b) // 1   //char'ı int'e atayabilirz
                {
                        //.....
                }

                public static void Func(int a, int b)   // 2
                {
                        //.....
```

```
        }

        public static void Func(char a, int b)          // 3
        {
                //.....
        }

        public static void Func(float a, float b)       // 4
        {
                //.....
        }

        public static void Func(double a, double b) // 5
        {
                //.....
        }

        public static void Func()                       // 6
        {
                //.....
        }
        public static void Func(int a)                  // 7
        {
                //.....
        }
        public static void Foo()                        // 8
        {
                //.....
        }
    }
//...
Sample.Func('a', 20);
```

Burada 1, 2, 3, 4, 5, 6, 7 numaralı fonksiyonlar **aday fonksiyonlardır(cadidate method)**. Çagrılma ifadesindeki argumanlarla parametre değişkenleri arasında karşılıklı olarak doğrudan otomatik dönüştürmenin yani atamanın mümkün olduğu fonksiyonlar **uygun fonksiyonlardır. (applicable fonksiyonlar)**

Şüphesiz fonksiyonun uygun olması için öncelikle parametre değişkeni sayısının aynı olması gerekir. Örneğimizde 1,2,3,4,5 numaralı fonksiyonlar uygun fonksiyonlardır.

Uygun fonksiyonlar yarışa sokulur ve **en uygun (the most applicable) fonksiyon** elde edilir. En uygun fonksiyon öyle bir fonksiyondur ki her parametresi çağrılma ifadesindeki argumanlar dikkate alınarak, diğer uygun fonksiyonlarla yarışa sokulduğunda daha iyi bir dönüştürme sağlayan ya da daha kötü olmayan dönüştürme sağlayan fonksiyondur.

Otomatik dönüştürmeler arasında kalite farkı vardır. Yarış dönüştürme kalitelerine bakılarak yapılmaktadır.

Otomatik dönüştürme kalitesini belirleyen üç farklı durum vardır.
t1, t2, t3 birer tür olmak üzere t1-> t2 t->3 dönüştürmelerinin kalitesi şöyle açıklanmaktadır;

1-) t2 ile t3 den hangisi t1 ise o dönüştürme daha iyidir, yani dönüştürmek istenen tür aynı tür ise her zaman bu dönüştürme diğerlerinden kalitelidir.

Örn: char->char
char -> int dönüştürmesi yarışa sokulsa char -> char dönüştürmesi daha iyidir.

2-) Eğer t2'den t3'e otomatik dönüştürme var fakat t3den t2ye yoksa t1->t2 dönüştürmesi t2->t3 den daha iyidir.

Örneğin int-> long ile int->float dönüştürmeleri yarışa sokulduğunda int dönüştürmesi daha iyidir.
char -> int ile char ->long dönüşe sokuldugunda char->int dönüştürmesi daha iyidir.

Ne t2'den t3'e ne de t3'den t2'ye dönüştürme var ise işaretli türe dönüştürme daha iyidir.
Örneğin ushort ->int ushort->uint dönüştürmeleri yarışa sokulsa ushort->int daha iyidir.

short ->uint
short ->long } bunlar daha applicable bile değil, direk başta elenirler

o halde örneğimizde "3" nolu fonksiyon en uygun fonksiyon olarak seçilir.

Çağırma şöyle olsun;

Sample.Foo(10, 'a');
Burada 1,2,4,5 nolu fonksiyonlar uygun fonksiyonlardır. (3 olmaz int'den char'a dönüştürme yoktur.)
4. parametreyi yarışa sokalım 1 ve 2 nolu fonksiyonlar birbirlerdinden iyi olmayan fakat kötüde olamayan bir dönüştürme sağlarlar ve yarışa devam ederler.
Görüldüğü gibi 2.parametreler yarışa sokuldugunda char->int dönüştürmesi daha iyidir, o halde 2 nolu fonksiyon en uygun fonksiyondur.
2 nolu fonksiyonun her bir parametresi, ya diğerlerinden daha iyi ya da daha kötü olmayan bir dönüştürme sağlar.

Bazen en uygun fonksiyon bulunamayabilir;

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         Sample.Foo(10, 20);
      }
   }

   class Sample
   {
      public static void Foo(int a, long b)
      {
      //..
```

```
        }

        public static void Foo(long a, int b)
        {
            //..
        }
    }
}
```

//hata verir

4. parametereler yarışa sokuldugunda 1. Fonksiyon daha iyi, 2.parameterelre yarışa sokuldugunda 2.parametereler daha iyidir.tüm parametreleri daha kötü dönüştürme sunmayan fonksiyon yoktur.

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample.Foo(100, 2.3);
        }
    }

    class Sample
    {
        public static void Foo(int a, long b)
        {
            System.Console.WriteLine("int, long");
        }

        public static void Foo(int a, float b)
        {
            System.Console.WriteLine("int, float");
        }
        public static void Foo(float a, double b)
        {
            System.Console.WriteLine("float, double");
        }
    }
     }     //3. Çalışır ekranda,   “float,double”  yazar
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample.Foo(100, 100);
        }
    }
```

```
    class Sample
    {
        public static void Foo(byte a, long b)
        {
            System.Console.WriteLine("byte, long");
        }

        public static void Foo(double a, float b)
        {
            System.Console.WriteLine("int, float");
        }
        public static void Foo(float a, double b)
        {
            System.Console.WriteLine("float, double");
        }
    }
     }                    //1.seçilir..."float,double"...yazar



namespace CSD
{
    class App
    {
        public static void Main()
        {

            Sample.Foo(100, 100);
        }
    }

    class Sample
    {
        public static void Foo(byte a, long b)   //byte'dan double'a dönüştürme var ama tam tersi yok
        {
            System.Console.WriteLine("byte, long");
        }

        public static void Foo(double a, float b)
        {
            System.Console.WriteLine("byte, long");
        }
        public static void Foo(float a, double b)
        {
            System.Console.WriteLine("int, float");
        }
    }
}
```

Aynı isimli fonksiyonları nesne yönelimli programlama teknigi nazarında önemi:
Nesne yönelimli programlama tekniği büyük projelerde oluşan algısal karmaşıklığı gidermeyi hedeflemektedir. Benzer işlem yapan fonksiyonlara aynı isimleri vermek, sanki çok sayıda

fonksiyon varmış duygusundan uzaklaştırır, tek bir fonksiyon varmış algısına yakınlaştırır.

Nesne yönelimli programlama tekniği büyük ölçüde insan, nesne ilişkisi ve algılayışı dikkate alınarak tasarlamıştır. Aslında doğada hiç birşey birbirinin aynı değildir mantığını içerir. Örneğin rengi ve modeli aynı olan iki sandalye bile birbirinin aynı değildir, fakat biz bu iki sandalyeye aynıymış gibi algılamaktayız. Dahası bizim için renkleri aynı olan sandalyelerde aynı değildir. İşte tıpki buradaki ilişkide olduğu gibi iki WriteLine fonksiyon birbirininin aynısı olmaya bilir fakat işlemleri birbirine çok benzemektedir, bunlara aynı ismi verirsek öğrenmeyi kolaylaştırırız.

```
namespace CSD
{
   class App
   {
      public static void Main()
      {

         System.Console.WriteLine(100L);
      }
   }

   class Sample
   {
      public static void Foo(int a, long b)
      {
         System.Console.WriteLine("int, long");
      }

      public static void Foo(byte a, long b)
      {
         System.Console.WriteLine("byte, long");
      }
      public static void Foo(double a, float b)
      {
         System.Console.WriteLine("double, float");
      }
   }
}       //çıktısı 100
```

Özetle benzer işlemleri gerçekleştiren fakat aralarında küçük farklılık olan fonksiyonlara aynı isimleri vermeliyiz.
.net sınıf kütüphanesindek sınıflarda aynı isimli fonksiyonlara çok sık rastlanmaktadır. Programcı bir fonksiyon öğrenirken aynı isimli benzerlerinide incelemelidir.

Sınıfların Başlangıç Fonksiyonları:

New operatörü ile bir sınıf türünden nesne tahsis edildiğinde new operatörü tahsisattan sonra sınıfın isminde **başlangıç fonksiyonunu(Constructor)** otomatik olarak çağırır. Başlangıç fonksiyonları sınıf ismiyle aynı fonksiyonlardır.
Başlangıç fonksiyonlarının geri dönüş kavramı yoktur. Geri dönüş değeri yerine hiçbir şey yazılmaz (void dahi yazılmaz) Başlangıç fonksiyonlarında return deyimi kullanılabilir fakat return anahtar sözcüğü yanına bir şey yazılamaz.

Örnek:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         Sample s;

         s = new Sample();
      }
   }

   class Sample
   {
      public Sample()
      {
         System.Console.WriteLine("Constructor");
      }
      //....
   }
}
```

Sınıfın parametrik yapıları farklı olmak koşuluyla birden fazla başlangıç fonksiyonu olabilir. Sınıfın parametresiz başlangıç fonksiyonuna **default başlangıç fonksiyonu (default constructor)** denilmektedir.
Aslında new ile tahsisat sırasında parantezler içine bir argüman listesi yazılabilir. Bu durumda tıpkı fonksiyon çağrılmasında olduğu gibi en uygun başlangıç fonksiyonu belirlenir ve o çağrılır. Aşağıdaki örnekte sınıfın int, int türü başlangıç fonksiyonu çağrılacaktır.

Örnek:
```
s =newSample (10, 26);
```

```
s =new Sample ();
```
Burada ise sınıfın default başlangıç fonksiyonu çağrılır.

Eğer programcı sınıf için hiçbir başlangıç fonksiyonu yazmamışsa default başlangıç fonksiyonu içi boş olarak derleyici tarafından yazılmaktadır. Bugüne kadar başlangıç fonksiyonun yazmamaktan dolayı sorun çıkmamasının nedeni budur.

Başlangıç fonksiyonları neden kullanılır? Başlangıç fonksiyonları sınıfın veri elemanlarına birtakım ilkdeğerleri atamak için ve gerekli bazı ilk işlemleri gerçekleştirmek için kullanılır. Yani başlangıç fonksiyonları nesneyi kullanıma hazır hale getirmektedir. Şüphesiz bir sınıf için başlangıç fonksiyonu gerekmeyebilir. Bu durumda programcı sınıf için hiç bir başlangıç fonksiyonu yazmaz ve zaten default başlangıç fonksiyonu derleyici tarafından yazılacaktır.
Örneğin DileStream sınıfının başlangıç fonksiyonu dosyayı açabilir, Seri Port işlemlerini yapan SerialPort isimli bir sınıfın başlangıç fonksiyonu seri portu set edebilir.

Örnek:
```
namespace CSD
{
   class App
```

```
    {
        public static void Main()
        {
            Sample s;

            s = new Sample(10, 20);

            s.Disp();
        }
    }

    class Sample
    {
        public int a;
        public int b;

        public Sample(int x, int y)
        {
            a = x;
            b = y;
            //...
        }
        public void Disp()
        {
             System.Console.WriteLine(a);
             System.Console.WriteLine(b);
        }
        //....
    }
}
```

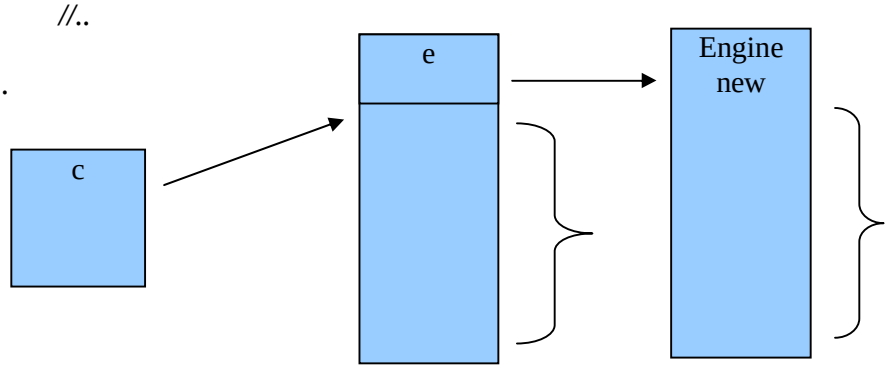
Bir Sınıfın Başka Bir Sınıf Türünden Referans Veri Elemanına Sahip Olması:

Bir sınıf başka sınıf türünden referans veri elemanlarına sahip olabilir. Bu durumda elemana sahip sınıfın başlangıç fonksiyonu içinde biz bir eleman için new operatörü ile tahsisat yapmalıyız.

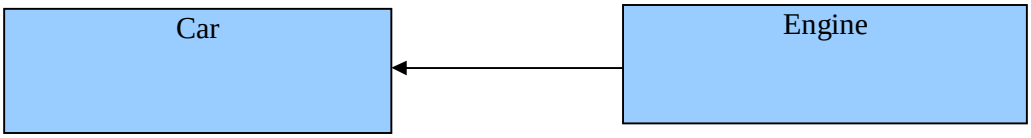
Örneğin otomobil temsil eden Car isimli bir sınıf olsun ve otomobilin motorunu da Engine isimli başka bir sınıfla temsil edelim. Biz bir otomobil nesnesi yarattığımızda onun içinde motorunda olmasını isteriz. Yani otomobil motoru içermektedir. **Nesne Yönelimli programlamada bu ilişkiye "composition" yani "içerme" ilişkisi denir.**

```
class Engine
{
        //...
}
class Car
{
        public Engine e;
        //...
        public Car()
        {
                e = new Engine();
```

```
            //..
        }
        //...
}
```



Bir yazılım projesi nesne yönemli olarak modellenecekse kabaca önce gereksinimler belirlenir sonra projeye konu olan tüm gerçek nesneler ve kavramlar sınıflarla temsil edilir.
Sonra sınıflar arasındaki ilşkiler belirlenir işte içerme ilişkisi bir kapsama durumunda ortaya çıkmaktadır.
Sınıflar arasındaki ilişkiler UML(Unifed Modelling Language) gibi modelleme araçlarında diyagramlarla gösterilmektedir. Sınıf diyagramlarında sınıflar dikdörtgenlerle temsil edilir. Bunların arasındaki ilişkiler betimlenir. İçerme ilşkisi UML sınıf diyagramlarında içi dolu küçük bir baklavacık veya yuvarlakcıkla temsil edilmektedir. Bu içi dolu baklavacık ve yuvarlakçık içeren tarafta bulunmaktadır.



Bu sınıf diyagramına göre Car sınıfı Engine sınıfını içermektedir. Yani Car sınıfının bir veri elemanı Engine sınıfı türünden bir referanstır ve car sınıfının başlangıç fonksiyonu içinde engine nesnesi yaratılmıştır.

Sınıf Bildirimi İçerisinde İlk Değer Verilmesi: new operatörü ile bir sınıf nesnesi yaratıldığı zaman new operatörü önce heap te tahsisatını yapar sonra tahsis ettiği alanı sıfırlar. Daha sonra başlangıç fonksiyonunu çağırır. Sıfırlamadan kastedilen şey sınıfın temel türlerden veri elemanlarına sıfır, referans türünden veri elemanlarına null ve bool türden veri elemanlarına da false değerinin atanmasıdır. Bu durumda biz sınıfın başlangıç fonksiyonu içinde veri elemanlarına değer atamazsak onların içinde sıfır değeri kalacaktır.
Başlangıç fonksiyonu içinde kullandığımız veri elemanları henüz tasis edilmiş nesnenin veri elemanlarıdır.
C# da sınıf bildirimi içinde sınıfın veri elemanlarına ilk değer verilebilir. Bu durumda derleyici verilen ilk değerleri atama komutlarına dönüştürerek sınıfın her başlangıç fonksiyonunun ana bloğunun başına gizlice yerleştirir.
Örneğin:

```
class Sample
{
        public int a;
        public int b = 10;

        public Sample()
        {
```

```
        }
        public Sample (int x)
        {

                a = x;

        }
}
```

Bu bildirimle aşağıdaki tamamen aynı anlamdadır.

```
class Sample
{
        public int a;
        public int b = 10;

        public Sample()
        {
                b = 10;
        }
        public Sample (int x)
        {
                b = 10;
                a = x;

        }
}
```

Bu durumda veri elemanlarına değer verme eylemi şu aşamalardan geçerek gerçekleşmektedir.

1. new operatörü henüz başlangıç fonksiyonunu çağırmadan önce tüm elemanları sıfırlar.
2. Akış başlangıç fonksiyonuna girdiğinde sınıf bildiriminde verdiğimiz ilkdeğerler atanır.
3. Nihayet başlangıç fonksiyonunda bizim yaptığımız atamalar işlem görür.

Veri elemanlarına verilen ilk değerlerin sabit ifadesi olması gerekmez.

Örneğin:

```
class Engine
{
        //...
}
class Car
{
        public Engine e = new Engine();
        //...
}
```

Bu durumda sınıf constructorlarına ilk değer olarak "public Engine e = new Engine();" atanır.

Örneğin:

```
class Sample
{
        public string name = "Kaan Aslan";
```

```
        //...
}
```

Atama deyimleri bildirimdeki sıraya göre yerleştirilmektedir.

İsim Alanları: Bir isim alanı aşağıdaki gibi bildirilir.

```
namespace <isim>
{
        //...
}
```

İsim alanları iç içe bildirilebilir **Örneğin:**

```
namespace  A
{
        class Foo ()
        {
                //...
        }
        namespace B
        {
                class Bar
                {
                        //...
                }
        }
}
```

Hiçbir isim alanı içinde bulunmayan bölge de bir isim alanı belirtir. Bu isim alanına **Global isim alanı (Global namespace)** denilmektedir.
Aynı isim alanı içinde aynı isimli birden fazla sınıf bulunamaz. Fakat farklı isim alanları içinde aynı isimli sınıflar bulunabilir.

```
namespace  A
{
        class Foo
        {
                //...
        }
}
namespace B
{
        class Foo
        {

        }
}
```

Sınıf alanlarının iç içe olması da farketmez.

```
namespace  A
```

```
{
        class Bar
        {
                //...
        }
        namespace B
        {
                class Bar
                {
                        //...
                }
        }
}
```

İç içe sınıf alanları tek hamlede bildirilebilir.

```
namespace A.B
{
        //...
}
```

ile

```
namespace A
{
        namespace B
        {
                //...
        }
}
```

aynı anlama gelir.

Aynı isim alanları içinde bulunan aynı isim alanları birleştirilir. Yani bir isim alanını ikinci kez bildirmek ekleme yapmak anlamındadır.

```
namesapce A
{
        class X
        {
                //...
        }
}
```

Burada A isim alanı içerisinde x ve y sınıfları vardır. Şüphesiz farklı isimli isim alanları içinde aynı isimli isim alanları bulunabilir. Burada test isim alanları birleştirilmez bunlar farklı isim alanlarıdır.

```
namesapce A
{
        namespace Test
        {
                //...
        }
```

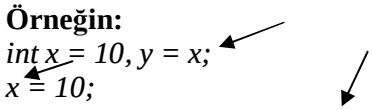
```
}
namespace Bar
{
        namespace Test
        {
                //...
        }
        //...
}
```

İsim Alanlarına Neden İhtiyaç Duyulmuştur? Eskiden programlama dillerinde isim alanı ve buna benzer kavramlar yoktu. Bütün isimler aynı yerde bulunurdu. Halbuki projelerin büyümesiyle ve yazılım endüstrisinin gelişmesiyle isim çakışması bir sorun olarak ortaya çıkmıştır. C# ve .net ortamı bileşen tabanlı geliştirme ortamlarıdır. Yani başkaları tarafından yazılmış olan sınıflar çok yoğun kullanılmaktadır.
Örneğin biz hem A firmasının hem de B firmasının oluşturduğu sınıf kütüphanesini birlikte kullanıyor olabiliriz. A ve B firmalarının birbirleriyle bir ilişkisi olmadığına göre bu firmalar sınıflarına tesadüfen aynı isimleeri vermiş olabilirler işte bu ortamda isim alanları olmasaydı derleyici hangi firmanın sınıfını kullandığını anlayamazdı. Halbuki isim alanları sözkonusu olduğunda her firma kendi sınıflarını kendi isim alanları içinde yazmalıdır.

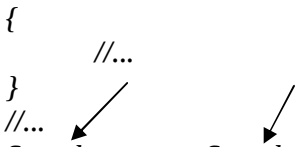
.net'in tüm sınıfları system isim alanı içerisinde ya da system isim alanı içindeki isim alanlarının içindedir. System isim alanı .net'in kendisine ayrılmıştır.

İsim Arama İşlemi: İsim(name) ismini bizim ya da başkasının istediği gibi oluşturduğu her türlü programlama ögesini anlatmaktadır. Örneğin isimalanı ismi, sınıf ismi, değişken isimleri birer isimdirler. Derleyici bir isimle karşılaştığında önce o ismin bildirimini bulmaya çalışır. Eğer ismin bildirimi bulunamazsa derleyici error la karşılaşır. Bildirimle faaliyet alanına sokulan isimler aranmaz. Kullanılan isimler aranır.

Örneğin:

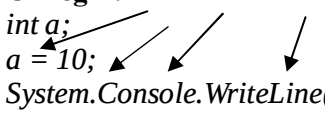


Burada okla gösterilen isimler aranacaktır.



okla gösterilen yerler aranacaktır.

İsim araması niteliksiz (unqualified) ve nitelikli (qualified) olmak üzere ikiye ayrılmaktadır. Doğrudan yazılmış olam isimler niteliksiz arama kuralına göre aranır. Nokta operatörünün sağındaki isimler nitelikli isim arama kuralına göre aranır.
Örneğin:



Oklar niteliksiz yuvarlaklar nitelikli arama kuralına göre aranır.

İsim araması sırasında derleyici bazı faaliyet alanlarına bakar. İsim bulunursa isim araması kesilir. Fakat sonuna kadar bulunamazsa error oluşur.

Niteliksiz İsim Arama İşlemi: İsim bir sınıfın bir fonksiyonu içinde kullanılmış olsun. İsim sırasıyla şu faaliyet alanları içinde aranır.

1. İsim kullanım yerinden yukarıya doğru fonksiyonun yerel bloklarında aranır.
2. İsim fonksiyonun içinde bulunduğu sınıf bildiriminin her yerinde aranır. (Bu durumda bir isim hem yerel değişken olarak hem de sınıfın veri elemanı olarak bildirilebilir. Böyle bir kullanımda isim araması sırasında fonksiyonun yerel değişkeni olan bildirim bulunur.)
3. İsim fonksiyonun ilişkin olduğu sınıfı kapsayan isim alanının her yerinde aranır.
4. İsim fonksiyonun ilişkin olduğu sınıfı kapsayan, isim alanını kapsayan isim alanlarında onların yer yerinde içten dışa doğru sırasıyla aranır.
5. Nihayet isim global isim alanının her yerinde aranır.

Yalnızca fonksiyonun içindeki isimler değil sınıf bildirimi içindeki isimlerde, isim alanı isimleri de aynı biçimde aranır.

Nitelikli İsim Arama İşlemi: Nokta operatörünün sol tarafındaki operand bir isim alanı ismi, bir sınıf ismi, ya da bir referans isimi olabilir. Nokta operatörünün sağındaki isimler nitelikli aranır.

1. Nokta operatörünün solunda bir isim alanı ismi varsa sağındaki isim yalnızca o isim alanının her yerinde aranır.
2. Nokta operatörünün solunda bir sınıf ismi varsa sağındaki isim o sınıfın bildiriminin her yerinde aranır başka bir yerde aranmaz.
3. Nokta operatörünün solunda bir referans varsa sağındaki isim referansın ilişkin olduğu sınıf bildiriminin her yerinde aranır. Şüphesiz bu durumda fonksiyonun statik olmaması gerekir.

System isim alanı ve Console sınıfını biz tanımlamadığımıza göre derleyici bu isimleri nasıl bulmaktadır? İsim alanları ve sınıflar DLL'lerin içine yerleştirilebilir. Sonra bu kütüphanelere referans edilebilir. İşte biz bir dll'le referans ettiğimizde onun içinde tüm isim alanları ve sınıflar derleyici tarafından otomatik olarak görülmektedir. Yani isim araması sırasında derleyici yalnızca kaynak kodumuzu değil referans ettiğimiz dll'leri de dikkate almaktadır.

USING DİREKTİFİ: Using direktifi isim çakışması olmadığı durumda gereksiz niteliklendirmeyi önlemek için düşünülmüştür. Using direktifinin genel biçimi şöyledir.

Using <isim alanı ismi>;

isim alanı ismi niteliksiz bir isim olabileceği gibi nokta operatörleriyle birleştirilmiş iç bir isim alanı ismi olabilir.

Using system; ***niteliksiz***
using A.B.C; ***nitelikli***

Using direktifleri isim alanlarının ilk elemanları olacak biçimde isim alanlarının başına yazılır. Using direktifi ismi global isim alanına yerleştirilecekse en tepeye yerleştirilir.

Using direktifinde iki isim alanı söz konusudur: Direktifin yerleştirildiği isim alanı ve direktifte sözü edilen isim alanı.
Niteliksiz isim sırasında eğer aranan isim using direktifinin yerleştirildiği isim alanında bulunamazsa direktifte belirtilen isim alanında da arama yapılır. Bu tanımda dikket edilmesi gereken noktalar şunlardır.

1. Using direktifi ancak eğer isim direktifin yerleştirildiği isim alanına kadar bulunamamışsa

ve direktifin yerleştirildiği isim alanında da bulunamamışsa devreye girer. Yani örneğin bir sınıf ismi hem using direktifinin yerleştirildiği isim alanında hem de direkitfte belirtilen isim alanında bulunuyorsa bu durum error oluşturmaz. Bu durumda direkitifin yerleştirildiği sınıf alanındaki sınıf ismi bulunur.

2. Using direktifi nitelikli aramada etkili olmaz.

Örneğin:

```
namespace A
{
        using B;
        //...
}

namespace B
{
        class Sample
        {
                //..
        }
}
```

Biz şimdi başka bir isim alanından aşağıdaki gibi bir bildirim yapmış olalım.

```
A.Sample s;
```

Burada Sample ismi nitelikli arandığı için A isim alanında bulunamazsa B'ye bakılmayacaktır.

3. Using direktiflerinde geçişlilik özelliği yoktur. Yani isim using direktifi ile belirtilen isim alanında bulunamazsa artık o isim alanındaki using direktifleri dikkate alınmaz.
4. İsim using direktiflerinin yerleştirildiği isim alanlarında bulunamamış olsun. Eğer isim birden fazla using direktifi ile belirtilen isim alanında bulunursa bu durum error oluşturur. Örneğin biz hem A şirketinin hem de B şirketinin yazmış olduğu sınıfları kullanacak olalım.
   ```
   Using A;
   Using B;
   ```
   Eğer bu iki şirkette bir sınıfına Sample ismini vermişse biz yalnızca Sample dediğimizde bu durum error oluşturur. Artık bu ismi nitelikli olarak belirtmemiz gerekir. Yani A.Sample veya B.Sample biçiminde. Görüldüğü gibi Using direktifi gereksiz niteliklendirmeyi önlemek için düşünülmüştür. Halbuki bu örnekte gerçekten niteliklendirme gerekmektedir.
5. Using direktifiyle, direktifte bildirilen isim alanında arama yapılırken. İsim alanı isimleri bulunamaz.

Örnek:

```
namespace A
 {
  //..
  namespace B
   {
     class Sample
     {
       // ..
     }
```

```
    //..
  }
  //..
}


namespace CSD
{
   using A;
   //..
   class App
   {
      public static void Main()
      {
         B.Sample s;   // error
      }
   }
}
```

Burada B ismi A isim alanında bulunamayacaktır çünkü using direktifi isim alanları isimlerini bulamayacaktır. Bu durumu çözmek için ya sınıf ismi A.B.Sample biçiminde tam olarak belirtilmeli ya da ayrıca using A.B; biçiminde bir direktif daha yerleştirilmelidir.

Anahtar Notlar: Pek çok geliştirme ortamında olduğu gibi visual studio IDE'sinde de bir projenin debug ve release versiyonları vardır. Proje oluşturulduğunda default durum debug versiyondur (zaten bim klasörünün içinde debug oluşturulmuştur). Release versiyonuna geçebilmek için build/configrution manager seçeneğini kullanmak gerekir.

Debug versiyon'da programcının yapabileceği bazı hatalara karşı kontroller vardır. Bu kontroller uygulamayı geliştirirken hatalara karşı programcının uyanık olmasını sağlar. Halbuki release versiyonda bu kontroller yoktur çok daha arı bir kod söz konusudur. Özetle programcı geliştirmeyi debug versiyonunda yapmalı fakat son derlemeyi release kısmında yaparak onu satmalıdır.

DİNAMİK KÜTÜPHANE DOSYALARI

.net dünyasında dosya uzantısı exe ve dll olan... "assembly" denilmektedir. Buradaki assembly dilinin sembolik makine dili anlamına gelen "assembly language sözcüğüyle" bir benzerliği yoktur. .exe ve .dll dosyaları içsel olarak portable executiable formatına uymaktadır. .exe ve .dll dosyaları arasında ciddi bir farklılık yoktur. aralarındaki tek fark exe dosyanın bir başlangıç noktasına sahip olmasıdır (yani main() fonksiyonudur) yani exe dosya dll gibi kullanılabilir fakat dll exe gibi kullanılamaz.

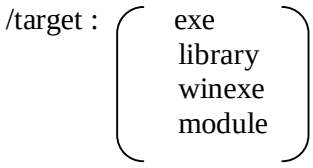
exe ve dll dosylarını .net ortamını göz önüne alarak iki bölüme ayırabiliriz.

1. CLR tarafından çalıştırılabilen exe ve dll dosyaları. Bu anlamda ingilizce manage terimi kullanılmaktadır.(yani clr tarafından yönetiliyor)
2. CLR tarafından çalıştırılamayan doğal exe dosyaları. Bu anlamda ingilizce unmanaged terimi kullanılmaktadır.

Managed ve unmanaged exe ve dll'lerin oluşturulması ve kullanılması birbirinden çok farklıdır. Fakat PE (portable executable) formatı hem managed hemde unmanaged dll ve exe dosyaları için kullanılan ortak formattır.

Managed exe ve dll dosyaları PE formatında ayrıca CLR header ve Metadata section içermektedir. Bu durumda biz bir exe ya da dll dosyasının managed mi yoksa unmanaged mi olduğunu anlamak için en pratik olarak PE dosyası içerisinde bir CLR header bölümü var mı diye bakmalıyız.
Biz bunu pratik olarak ILDASM ilgiyi dosyayı yüklemeye çalışarak anlayabiliriz. Eger dosyayı ILDASM ile çalıştırabiliyorsak managed'dir, yükleyemezsem unmanaged bir dosyadır.

Aslında derleme sonucunda exe ya da dll elde etmek, CSC derleyicisinde /target seçeneğiyle belirlenmektedir. /target seçeneğinin genel biçimi şöyledir.



exe seçeneği Console uygulaması oluşturmak için kullanılır. /target seçeneği belirtilmezse /target:exe belirtilmiş gibi işlem görür. Bu seçenekte kesinlikle herhangi bir sınıfta main() fonksiyonu olmak zorundadır. (aslında CLR için fonksiyonun isminin main() olmasının hiçbir anlamı yoktur. CLR enter point direktifine bakmaktadır.)

Anahtar Notlar:
Aslında bir C# programında birden fazla main() programıda bulunabilir ancak bu durumda derleyici hangi main() fonksiyonunun enter point belirtiğini bilemeyeceğinden bizim /main seçeneği ile bunu belirtmemiz gerekir. Örneğin;

```
CSD/main:CSD.Sample Sample.cs
```

Burada programcı derleyiciye programında birden fazla main() olduğunu fakat enter point olarak CSD isim alanı içerisindeki Sample sınıfındaki main in() fonksiyonun alınması gerektiğini söyler.

*IDE menülerde belirlediğimiz derleme seçeneklerinin CSC derleyicisindeki komut satırlı bir karşılığı vardır. Fakat tersi olmak zorunda değildir, yani gerçek ve en geniş kullanım doğrudan CSC derleyicisinin kullanılmasıyla olur.

Console uygulaması demek işletim sisteminin programı yükledikten sonra hemen bir siyah console ekranını açması demektir.

/target:=win.exe GUI uygulaması oluşturmak için kullanılır. GUI uygulamasıyla konsol uygulaması arasındaki tek fark konsol penceresinin açılmasıdır. /target: library seçeneği dll yaratmakta kullanılır. Normal olarak dll uygulamasında main fonksiyonunun bulunması anlamsızdır. Fakat main fonksiyonu olsada dikkate alınmaz.

Visual Studio IDE sinde dll derlemesi yapmak için project / properties / ouput guide class library olmalıdır.

/target:module seçeneği ile modül dosyası oluşturulur. Modul dosyası oluşturmak ide den yapılamamaktadır.

DLL Dosyalarının Kullanılması: Bir dll li kullanmak için tek yapılacak şey o dll'e referans etmektir. DLL'in içindeki tüm bilgiler zaten meta data olarak bulunmaktadır. Referans etme

işleminden sonra derleyici dll deki tüm bilgileri elde edebilmektedir. Komut satırından dll'i referans etmek için
/referance veya kısaca r seçeneği yazılır

csc /r:<dll dosyasının yol ifadesi>

Örneğin: */r:test.dll Sample.cs Enter*
Burada Sample.cs programı test.dll indeki ögeleri veya sınıfları kullanmıştır. Birden fazla dll'ese

csc /r: doo.dll /r:bar.dll Sample.cs

Bir sınıf bildiriminin soluna da public ya da internal belirleyicileri getirilebilir. Eğer hiçbir belirleyici getirilmemişse sanki internal getirilmiş gibi etki gösterir.

```
public class Sample
{
        //...
}
```

public sınıflar referans edilerek başka bir assembly den kullanılabilir. Halbuki internal sınıflar ancak aynı assembly den kullanılabilir. O halde bir dll deki bir sınıfı kullanmak istiyorsak o sınıfı dll de public olarak bildirmeliyiz.

Örnek:
Test.cs

```
using System;

namespace CSD
{
   public class Sample
   {
      public Sample()
      {

      }
      public void Func()
      {
         Console.WriteLine("Func");
      }
   }
}
```

csc /t:library Test.cs

Sample.cs

```
using System;

namespace CSD
{
   class App
   {
```

```
    public static void Main()
    {
       Sample s = new Sample();

       s.Func();
    }
  }
}
```

***csc /r:Test.dll Sample.cs** referans edilir. Aynı klasör içinde*

DLL le referans ederken dll dosyası başka bir klasörde bulunmuş olabilir. Bu durumda / r seçeneğinde dosyanın yol ifadesi yazılmalıdır. Eğer bir exe bir dll i kullanıyorsa exe çalışırken dll e gereksinim duyar yani biz projemizi başka bir makinaya kuracaksak yalnızca exe yi değil dll dosyalarını da taşımamız gerekir. Program çalışırken dll'in exe ile aynı klasörde bulunması gerekir.

Bir dll le ide den referans etmek için solution explore da reference üzerine gelerek bağlan menüsünden Add referans menüsü seçilir sonrasında browse sekmesine geçilerek referans edilecek dll seçilir. Bir dll le referans edildiğinde ide otomatik olarak ilgili dll lide exe nin bulunduğu klasöre kopyalamaktadır. Böylece çalışırken bir sorun oluşmaz.

Anahtar Notlar: dll uygulaması için önerilebilecek bir yöntem boş bir solution oluşturup bir dll projesi birde exe projesi yaratmaktır. Aslında uygulama (appilication) projeyi oluşturan exe ve dll lerin toplamını anlatmaktadır.

.NET'in sınıfları ve DLL'ler: .net'in içinde yüzlerce sınıf vardır. Bu sınıflar da çeşitli dll lerin içine yerleştirilmiştir. Fakat Microsoft en çok kullanılan sınıfları mscorlib.dll isimli dll'lin içine yerleştirmiştir. CSC derleyicisi hiç /r seçeneği kullanılmasa bile bu dll le içsel olarak referans etmektedir. O halde bir .net sınıfı kullanacaksak onun hangi dll içinde olduğunu bilmeliyiz. Eğer o sınıf mscorlib.dll içinde değilse onun bulunduğu dll le referans etmeliyiz. IDE den .net dll lerine referans etmek için Add referans menüsündeki .net sekmesi kullanılır. O halde bir .net sınıfını hangi sınıf alanında ve hangi dll içinde olduğunu bilmemiz gerekir.

Bir dll li pek çok uygulamanın kullandığını düşünelim. O dll lin kopyasını her uygulamanın klasörüne yerleştirmemiz gerekir. Fakat microsoft bunu engellemek için **G**lobal **A**ssembly **C**ache denilen bir kavram geliştirmiştir. Böylece bir dll **GAC**'a yerleştirilirse artık o dll kullanılırken o dll lin exe nin bulunduğu klasörün içinde bulunması gerekmez. İşte .net'in kendi dll leride GAC içinde bulunmaktadır. Yani biz IDE den referans ettiğimizde ide onun kopyasını çıkarmaz.

Bir dll başka bir dll'i kullanıyor olabilir. Örneğin:
csc /target:library /r:foo.dll bar.cs **Enter**
Burada derleme sonucunda bar.dll elde edilecektir. Fakat bar.cs içinde muhtemelen foo.dll içindeki bir takım sınıflar kullanılılmıştır. Şimdi biz aşağıdaki gibi bir derleme yapalım.

csc /target:exe /r:bar.dll test.cs Enter Biz burada test.exe elde etmek istiyoruz. Yalnızca text.exe yalnızca bar dll le değil foo dll le de bağımlı haldedir. Biz projemizi başka bir makinaya kopyalamak istersek test.exe, foo.dll ve bar.dll dosyalarının hepsini taşımalıyız.

Sınıflarda Temel Erişim Kuralları: Bugüne kadar sınıflarda erişim belirleyici anahtar sözcük olarak hep public kullandık. Aslında herhangi bir sınıf elemanı aşağıdaki erişim belirleyicilerinden birine sahip olabilir.
public

protected
private
internal
protected internal

Sınıf elemanlarının önüne hiçbir erişim belirleyici yazılmazsa default olarak private yazıldığı kabul edilir.

Anlatımsal olarak sınıfın public bölümü denildiğinde sınıfın public elemanları anlaşılmalıdır. Private bölümü denildiğinde private elemanları anlaşılmalıdır.

Sınıflarda temel erişim kuralı iki bölüme ayrılır.

1. Sınıfın dışından yani başka bir sınıfın fonksiyonu içinden ilgili sınıfın yalnızca public bölümüne erişilebilir.
   Örneğin:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         Sample s;

         s = new Sample();
         s.x = 10; // error!
         s.y = 20; // gecerli
         s.Foo(); //  error!
         s.Bar(); // geçerli
      }
   }

   class Sample
   {
      private int x;
      public int y;

      private void Foo()
      {
         //...
      }
      public void Bar()
      {
         //...
      }

   }
}
```

2. Sınıfın kendi içinden sınıfın her bölümüne erişilebilir.

Nesen Yönelimli Programlama Tekniği Nedir? Nesne yönelimli programlama tekniği kabaca sınıflar kullanılarak program yazma tekniğidir. Temel hedefi algısal kolaylık sağlamaktır. Nesne

yönelimli programlama tekniğini bir takım anahtar kavramlarım toplamı olarak düşünebiliriz. Nesne yönelimli programlama tekniğinin en önemli anahtar kavramlarından biri **kapsülleme (Encapsulation)** kavramıdır. Kapsülleme sınıfın dışarıyı ilgilendirmeyen elemanlarının private bölüme yerleştirilerrek gizlenmesi ve yalnızca dışırayı ilgilendiren elemanlarının public bölüme yerleştirilmesidir.
Bir sınıfı için iki bakış açısı söz konusudur.

1. Sınıfı kullanan kişinin bakış açısı
2. Sınıfı yazan kişinin bakış açısı

Sınıfı kullanan kişi için sınıf public bölümünden ibarettir. Bu kişiler public bölümüne odaklanmalıdır. Günlük yaşamda da biz nesneleri daha çok kullanan kişi bakış açısıyla algılamaktayız. Örneğin biz bir aygıt yapacaksak onu oluşturan her parçanın her detayını bilmek zorunda değiliz. İşte yazılımlar içinde aynı durum söz konusudur. Şüphesiz sınıfı tasarlayan kişi sınıfın her bölümünü bilmek zorundadır. Sınıfı tasarlayan kişi dışarıyı ilgilendirmeyen elemanları private bölüme yerleştirerek kafa karışıklığını engellemeye çalışır. Bu çok önemli bir prensiptir. Örneğin Sample isimli bir sınıf yazacak olalım sınıfın DoSomethingImportant isimli faydalı bir fonksiyonu olsun bu fonksiyonda işini yaparken çeşitli bazı fonksiyonları çağırıyor olsun bu fonksiyonlar. Foo, Bar, Zar fonksiyonları diyelim. Sınıfı kullanacak kişinin dışarıdan Foo, Bar, Zar fonksiyonlarını çağırmasının bir anlamı yoktur. İşte kapsülleme tekniğini uygulayacaksak biz foo, Bar, Zar fonksiyonlarını private bölüme yerleştirmeliyiz. DoSomethingImportant fonksiyonunu public bölüme yerleştirmeliyiz.

Kullandığımız bir x sınıfının bir kaç public elemanı var diye bu sınıfı küçük bir sınıf oladuğunu düşünmemeliyiz. Sınıfta çok sayıda private bölümde gizlenmiş olabilir.

Veri Elemanlarının Gizlenmesi: Nesne yönelimli programlama tekniğinin bir önemli kavramı da veri elemanlarının **gizlenmesidir (data hiding).** Sınıfın veri elemanları iç işleyişe ilişkindir. Bu nedenle veri elemanlarının private bölümünde gizlenmesi iyi bir tekniktir.
Sınıfın veri elemanlarını privatge bölüme yerleştirdikten sonra onlara dışarıdan erişemeyiz. Onlara erişebilmek için public get ve set fonksiyonları yazmak gerekebilir. Örneğin veri elemanının ismi A olmak üzere, getA fonksiyonu bunun değerini verebilir, setA fonksiyonu buna değer atayabilir.
Örnek:

```
class Sample
{
        private int a;
        public int getA()
        {
                return a;
        }
        public void setA(int x)
        {
                a=x;
        }
        //...
}
```

İşte C# da private veri elemanına erişmek için gereken get ve set fonksiyonları property denilen yöntemle kolay bir biçimde yazılabilimektedir. Property java ve C++ da yoktur.
Veri Elemanlarının Private Bölüme Yerleştirilmesinin Anlamı: Sınıfın veri elemanları iç işleyişe ilişkindir ve tasarımcı tarafından değiştirilmeye aday elemanlarıdır. Deneyimler veri elemanlarının gerek isimlerinin gerekse genel yapısının çeşitli nedenlerden dolayı değiştirildiğini göstermektedir. İşte biz sınıfın veri elemalarını public bölüme yerleştirirsek sınıfı kullanan kodlar onlara doğrudan erişebilir. Sınıfın genel yapısı da değiştirildiğinde de onu kullanan tüm kodları değiştirmek zorunda

kalırız. Halbuki veri elemanların private bölüme yerleştirirsek bunları değiştirdiğimizde sınıfı kullanan doğrudan etkilenmez. **Örneğin:**

```
class Date
{
        public int day, month, year;

        public Date (int d, int m, int y)
        {
                day = d;
                month = m;
                year = y;
        }
        //...
}
```

Burada sınıfın veri elemanları kötü bir teknik uygulanarak public bölüme yerleştirilmiştir. Artık bu elemanlara herkes erişebilir. Şimdi sınıfı kullanan aşağıdaki gibi bir kod olduğunu varsayalım.

```
Date d= new Date(10,12,2003)
if(d.math==12)
        d.day=1;
```

Normal olarak bir sınıfı kullanan kodlar sınıfın kendi kodlarına göre çok daha fazladır. Şimdi biz Date sınıfında tarihi üç int elemanla tuttuğumuzda pişman olmuş olalım ve tek bir string de tutmak isteyelim;

```
class Date
{
        public string date;

        public Date (int d, int m, int y)
        {
                date=string.Format("{0:D2}/{1:D3}/{2:D4}", d, m, y);
        }
        //...
}
```

işte şimdi sınıfı kullanan tüm kodlar geçersiz hale gelmiştir. Herkesin bu kodları düzeltmesi gerekir. Uygulanması gereken doğru teknik veri elemanlarının private bölüme yerleştirilmesi ve onlara get/set fonksiyonlarıyla erişilmesidir.

```
class Date
{
        private int day, month, year;

        public Date (int d, int m, int y)
        {
                day = d;
                month = m;
                year = y;
        }
        public int GetDay()
        {
```

```
            return day;
        }
        public void SetDay(int d)
        {
            day=d;
        }
        public int GetMonth()
        {
            return month;
        }
        public void SetMonth(int m)
        {
            month=m;
        }
        public int GetYear()
        {
            return year;
        }
        public void SetYear(int y)
        {
            year=y;
        }
}
```

İşte veri elemanları private bölüme yerleştirildiğinde artık dışarıdan bunları kullanacak kişiler mecburen bu elemanlara get ve set fonksiyonlarıyla erişeceklerdir.

```
Date d= new Date(10,12,2003)
if(d.getMonth==12)
    d.SetDay=1; artık bu şekilde kod yazılacaktır.
```

Artık sınıfın veri yapısı değiştirilirse onları kullanan kodların değiştirilmesine gerek kalmaz. Tek yapılacak şey sınıfın public fonksiyonlarının içini parametrik yapıları ve isimleri aynı kalmak koşuluyla yeni durum için yeniden yazmaktır.

Örneğin:

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
        //...
      }

     class Date
     {
         private string date;

         public Date(int d, int m, int y)
         {
            date = string.Format("{0:D2}/{1:D2}/{2:D4}", d, m, y);
         }
         public int GetDay()
         {
            return int.Parse(date.Substring(0, 2));
```

```
        }
        public int GetMonth()
        {
            return int.Parse(date.Substring(3, 2));
        }
        public int GetYear()
        {
            return int.Parse(date.Substring(5, 4));
        }
        public void SetDay(int d)
        {
            string s = string.Format("{0:D2}", d);
            date = date.Remove(0, 2);
            date = date.Insert(0, s);
        }
        public void SetMonth(int m)
        {
            string s = string.Format("{0:D3}", m);
            date = date.Remove(3, 2);
            date = date.Insert(0, s);
        }
        public void SetYear(int y)
        {
            string s = string.Format("{0:D4}", y);
            date = date.Remove(5, 4);
            date = date.Insert(5, s);
        }
        //...
    }
  }
}
```

Veri elemanlarının private bölüme yerleştirilmesinin bir diğer gerekçesi de sınıfın birbiriyle ilişkili veri elemanlarının bulunması durumunda bunlar arasındaki ilişkinin sınıfı kullanan kişiler tarafından bilinmesi zorunluluğunu ortadan kaldırmaktır.

Örneğin:

```
class Circle
{
        private double radius;
        private double centerx, centery;
        private double area;
public Circle(double x, double double r)
{
        centerx=x;
        centery=y;
        centerr=r;

        area=3,14*r*r;
}
//...
```

}

Burada programcı bir daire üzerinde işlemler yapmak için bir sınıf yazmak istemiştir. Dairenin alanı çok fazla kullanıldığı için her defasında pi r kare yapmamak için programcı alan bilgisini area veri elemanında tutmak istemiş olabilir. Ne zaman radius elemanına yeni bir değer atanacak olsa area veri elemanınında değiştirilmesi gerekir.
İşte veri elemanları public bölümde tutulacak olsa radius set edildiğinde area'nın set edilmesi programcının sorumluluğunda olacaktır. Halbuki veri elemanları private bölüme yerleştirildiğinde tüm bu ilişkiler gizlice sınıfın içinde yapılabilecektir.

Bazen bir veri elemanına bir değer yerleştirildiğinde başka bir takım işlemlerin yapılması gerekir. Örneğin: Pencere işlemlerini temsil eden Control isimli bir sınıf olsun. Pencerenin o andaki zemin rengi backcolor isimli bir veri elemanında tutulacak olsun. Bu veri elemanına değer atamakla bencerenin zemin rengi değişmez. Pencerenin zemin rengini değiştirmek için ayrıca bir takım işlemler de yapmak gerekir. İşte veri elemanlarını public bölüme yerleştirirsek bütün bu işlemleri kullanıcının yapması gerekir. Halbuki bu işlemler set fonksiyonu içinde gizlice yapılabilmektedir. Veri elemanlarının private bölüme yerleştirilmesi iyi bir tekniktir. Fakat bazen programcı veri elemanlarını gizlemek istemeyebilir. Örneğin sınıfıın veri eleman yapısından programcı çok emin ise ve get/set fonksiyonlarının çağrılması kritik uygulamalarda istenmiyorsa veri elemanları doğrudan public bölüme yerleştirilebilir.

Sınıfın Property Elemanları: Property ler sınıfın private veri elemanlarını get ve set yapmak için kullanılan özel fonksiyonlardır. Java da ve C++ da property kavramı yoktur. Property ler kodun okunabilirliğini ve algılanabilirliğini arttırmaktadır.

Property bildiriminin genel bildirimi şöyledir:
[erişim belirleyicisi] <tür> <property ismi>
```
{
        get
        {
                //...
        }
        set
        {
                //...
        }
}
```

Bir property get ve set bölümlerinden oluşur. Propertyler genellikle public olduğu için pascal tarzıyla isimlendirilir. Yani property ismi büyük harfle başlar. Noktalı virgül yok. Get ve set fonksiyonlarının sırası farklı olabilir. Property nin get ve set bölümleri adeta bir fonksiyon gibidir.

Property elemanlar veri elemanı gibi kullanılan fonksiyonlardır. Bir property eleman sanki bir veri elemanıymış gibi kullanılır.
Örneğin:
```
Sample s = new Sample();
int result;

result = s.A + 10;
s.A =10;
```

Property eleman ifade içinde değer alma ya da değer yerleştirme biçiminde kullanılır. Yukarıdaki

örnekte A property elemanı değer alma amacıyla kullanılmıştır. (*result = s.A + 10;*)

s.A=10; Burada property elemanı değer yerleştirme amacıyla kullanılmıştır.
Özel bir durum olarak property eleman ++ ya da – operatörünün operandı biçiminde ise hem değer alma hem de değer yerleştirme amacıyla kullanılır.

Örneğin: *++s.A;*

Property nin get bölümü parametresi olmayan geri dönüş değeri property türünden olan bir fonksiyon gibidir. Bu bölümde return işlemi uygulamak gerekir. Tipik olarak private veri elemanıyla geri dönülür.

Property nin set bölümü ise geri dönüş değeri olmayan parametresi property türünden olan bir fonksiyon türüdür. Set bölümünde value eleman sözcüğü bu set bölümünün parametresi anlamına gelir. Programcı tipik olarak set bölümünde value değerini private veri elemanına yerleştirir.
Örneğin:

```
class Sample
{
        private int a;

        public int A
        {
                get
                {
                        //...
                        return a;
                }

                set
                {
                        a = value;
                        //...

                }
        }
        //...
}
```

Property eleman değer alma amacıyla kullanılmışsa otomatik olarak property nin get bölümü çağrılır ve geri dönüş değerinden elde edilen değer işleme sokulur. Yani
result= s.A + 10; işleminin java karşılığı şöyledir.
result = s.GetA() + 10;
Bir property eleman değer yerleştirme amacıyla kullanılmışsa bu durumda property elemanın set bölümü çalıştırılır. Set bölümü içindeki value property'e atanan değeri temsil etmektedir. Yani:
s.A = 10; işleminin java daki eş değeri
s.SetA(10); biçimindedir.

Örnek:

```
namespace CSD
{
```

```
class App
{
    public static void Main()
    {
        Sample s = new Sample();

        s.A = 10;
        System.Console.WriteLine(s.A);
    }

}
class Sample
{
    private int a;

    public int A
    {
        get
        {
            //...
            return a;
        }

        set
        {
            a = value;
            //...
        }
    }
    //...
}
}
```

O halde C# da veri elemanları private bölüme yerleştirilip onlara propertilerle erişilmelidir. Örneğin dizilerin uzunluğunu almakta kullandığımız Length elemanı gerçek bir veri elemanı değil bir property elemanıdır.

Bir property nin yalnızca get bölümü olabilir bu tür propertylere **read only propertyler** denir. Read Only propertylere değer yerleştirilmeye çalışılırsa error oluşur. Örneğin **dizilerin Length property si read only property dir.**

Bir property nin yalnızca set bölümü de bulunabilir. Bu tür property lere "Write Only" property ler denir. Bu tür property ler değer alma amaçlı kullanılamazlar ve çok seyrek kullanılırlar.

Nihayet bir property nin her iki bölümü de varsa bunlara read/write property ler denir. Bir property ++ ya da – operatörü ile kullanılırsa property nin önce get sonra set bölümü çalıştırılır.
Örnek:

```
using System;

namespace CSD
```

```
{
   class App
   {
      public static void Main()
      {
         Person p = new Person();
         p.Name = "Kaan Aslan";
         Console.WriteLine(p.Name);
      }

   }
   class Person
   {
      private string name;

      public string Name
      {
         get
         {
            return name;
         }

         set
         {
            name = value;

         }
      }
      //...
   }
}
```

Şüphesiz property ler basit bir get/set işlemi yapmak zorunda değildir. Bir propertye değer atarken set bölümü sayfalarca uzunlukta bir kodu çalıştırıyor olabilir. Programcı basit bir değer atama sintaksında karmaşık bir işlemi hallediyor olabilir. Bu durum uygulama programcılığını çok kolaylaştırmaktadır.

Örneğin .net te bir GUI programın nerdeyse yarısı propertyler set edilerek oluşturulabilmektedir. Propertyler set ve get edilirken aslında bir fonksiyon çağırma işlemi gerçekleştirilmektedir.

Bir property elemanın sintaks bakımından bir private veri eleman ile ilişkilendirilmesi de söz konusu değildir. Örneğin bir property elemana değer atandığında hiçbir veri elemanı set edilmeyip bir takım işlemler yapılıyor olabilir. Ya da birden fazla veri elemanı set ediliyor olabilir.

Sınıfın Statik Propertyleri: Sınıfın statik veri elemanlarına da private bölümde gizlenerek onlara get ve set fonksiyonlarıyla erişilebilir. Sınıfın statik veri elemanlarına erişmek için statik propertler kullanılır. Statik propertyler sınıf ismiyle kullanılır.
Örneğin:

```
class sample
   {
      private static int a;
```

```
    public static int val
    {
        get
        {
            return val;
        }

        set
        {
            val = value;

        }
    }
    //...
}
```

```
sample.val = 10;
Console.WriteLine(sample.val);
```

Property Elemanların GUI Uygulamalarındaki Önemi: Propertiler programlamayı oldukça kolaylaştırmaktadır. Örneğin pencereli bir ekranın tasarımının neredeyse yarısı yalnızca sınıfın bazı propertylerinin set edilebilmesiyle yapılabilmektedir. Property kavramı aynı zamanda görsel programlamayı da kolaylaştırır. Bir property ekranıyla programcı porpertyleri görsel olarak set edebilir. IDE de bunun için gerçek kodları üretebilir.

this Referansı: Aslında bilgisayar sisteminde aşağı seviye çalışma dikkate alındığında statik olmayan fonksiyon diye bir şey yoktur. Statik olmayan fonksiyonların sınıfın statik olmayan veri elemanlarına erişmesi de aslında doğrudan yapılmamaktadır. Statik olmayan bir fonksiyonun statik karşılığı her zaman yazılabilir.

```
class Sample
{
        int a, b;
        public void Func()
        {
                a = 10;
                b= 20;
        }
        //...
}
//...
Sample s = new Sample();
s.Func;
```

Yukarıda yapılan işlemin eşdeğeri de şöyle de gerçekleştirilebilirdi.

```
class Sample
{
        int a, b;
        public static void Func(Sample s)
        {
                s.a = 10;
```

```
            s.b= 20;
        }
        //...
}
//...
Sample s = new Sample();
Sample.Func(s);
```

Aslında biz statik olmayan bir fonksiyonu yazdığımızda derleyici o fonksiyona ek olarak bir referans parametresi eklemekte ve onu statik bir fonksiyon gibi yazmaktadır. Statik olmayan fonksiyonu biz referansla çağırdığımız zaman aslında derleyici çağırma işleminde kullanılan referansı fonksiyona parametre olarak geçirmektedir. Statik olmayan veri elemanlarına statik olmayan fonksiyon içinde doğrudan eriştiğimizde aslında derleyici bu veri elemanlarına doğrudan değil gizlice geçirdiği referans parametresini kullanarak erişmektedir.
Örneğin statik olmayan bir fonksiyon için biz n tane parametre değişkeni yazmışsak aslında fonksiyonun n +1 tane parametre değişkeni vardır.

Derleyicinin statik olamayan fonksiyonlara gizlice geçirdiği referans parametresini biz fonksiyonun içinde açıkça "**this" anahtar sözcüğüyle** kullanabiliriz. This Anahtar sözcüğü fonksiyonun çağrılmasında kullanılan referansın aktarıldığı gizli parametre değişkenini temsil etmektedir. Statik olmayan bir fonksiyon içinde sınıfın a isimli bir veri elemanına doğrudan a biçiminde erişmekle this.a biçiminde erişmek arasında hiçbir fark yoktur. Zaten biz yalnızca a biçiminde erişsek bile derleyici bu veri elemanına aslında this.a biçiminde erişmektedir.

this anahtar sözcüğü hangi sınıf içinde kullanılıyorsa o sınıf türünden bir referans belirtir. this refernası read-only dir içine atama yapamayız.

Sınıfın statik olmayan bir fonksiyonunun sınıfın başka bir statik olmayan fonksiyonunu doğrudan çağırdığını düşünelim. Aslında bu durumda derleyic çağrılan fonksiyonu çağıran fonksiyona geçirilen this referansıyla çağırmaktadır. Bu durumda statik olamayan bar isimli bir fonksiyonu Bar(); biçiminde çağırmakla this.Bar() biçiminde çağırmak arasında hiçbir farklılık yoktur.

Şüphesiz statik bir fonksiyon içinde this anahtar sözcüğü kullanılamaz. Çünkü statik fonksiyonlara böyle bir referans geçirilmemektedir.

this Anahtar sözcüğü neden kullanılır? Bazen bu anahtar sözcüğü kullanmak gerekebilir. Örneğin bir yerel değişken ya da parametre değişkeni ile aynı isimli sınıfın bir veri elemanı varsa sınıfın veri elemanına erişmek için this referansı ile niteliklendirme yapmak gerekir.
Örneğin:

```
class Date
{
        private int day, month, year;

        public Date(int day, int month, int year)
        {
                this.day = day;
                this.month = month;
                this.year = year;
        }
        //...
}
```

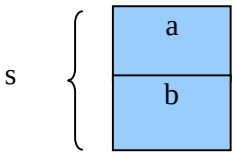
Veri Elemanlarının Vurgulanması: Bir fonksiyonun içinde a gibi bir değişkenin kullanıldığını görmüş olalım. Bu a, fonksiyonun yerel değişkeni ya da sınıfın bir veri elemanı olabilir. Kodu anlamlandırabilmemiz için bunu anlamak durumunda kalabiliriz. İşte bazı programcılar hiçbir çakışma olmasa bile sınıfın tüm veri elemanlarına this referansı ile erişirler. Böylece kodu inceleyen kişi bunun bir yerel değişken olmadığını sınıfın bir veri elemanı olduğunu hemen anlar. Bazı programcılar bu yöntemin yerine sınıfın veri elemanlarını m_(member), d_(data member) gibi ön eklerle başlayarak isimlendirirler. Böylece kodu inceleyen kişi örneğin m_a gibi bir isim gördüğünde bunun sınıfın bir veri elemanı olduğunu anlar. Microsoft örneklerinde this'i kullanmaktadır. Fakat biz kursumuzda m öneki yöntemini kullanacağız.

YAPILAR: Bir yapı tıpkı sınıf gibi bildirilir. Fakat class yerine struct anahtar sözcüğü kullanılır. Örneğin:

```
struct Sample
{
        //...
}
```

Bir yapı proctected ve protected internal elemanlara sahip olamaz. Nedeni türetilememesidir.

Yapılar kategori olarak değer türlerine ilişkindir. Yani bir yapı türünden değişken bildirdiğimizde bu bir referans değildir. Bileşik bir nesnedir.




```
struct Sample
{
        public int a;
        public int b;
        //...
}
//...
Sample s;
```

Bir yapı değişkeni yerel olarak bildirilmişse otomatik yaratılır ve yok edilir.
Bir yapı değişkeninin her elemanına değer atanmadan o yapı değişkeni bütünsel olarak kullanılamaz. Fakat değer atanan parçası parçalı olarak kullanılır.

```
Sample x, y ;
x.a = 10;
y = x ; //error

int z;
z = x.a; // geçerli
```

Aynı türden iki yapı değişkeni birbirine atanabilir. Bu durumda yapının karşılıklı elemanları birbirine atanmaktadır.

```
using System;
namespace CSD
{
   class App
   {
```

```
    public static void Main()
    {
        sample x;
        sample y;

        x.m_a = 10;
        x.m_b = 20;

        y = x;

        y.m_a = 30;
        x.m_b = 40;

        Console.WriteLine("x.m_a = {0},x.m_b = {1}", x.m_a, x.m_b);
    }

}
struct sample
{
    public int m_a;
    public int m_b;
}
}
```

Yapılar da sınıflar gibi statik elemanlara sahip olabilir. Yapıların veri elemanlarını da private bölümde gizleyerek public propertyler le erişmek iyi bir tekniktir.

Aslında int, long, double gibi temel türler de birer yapıdır.
Örneğin int anahtar sözcüğü aslında system.int32, long anahtar sözcüğü system.int64 anlamına gelir.

Yapılar için de new operatörü ile tahsisat yapılabilir. Fakat bu durumda new operatörü tahsisatı heap de yapmamaktadır.

New operatörünün operandının yapı olmasıyla veya sınıf olması arasında anlam farkı vardır.

New operatörü ile bir yapı türünden tahsisat yapıldığında new operatörü önce stack te ilgili yapı türünden geçici bir nesne yaratır. Sonra bu geçici nesne için new sintaksında belirtilen başlangıç fonksiyonunu çağırır. Böylece new işleminden aslında stack te yaratılmış olan geçici nesne elde edilmiş olur.
Örneğin:
Sample s;
s = new Sample();
Burada s = new Sample(); işlemiyle aslında aynı türden iki yapı nesnesi birbirine atanmaktadır. Eğer sample bir sınıf olsaydı new operatöründen elde edilen ürün bir adres olacaktı. New operatörü ile yaratılan geçici nesne. Struct konusu java da yoktur.

New operaötürünün yapılar için kullanılmasının en önemli faydası başlangıç fonksiyonundan faydalanmaktır.

```
struct sample
 {
```

```
        private int m_a;
        private int m_b;

        public int A
        {
            get {return m_a;}
            set {m_a = value;}
        }
        public int B
        {
            get {return m_b;}
            set {m_b = value;}
       }
        //...

    }
    //...
    Sample s;
    s = new Sample (10, 20);
    Console.WriteLine(s.A);
    Console.WriteLine(s.B);
}

Örnek:
using System;
namespace CSD
{
    class App
    {
        public static void Main()
        {
          //...
            sample s;
            s = new sample (10, 20);
            Console.WriteLine(s.A);
            Console.WriteLine(s.B);
        }

    }
    struct sample
    {
        private int m_a;
        private int m_b;

        public sample(int a, int b)
        {
            m_a = a;
            m_b = b;
        }

        public int A
        {
```

```
        get {return m_a;}
        set {m_a = value;}
      }
      public int B
      {
        get {return m_b;}
        set {m_b = value;}
      }
      //...

   }
}
```

Yapılar için default başlangıç programını programcı yazamaz. Default başlangıç fonksiyonu her zaman derleyici tarafından yazılmaktadır. (Sınıflarda programcı hiçbir başlangıç fonksiyonu yazmamışsa default başlangıç programı derleyici tarafından yazılır. Halbuki yapılarda her zaman default başlangıç fonksiyonu derleyici tarafından yazılır.) **Derleyici yapı için yazdığı default başlangıç fonksiyonun da yapının tüm elemanlarını sıfırlar.**

Yapılar için başlangıç fonksiyonu programcı tarafından yazılacaksa başlangıç fonksiyonunun içinde yapının tüm elemanlarına değer atamak zorunludur. Yani yapının bir elemanına bile değer atamazsak derleme sırasında error oluşur. Aslında tasarımcı yaratılmış bir nesnenin tüm elemanlarına sıfırda olsa bir değer atanmasını istemiştir. Bu işlem sınıfı nesnesi new ile tahsis edilirken henüz başlangıç fonksiyonu çağrılmadan new operatörü tarafından yapılmaktadır. Halbuki new ile bir yapı tahsis edildiğinde new operatörü yapı için stack te yer tahsis etmektedir. Fakat orayı new operatörü sıfırlamamaktadır. İşte bu alana yapının başlangıç fonksiyonunun değer ataması uygun görülmüştür. Int, Long gibi türlerde bir yapı olduğuna göre onların da başlangıç fonksiyonu derleyici tarafından yazılmıştır. O halde : *int a = new int(); ile int a = 0;* aynı anlamdadır.

Sınıfın Yapı Türünden Veri Elemanları: İster bir referans olsun isterse bir yapı değişkeni olsun fonksiyonların içinde bildirilmiş olan her değişken stack te yer almaktadır. Stack teki değişkenlerin yaratılmaları ve yok edilmeleri akış sırasında otomatik yapılmaktadır.
Bir sınıfın bir veri elemanının bir yapı olduğunu düşünelim. Şimdi sınıf nesnesini new operatörü ile tahsis ettiğimizde yapı değişkeni heap te sınıf nesnesinin içinde yaratılacaktır. Yok edilmeleri de çöp toplayıcı sistem tarafından yapılacaktır.

Yapının Sınıf Türünden Veri Elemanları: Yapının veri elemanı sınıf türünden referans olabilir. Bu durumda akış yapı değişkeninin bildirildiği bloktan çıkınca yapı değişkeni bellekten silinecek ve böylece yapının veri elemanının gördüğü nesne seçilebilir duruma gelecektir. Bu durumda herhangi bir sorun oluşmaz.

enum Türü ve Sabitleri: Bazen belli bir durum için az sayıda geçerli olasılık vardır. Bunların string türüyle ifade edilmesi okunabilirliği arttırsa da performansı olumsuz etkiler. Örneğin func() isimli bir fonksiyonun bizden haftanın bir gününü istediğini düşünelim. Func fonksiyonunun bu parametresi hangi türdendir? Gün string parametesi ile bizden istense fonksiyonu aşağıdaki gibi çağırırız.
Func("Pazartesi"); Bu yöntem de bizden yazıyı alan func fonksiyonu hangi günün girildiğini anlamak için yazı karşılaştırması yapacaktır. Yazı karşılaştırmaları sayı karşılaştırmalarına göre çok yavaştır. Üstelik bu yöntemle yazı fonksiyonu kullanacak kişi tarafından yanlış girilmiş olabilir. Yani hataya açıktır.
Akla gelecek diğer bir yöntem parametrenin int türden olmasıdır. Bu konuda baştan bir anlaşma sağlanır. Örneğin 7="pazar", 1="pazartesi" vb gibi. Bu durumda biz çağırmayı şöyle yapabiliriz.

Func(1); Bu yöntemde okunabilirlik oldukça zayıftır. Koda bakan birisi fonksiyona hangi günün girildiğini hemen anlayamaz. Bu yöntem de yanlış girişlere olanak sağlamaktadır.

Yukarıda açıklanan durumlar için bizim şu olanakları sağlayan bir mekanizmaya ihtiyacımız vardır.
-Yazısal bir tevsir olmalı
-işlemler aslında sayısal düzeyde yapılmalı
-yanlış girişe izin verilmemeli
işte enum türü bu nedenlerle kullanılır.
enum bildiriminin genel biçimi şöyledir:
enum <isim>
{
<enum sabit listesi>
}
enum sabit listesi virgüller le ayrılmış bir grup isimden oluşur.
Örneğin:
enum Gunler
{
Pazar, Pazartesi, Salı, Çarşamba, Perşembe, Cuma, Cumartesi
}
enum Colors
{
Red, Green, Blue
}

enum bildirimi içinde belirtilen sabitlere enum sabitleri denir. İlk enum sabiti 0 "Sıfır" değerindedir. Sonraki her sabit öncekinden bir fazladır.

Bir enum sabitine doğrudan değil enum ismi kullanılarak erişilir.
Örneğin: *Gunler.Pazartesi Colors.Red*
Bu durumda Gunler.Pazartesi aslında bir sayıdır ve 1 "Bir" değerindedir. Colors.Red de bir sayıdır sıfır değerindedir.

Enum aynı zamanda bir tür belirtmektedir. Biz enum türünden değişkenler bildirebiliriz.
Gunler g;
Color c;
enum sabitleri de ilgili enum türündendir. Burada g Gunler türünden c Colors türündendir. Enum türünden bir değişkene doğrudan int bir değer atayamayız.
Örneğin:
Colors c;
c = 1; //error

Bir enum değişkenine aynı türden bir enum değeri atanır.
Örneğin:
Colors c;
c = Colors.Red; Burada c ve Colors.Red Colors isimli enum türündendir.
Bu durumda bizden bir gün isteyen Func() fonksiyonunun parametre değişkeninin Gunler isimli bir enum türünden olması anlamlıdır. Biz de fonksiyonu aynı türden bir enum sabitiyle çağırırız.

Console sınıfının Write ve WriteLine fonksiyonları ile bir enum türü yazdırılmak istenirse fonksiyon enum değerine ilşkin sayıyı değil sabitin yazısını basar.
Örneğin:
using System;

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         Func(Gunler.Sali);
      }
      public static void Func(Gunler g)
      {
         Console.WriteLine(g);
      }

   }

   enum Gunler
   {
      Pazar, Pazartesi, Sali, Carsamba, Persembe, Cuma, Cumartesi
   }
}
```

Bir enum sabitine eşittir = ile bir değer karşılık getirilebilir. Bu durumda diğerleri o değerleri izler.
Örneğin:

```
enum Test
{
        XX = 10, YY, ZZ = -1, KK, MM
}
```

Burada XX = 10 YY=11 ZZ=-1 KK=0 ve MM=1 dir. Farklı enum sabitleri aynı değerde olabilir.
Örneğin:

```
enum Test
{
        XX , YY, ZZ = -1, KK, MM
}
```

Burada XX ile KK YY ile MM aynı değerdedir. Enum sabitine eşittir ile verilen değerlerin sabit ifadesi olması gerekir. Enum sabitleri de sabit ifadesi kabul edilmektedir. Örneğin biz onları Case ifadelerinde kullanabiliriz.
Örneğin:

```
enum Colors
{
        Red, Green, Blue
}
//...
```

Örneğin:

```
 switch (c)
 {
       case Cloros.Blue:
            //...
       case Cloros.Green:
            //...
       case Cloros.Red:
```

```
        //...
}
```

Her enum türünün ilişkin olduğu bir tam sayı türü vardır. Aslında enum değerleri arka planda o tamsayı türündenmiş gibi işleme sokulmaktadır. Enum türünün ilişkin olduğu tamsayı türü " :" iki nokta üst üste sintaksıyla belirtilmektedir.

```
enum Test : long
{
        //...
}
//...
```

Eğer enum türünün ilişkin olduğu tamsayı türü belirtilmezse default olarak int kabul edilir.
Hiçbir enum sabiti enum türünün ilişkin olduğu tamsayı türünün sınırları dışında değer alamaz.
Örneğin:

```
enum Test : byte
{
        XX = 254, YY, ZZ //Derleme zamanı hatası Error ZZ=256 olur sınır taşması
}
//...
```

Burada ZZ değeri 256 olacaktır fakat enum türünün ilişkin olduğu tamsayı türü buna izin vermeyecektir. Görüldüğü gibi aslında bir enum değeri kullandığımızda biz enum türünün ilişkin olduğu tamsayı türünden bir değer kullanıyor gibi oluyoruz diyoruz.
Enum türünden bir değişken için kaç byte yer ayrılır? Enum türünün ilşkin olduğu tamsayı türü kaç byte'sa o kadar yer ayrılır. Enum türleri katagori olarak değer türlerine ilişkindir.
Bir tamsayı türü tür dönüştürme operatörü ile enum türüne dönüştürülebilir. Bu dönüştürmede sanki tamsayı türü enum türünün ilişkin olduğu tamsayı türüne dönüştürülüyormuş gibi kurallar uygulanır. Örneğin:

```
enum Meyva
{
        Elma = 10, Armut = 20, Kayisi = 30, Ayva = 40, Visne = 50
}
//...
```

Bir tamsayı türü bir enum türünden değişkene doğrudan atanamaz. Çünkü tamsayı türlerinden enum türlerine doğrudan dönüştürme yoktur. Fakat tamsayı türlerinden enum türlerine tür dönüştürme operatörüyle dönüşüm yapılabilir.

```
Enum Meyva
{
        Elma, Armut, Çilek
}
//...
Meyva m;
m=2; // error
m = (Meyva) 2; Geçerli
```

Dönüştürülmek istenen tamsayı türü T, enum türü E, enum türünün ilişkin olduğu tamsayı türü U olsun. Bu durumda:

```
T t;
E e;
e = (E) t;
```

e = (E)(U)t; biçiminde gerçekleştirilir.

Tamsayı türü enum a dönüştürülürken ilgili enum türünün o tamsayı değerine ilişkin bir enum sabiti bulunmak zorunda değildir. Yani enum türünden bir değişken ilişkin olduğu tamsayı türünden herhangi bir değeri tutabilir.

Bir enum türü doğrudan bir tamsayı türüne atanamaz fakat tür dönüştürme operatörüyle bu işlem yapılabilimektedir.
Örneğin:
Meyva m = Meyva.Ayva;
int val;
val = (int) m;

Aynı türden iki enum karşılaştırma operatörleriyle karşılaştırılabilir. Bu enum değişkenlerinin değerleri karşılaştırılır.
Örnek:

```
using System;
namespace CSD
{
   class App
   {
      public static void Main()
      {
         Meyva m = Meyva.Ayva;
         Meyva t = Meyva.Elma;

         if (m > t)
            Console.WriteLine("Evet");
         else
             Console.WriteLine("Hayir");
      }
   }

   enum Meyva
   {
      Elma=0, Armut, Kayisi, Visne, Ayva
   }
   //...
}
```

Bir enum türü ile bir tamsayı toplanabilir. Sonuç aynı türden bir enum olur.
e E enum türünden t T ile temsil edilen tamsayı türünden, Enum türünün default değeri U olsun. E + t ve e-t ifadeleri geçerlidir. Bunlar E türündendir. Burada koşul olarak T türünden U türüne doğrudan dönüştürmenin mümkün olması gerekir. Standartlara göre e + t işlemi
 (E) ((u U)e + (U)t)) işlemi biçimindedir.
Örnek :

```
using System;
namespace CSD
{
   class App
   {
      public static void Main()
```

```
        {
            Meyva m = Meyva.Ayva;
            Meyva t = Meyva.Elma;

            t = m + 1;

            Console.WriteLine(t);
        }
    }

    enum Meyva
    {
        Elma=0, Armut, Kayisi, Visne, Ayva
    }
    //...
}
```

Örnek:

```
using System;
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Meyva m = Meyva.Ayva;
            Meyva t = Meyva.Elma;

            t = m + 1L; // Error

            Console.WriteLine(t);
        }
    }

    enum Meyva
    {
        Elma=0, Armut, Kayisi, Visne, Ayva
    }
    //...
}
```

Enum türü ++ ve – operatörleriyle birlikte kullanılabilir.
Örneğin: ++e
e=e+1;
--e;
e=e-1

Aynı türden iki enum çıkartma operatörü ile çıkartılabilir. Bu durumda sonuç ilişkin olunan tamsayı türünden çıkar.

Örneğin:

```
using System;
namespace CSD
{
   class App
   {
      public static void Main()
      {
         Meyva m = Meyva.Ayva;
         Meyva t = Meyva.Elma;
         int result;

         result = m - t;

         Console.WriteLine(result);
      }
   }

   enum Meyva
   {
      Elma=0, Armut, Kayisi, Visne, Ayva
   }
   //...
}
```

e1 ve e2 enum türünden olsun E'nin ilişkin olduğu tamsayı türü U olsun e1-e2 işilemi (U)e1-(U)e2 işlemi ile eşdeğerdir.

Sonucu enum türünden istiyorsak işlemi tür dönüştürme operatörü ile enum türüne dönüştüreceğiz.

Aynı türden iki enum + operatörü ile toplanamaz.

ENUM türünün kullanımına ilişkin örnekler;

.net kütüphanesinde yoğun biçimde enum kütüphanesi kullanılmaktadır. DateTime isimli yapı tarih ve zaman işlemleri yapmak için kullanılmaktadır. Yapının now isimli statik properties'i o anki tarih ve zaman biligisini DateTime biçiminde verebilir.
Örneğin: DateTime dt = DateTime.Now; Burada dt'nin içinde o anki tarih ve zaman bilgisi bulunacaktır.
Date.Time yapısının DayofWeek isimli property elemanı DayofWeek isimli enum türündendir. DayOfWeek isimli enum haftanın günlerini belirten elemanlara sahiptir.
Örneğin:

```
using System;
namespace CSD
{
   class App
   {
      public static void Main()
      {
         DateTime dt = DateTime.Now;
         DayOfWeek dow = dt.DayOfWeek;

         Console.WriteLine(dow);
```

```
        }
    }
}
```

Aslında aynı işlem tek satırda şöyle halledilebilirdi:

```
using System;
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Console.WriteLine(DateTime.Now.DayOfWeek);
        }
    }
}
```

MessageBox diyalog penceresi çıkartan MessageBox sınıfın statik Show fonksiyonları vardır. Bir MessageBox da şu özellikler belirtilebilmektedir.

- Başlık kısmındaki yazılabilimektedir
- Pencere içindeki yazı
- Tuş takımı
- İkon görüntüsü

Tuş takımı için Birkaç seçenekten biri seçilebilir. Benzer biçimde ikon görüntüsü için yine birkaç seçenekten biri kullanılır. MessageBox sınıfının pek çok Show fonksiyonu vardır. Bu Show fonksiyonlarında yukarıdaki bazı öğeler default alınmaktadır.

```
public static MessageBoxResult Show{
        string messageBoxText,
        string caption,
        MessageBoxButton button,
        MessageBoxImage icon
}
```

Visual Studio IDE si 2005 ve sonraki versiyonlarında bir enum ismi yazıldığında ve nokta tuşuna basıldığında inetlisens enum ın bütün elemanlarını göstermektedir. Böylece programcı giriş yaparken bütün seçenekleri görebilmektedir.

Sınıfların Türetilmesi:

Bir dilin nesne yönelimli olabilmesi için üç özelliğe sahip olması gerekir.

1. Dilde sınıf kavramı olmalıdır.
2. Dilde türetme (Inheritence) özelliği olmalıdır.
3. Çok biçimlilik(Polymorphism) özelliği bulunabilmelidir.

Türetme nesne yönelimli programlamının olmazsa olmazlarındandır.
Türetme(inheritence) mevcut bir sınıfa eklemeler yaparak onu genişletmeyi hedefleyen bir tekniktir. Örneğin elimizde bir A sınıfı olsun. Bu sınıfın pek çok fonksiyonu istediğimiz gibi olabilir. Programcı sınıfa bu sınıfta olmayan çeşitli fonksiyonları eklemek isteyebilir. İşte türetme konusu bununla ilgilidir.

Anahtar Notlar: Türetme konusuna ingilizce “inheritence-Kalıtım” denilmektedir. Aslında türetme

(derivation) kalıtım işlemini sağlamak için yapılan işleme denilmektedir. Fakat biz konuya bütünsel olarak türetme diyeceğiz.

Bir sınıfa yeni bir takım fonksiyonlar eklemek için üç yöntem akla gelebilir. Örneğin elimizde bir A sınıfı bulunuyor olsun. Biz bu sınıfı genişleterek bir B sınıfı elde etmek isteyelim.

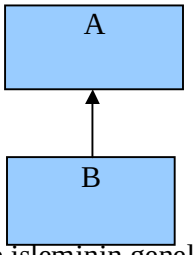
1. Eğer A sınıfının kaynak kodları varsa biz A sınıfından bir kopya çıkartıp ismini B yapabiliriz. Gerekli eklentileri B sınıfı üzerinde yapabiliriz. Bunun dezavantajları şunlardır. Birincisi gereksiz tekrara yol açmasıdır. Hem A sınıfında hem de B sınıfında aynı fonksiyonlardan bulunucaktır. İkinicisi bu yöntemin uygulanması için A'nın kaynak kodlarının bulunuyor olması gerekir.
2. B sınıfı içinde A sınıfı türünden bir referans veri elemanı alırız. Ve B sınıfının başlangıç fonksiyonu içinde A sınıfı türünden nesneyi yaratırız.

```
class A
{
        //...
}

class B
{
        public A m.a;

        public B()
        {
                m.a = new A();
                //...
        }
                //..
}
```

Nesne yönelimli bu yönteme içerme(Composition) ilişkisi denilmektedir. Bu yöntemde kişi B sınıfı türünden bir nesne yaratır. B b = new B(); b.M.a ifadesiyle A sınıfının elemanlarını kullanabilir. b referansıyla da B sınfının elemanları kullanılabilir. Bu yöntem uygun gibi gözükse de genişletme anlamında dezavantajlara sahiptir. Genellikle çok biçimlilik bu yöntemle sağlanamaz.

3. Türetme yöntemi genişletmeye en uygun yöntemdir.

Türetme yönteminde işlevini genişletmek istediğimiz sınıfa **Taban Sınıf (Base Class)**, genişletme ile elde ettiğimiz sınıfa **Türemiş Sınıf(Derived Class)** denilmektedir. Türetme durumu UML sınıf diyagramlarında türemiş sınıftan taban sınıfa çekilen bir okla temsil edilmektedir.



Türetme işleminin genel biçimi şöyledir.

class <türemiş sınıf ismi> : <taban sınıf ismi>

Örneğin:

```
class A
{
        //...
```

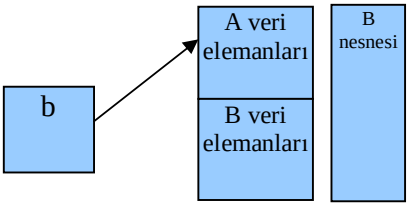
```
}
clas B : A
{
        //...
}
```

Burada B türemiş sınıf ismi A taban sınıf ismidir. Türetme de türemiş sınıf hem taban sınıf gibi kullanılabilir hem de ek bir takım özelliklere sahiptir. Ek özellikler türemiş sınıfa eklenmektedir.

Türemiş sınıf nesnesi hem kendi veri elemanlarını hemde taban sınıf veri elemenlarını içerir. Yani biz türemiş sınıf türünden bir nesneyi new operatörü ile yarattığımızda nesne iki parçadan oluşmaktadır. Taban sınıf ve türemiş sınıf parçaları.

Örneğin:

```
class A
{
        //...
}
class B : A
{
        //...
}
//...
B b = new B();
```



Türetme işleminde türemiş sınıf nesnesi hem türemiş sınıf nesnesi gibi hem de taban sınıf nesnesi gibi kullanılabilir.

Örnek:

```
using System;
namespace CSD
{
   class App
   {
      public static void Main()
      {
        B b = new B();

        b.ValA = 10;
        b.ValB = 20;

        Console.WriteLine(b.ValA);
        Console.WriteLine(b.ValB);
      }
   }
   class A
   {
      private int m_a;

      public int ValA
      {
         get {return m_a;}
         set {m_a = value;}
```

```
    }

  }
  class B : A
  {
    private int m_b;

    public int ValB
    {
      get {return m_b;}
      set {m_b = value;}
    }
  }
}
```

Türemiş Sınıflarda Erişim Kuralları:

1. Türemiş sınıf türünden bir referans yoluyla ya da statik eleman söz konusuysa türemiş sınıf ismiyle dışarıdan türemiş ve taban sınıfın yalnızca public bölümlerine erişilir.
2. Türemiş sınıfın fonksiyonları içinde taban sınıfın public ve protected veri elemanları doğrudan kullanılabilir. Fakat taban sınıfın pirvate bölümüne doğrudan erişilemez.

```
using System;
namespace CSD
{
  class App
  {
    public static void Main()
    {
      B b = new B();

      b.ValA = 10;
      b.ValB = 20;

      Console.WriteLine(b.ValA);
      Console.WriteLine(b.ValB);
    }
  }
  class A
  {
    private int m_a;

    public int ValA
    {
      get {return m_a;}
      set {m_a = value;}
    }
    protected int m_x;

    protected void Foo()
    {
    }
```

```
    }
    class B : A
    {
        private int m_b;

        public int ValB
        {
            get {return m_b;}
            set {m_b = value;}
        }
        public void Bar()
        {
            Foo();
        }

    }
}
```

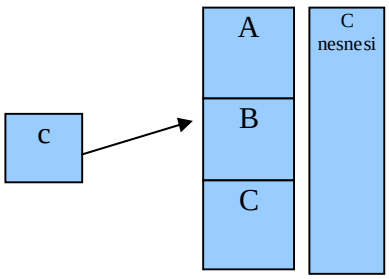
Protected Bölümün Anlamı: Protected bölüm dışarıdan erişilmeyen fakat türemiş sınıf tarafından erişilen bölümdür. Sınıfın en korunmuş bölümü private bölümüdür. Private bölüme yalnızca sınıfın kendisi erişebilir. İkinci korunmuş bölüm protected bölümdür. Protected bölüme türemiş sınıf tarafından erişilebilir. Public bölümde herhangi bir koruma yoktur.

Bir sınıf tasarlarken daha tasarım aşamasında sınıftan türetme yapılıp yapılmayacağını öngörmeliyiz ve türemiş sınıfı yazanların işini kolaylaştırmak için bir takım elemanları protected bölüme yerleştirebiliriz. Dökümantasyon yapılırken sınıfın public bölümünün yanısıra türetme yapacaklar için protected bölümünün de açıklanması gerekir.

Türetilmiş bir sınıftan yeniden bir türetme yapılabilir.

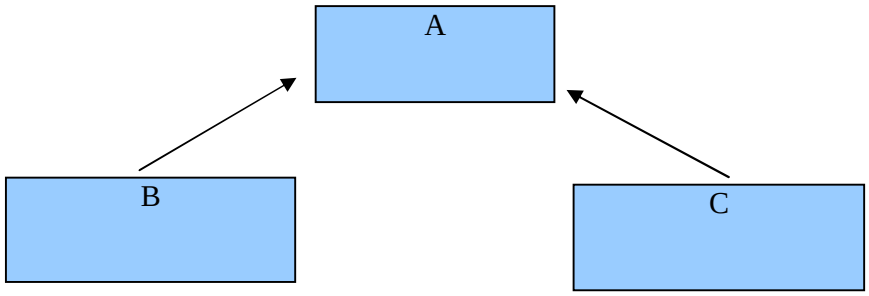
Örneğin:

```
class A;
{
        //...
}
class B : A
{
        //...
}
class C : B
{
        //...
}
C c = new C();
```

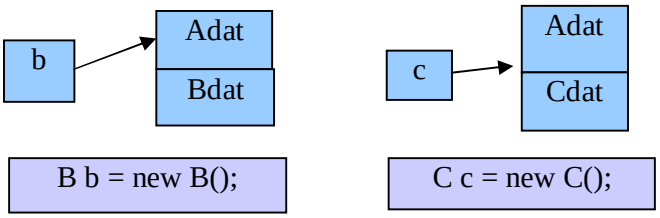


Burada türemiş sınıf referansıyla dışarıdan tüm taban sınıfların public bölümlerine erişebiliriz. Yine türemiş sınıfın içinden tüm taban sınıflarım public ve protected bölümlerine erişebiliriz.
Bir sınıf birden fazla sınıfın taban sınıfı olabilir.

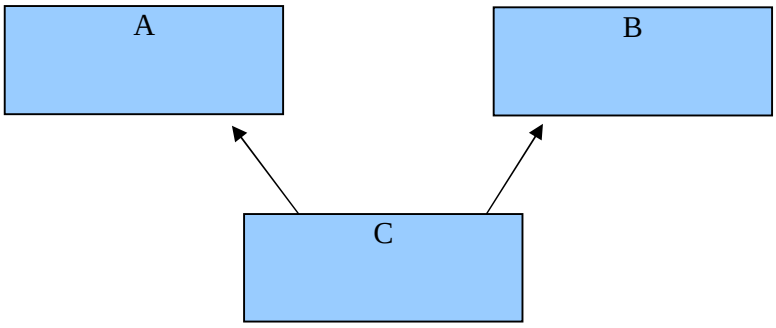
Örneğin:




```
class A;
{
        //...
}
class B : A
{
        //...
}
class C : A
{
        //...
}
```



Burada C türetilmiş sınıfı B türetilmiş sınıfının elemanlarını kullanmaz. Yukarıdaki durumun tersi yani bir türemiş sınıfın bir den fazla taban sınıfa sahip olması durumu ilginç bir durumdur. Bu duruma **çoklu türetme**(Multiple İnheritence) denilmektedir.



Çoklu türetme karışık olduğu gerekçesiyle C# ve Javaya sokulmamıştır. Fakat C++ da çoklu türetme özelliği vardır.

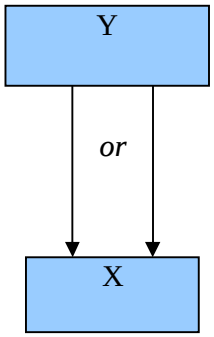
Ne Zaman Türetme Uygulanmalıdır? Nesne yönelimli programlama tekniğinin anahtar kavramlarından biri de **yeniden kullanılabilirliktir.(reusability**) Bir sınıf mümkün olduğunca sıfırdan değil mevcut sınıflardan faydalanılarak tasarlanmalıdır.

Bir X sınıfı yazmak isteyelim. Elimizde mevcut bir Y sınıfı olsun eğer X bir çeşit Y ise burada türetme uygundur. İngilizce bu ilişkiye “is-a” ilişkisi denir.
Örneğin: Tüm taşıtların ortak özellikleri yani her taşıtta bulunan ortak bilgiler bir taşıt sınıfıyla temsil edilmiş olsun biz de bir otomobil sınıfı tasarlamak isteyelim. Otomobil bir çeşit taşıttır. O halde otomobil sınıfı sıfırdan değil taşıt sınıfından türetilerek yazılmalıdır.

Türetmeyle içerme ilişkisi bazen birbirine karışmaktadır. Biz bir x sınıfı tasarlayacak olalım. Elimiz de de bir y sınıfı bulunuyor olsun. Eğer x bir çeşit y değilse fakat x'in parçalarından biri y ise bir içerme ilişkisi söz konusudur. **İçerme ilişkisi(Composition)** veri elemanı olarak kullanma anlamına gelir.

Örneğin:

```
class Y
{
        //...
}
class X
{
        pirivate Y m_y;

        public X()
        {
                M_y = new Y()
                //...
        }
}
//...
}
```



İçerme ilişkisi UML sınıf diyagramlarında içi dolu bir yuvarlakçık veya baklavacılıkla temsil edilir. Bu içi dolu baklavacık içeren sınıf tarafından bulunur. İçerme ilişkisi ingilizce **"Has a"** şeklinde yazılır.

Örneğin: Elimizde motor özellikleri üzerinde işlem yapan bir motor sınıfı olsun. Biz bir otomobil sınıfı yazmak isteyelim. Otomobil bir çeşit motor değildir. Motoru içermektedir. O halde otomobil sınıfını motor sınıfından türetmemeliyiz. Otomobil sınıfı motor sınıfını veri elemanı olarak kullanmalıdır.

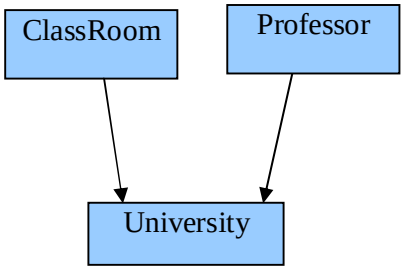
Nesne yönelimli programlama tekniğinde iki sınıf arasında Birkaç ilişki daha söz konusudur. Çok karşılaşılan bir ilişki biçimi **toplanma**(**aggregation)**dır. Toplanma ilişkisi içerme ilişkisine benzemektedir. Fakat aralarında küçük ama önemli farklılıklar vardır.

1. İçerme ilişkisinde içeren nesne ile içerilen nesnenin ömürleri aynıdır. Tipik olarak içerilen nesne içeren sınıfın başlangıç fonksiyonunda yaratılır. Örneğin: Motor otomobilin dışında bağımsız amaçlarla kullanılamamaktadır. Halbuki toplanma ilişkisinde içerilen nesne daha önce yaratılmıştır. Tipik olarak içeren sınıf onun referansını başlangıç fonksiyonuna alarak bir veri elemanına yerleştirir.

```
class ClassRoom
{
        //...
}
class ;Professor
{
        //...
}
class University
{
        class M_cr;
        Professor M_p;

        public University(Profesor p)
```

```
        {
                M_cr = new ClassRoom();
                M_p = p;
        }
        //...
}
//....
Professor p = new Professor("Jhon Smith");
//...
University u = new University (p);
```

Görüldüğü gibi Professor University tarafından kullanılmaktadır fakat professor universite olmadan da anlamlıdır. Toplanma ilişkisinde bir nesne birden fazla sınıf tarafından kullanılabilir. Fakat içerme ilişkisinde böyle bir durum söz konusu değildir.
Toplanma ilişkisi içi boş bir yuvarlakçık ya da baklavacıkla temsil edilir. **İçi boş kutucuk ya da yuvarlakçık.**
Nesne yönelimli programlama tekniği ile bir problem modellenecekse tüm gerçek nesneler ve kavramlar sınıflar la temsil edilmeli sonra sınıflar arası ilişkiler belirlenmelidir. Nihayet kodlama aşamasına geçilir.



Türetme Durumunda Başlangıç Fonksiyonlarının Çağrılması: Türemiş sınıfı türünden bir nesne new operatörü ile yaratıldığında yalnızca türemiş sınıfın başlangıç fonksiyonu çağrılmaz. Taban sınıfın başlangıç fonksiyonu da çağrılır. Türemiş sınıfın başlangıç fonksiyonu nesnenin türemiş sınıf datalarına taban sınıfın başlangıç fonksiyonu nesnenin taban sınıf datalarına değerlerini verir. Başlangıç fonksiyonlarının çağrılma sırası önce taban sonra türemiş sınıf şeklindedir.

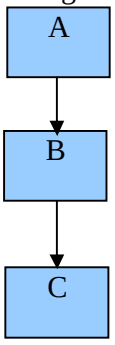
Örnek:

```
using System;
namespace CSD
{
   class App
   {
      public static void Main()
      {
        B x = new B();
        Console.WriteLine("{0} {1}", x.ValA, x.ValB);
      }
   }
   class A
   {
      private int m_a;

      public A()
      {
```

```
        m_a = 20;
    }

    public int ValA
    {
        get {return m_a;}
        set {m_a = value;}
    }
    protected int m_x;

    protected void Foo()
    {
    }

  }
  class B : A
  {
    private int m_b;

    public B()
    {
        m_b = 10;
    }

    public int ValB
    {
        get {return m_b;}
        set {m_b = value;}
    }
    public void Bar()
    {
        Foo();
    }

  }
}
```

Derleyici tipik olarak türemiş sınıf başlangıç fonksiyonunun ana bloğunun başına gizlice yerleştirdiği çağırma kodu yoluyla taban sınıfın başlangıç fonksiyonunu çağırır.

Aşağıdaki gibi bir dizi türetme yapılmış olsun:



Burada C sınıfı türünde bir nesne yaratıldığında balangıç fonksiyonları önce A sonra B sonra C

olacak biçimde çağrılır.
Türemiş sınıfın taban sınıfın hangi başlangıç fonksiyonunu çağıracağı :base(...) sintaksıyla belirlenir.
Örneğin:

```
public B(int b) : base(10)
{
        m_b = b;
        //...
}
```

Burada 10 int türden olduğuna göre taban sınıfın int parametreli başlangıç fonksiyonu çağrılacaktır. Türemiş sınıfın başlangıç fonksiyonunun parametreleri doğrudan base parantezi içinde kullanılabilir.

Örneğin:

```
public B(int a, int b) : base(a)
{
        m_b = b;
        //...
}
```

Türemiş sınıf balangıç fonksiyonunda base sintaksı hiç kullanılmayabilir. Bu durumda taban sınıfın default başlangıç fonksiyonu çağrılır. Yani base sintaksını hiç yazmamakla :base() aynı anlamdadır.

base sintaksı yalnızca başlangıç fonksiyonlarında belirtilenilir.

Anahtar Notlar: Bir sınıfın doğrudan taban sınıfı(direct base classes) hemen yukarısındaki taban sınıftır. Diğer taban sınıflarına dolaylı taban sınıfları (indirect base classes) denir. Bir sınıfın taban sınıfları denildiğinde onun bütün taban sınıfları anlaşılır.

base sintaksıyla ancak doğrudan taban sınıf belirtilir. Yani her sınıf doğrudan taban sınıfının başlangıç fonksiyonunu belirleyebilir.

Anahtar Notlar: Sınıfının default balangıç fonksiyonuna sahip olması her zaman iyi tekniktir. Yani sınıfın herhangi bir başlangıç fonksiyonu varsa derleyici default başlangıç fonksiyonunu yazmaz. Programcı içi boş bile olsa default başlangıç fonksiyonunu yazmalıdır.

:this(...) Sintaksı: Sınıfın başlangıç fonksiyonunda this Ya da base sintaksından sadece biri kullanılabilir. İkisi birden kullanılamaz. This sintaksı kendi sınıfının başka bir başlangıç fonksiyonunun çağrılacağını belirtir. Yine çağırma işlemi başlangıç fonksiyonunun ana bloğunun başına yerleştiren gizli bir çağırma koduyla yapılmaktadır. Yani this sintaksıyla belirtilen başlangıç fonksiyonu daha önce çağrılmaktadır.

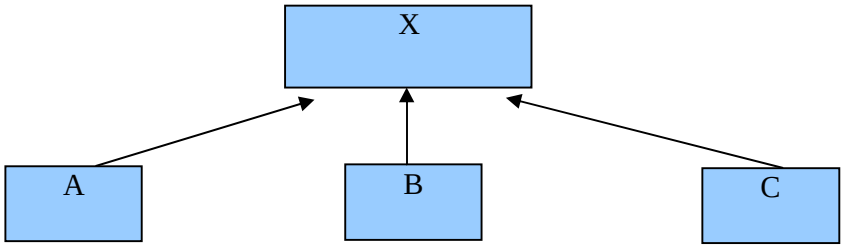
```
class A
{
        //...
}

class B:A
{
        public B()
        {
                //...
        }
```

```
        public B(int a) : this()
        {
                //...
        }
        //...
}
```

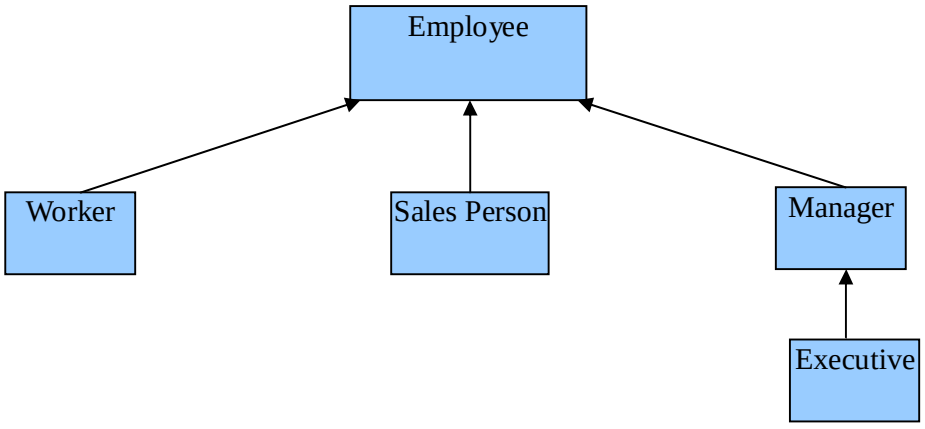
Burada B sınıfının int parametreli başlangıç fonksiyonu ile nesne yarattığımızı düşünelim. Bu başlangıç fonksiyonu kendi sınıfının default başlangıç fonksiyonunu çağıracak o da taban sınıfın default başlangıç fonksiyonunu çağıracaktır. Yani başlangıç fonksiyonlarının çağrılma sırası yine önce taban sonra türemiş biçiminde olacaktır.

Türetme Durumunda Çeşitli Örnekler:

1. Proje için A, B, C sınıfları tasarlayacak olalım fakat bu sınıfların ortak bazı elemanları olsun. Bu ortak elemanların yeniden her sınıfta bulundurulması kod tekrarına yol açar. Doğru teknik ortak elemanların bir taban sınıfta toplanması ve A, B, C snıflarının o taban sınıftan türetilmesidir



2. Bir personel takip programı yazacak olalım. Bu programda çeşitli çalışan gruplarını çeşitli sınıflarla temsil etmek isteyelim. Çalışan kişiler hangi görevde olursa olsun isim gibi adresi gibi ortak elemanlara sahiptir. İşte bu ortak elemanlar employee isimli bir sınıfta toplanabilir. Diğer sınıflar bu sınıftan türetilebilir.

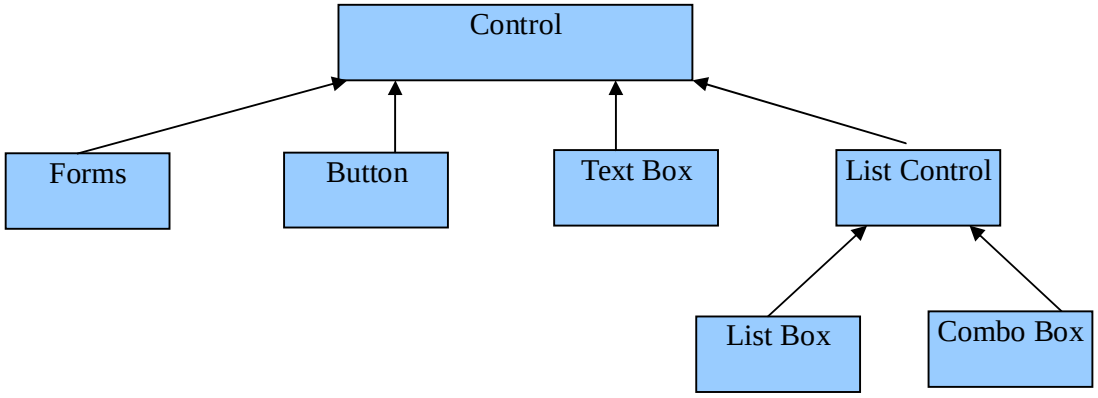


3. Görüldüğü gibi executive de bir çeşit managerdir. Windows işletim sisteminde ekranda bağımsız olarak kontrol edilebilen bölgelere pencere denilmektedir. Programın ana penceresi de, düğmeler de, edit alanları dahep birer penceredir. Bütün pencerelerin ortak bazı özellikleri vardır. Her pencere için sözkonusu olabilecek ortak birtakım özellikler ve eylemler sözkonusudur. Örneğin her pencerenin bir zemin rengi vardır. Her pencerenin bir konumu söz konusudur. Her pencere görünür veya görünmez hale getirilebilir. İşte.net 'in GUI kütüphanesinde pencereye ilişkin tüm ortak özellikler tepedeki kontrol sınıfında

toplanmıştır. Diğer sınıflar bu kontrol sınıfından türetilmiştir. Örneğin kontol sınıfının back color isimli property elemanı pencerenin zemin rengini belirlemede kullanılır. O halde biz bu elemanı tüm pencere sınıflı sınıflarda kullanabiliriz.

Button butonOk = new Button();
buttonOk.BackColor = Color.Red;

List Box ile Combo Box birbirlerine çok benzer ögelerdir. Microsoft bunların ortak ögelerini List Control ortak sınıfında toplamıştır.



System.Object Sınıfı: System isim alanındaki Object sınıfı tüm sınıfların taban sınıfı durumundadır(Her şey bir nesnedir). C# da her şey doğrudan ya da dolaylı olarak object sınıfından türemiştir. Biz türetme sintaksı kullanmamış olsak bile C# derleyicisi sınıfın System.Object sınıfından türetilmiş olduğunu varsayar.

```
class Sample
{
        //...
}
```

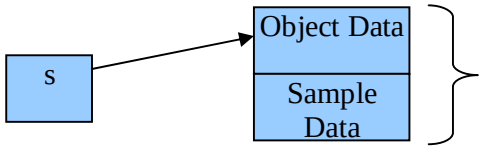
ile

```
class Sample : System.Object
{
        /....
}
```

aynı anlmadadır.

Sınıfın açıkça object sınıfından türetilmesi error a yol açmaz ama gerek yoktur.
System.Object sınıfı çok kullanıldığı için **"object"** anahtar sözcüğü ile temsil edilmiştir.

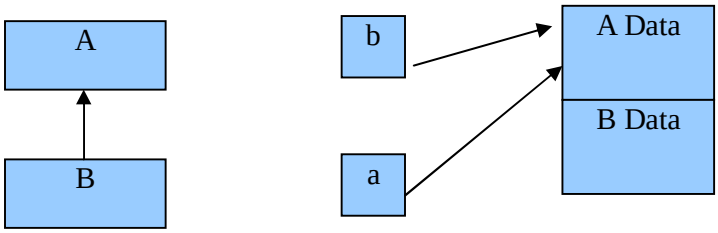
Anahtar Notlar: Bu durumda aslında her nesnenin bir de object kısmı vardır.



Tüm sınıflar object sınıfından türetilmiş olduğuna göre biz bir sınıf referansı ile object sınıfının public elemanlarını kullanabiliriz. Object sınıfının public elemanları ileri de ele alınacaktır.

Türemiş Sınıftan Taban Sınıfa Referans Dönüştürmeleri: C# da rastgele iki sınıf referansı birbirine atanamaz. Aynı türden iki referans birbirine atanabilir. Fakat ayrıca türemiş sınıf türünden referans taban sınıf türünden referansa doğrudan atanabilmektedir.
Türemiş sınıf referansı taban sınıfı referansına atandığında artık taban sınıf referansı türemiş sınıf nesnesinin taban sınıf parçasını gösteriyor durumda olur.

B b = new B;
A a;
a = b;

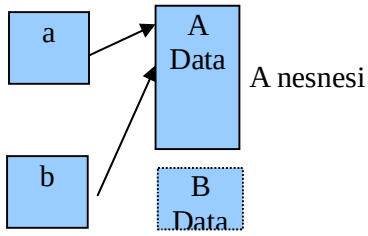


Biz şimdi a referansı ile işlem yaptığımızda bağımsız bir a nesnesi üzerinde değil B'nin A kısmı üzerinden işlem yapmış oluruz.

```
using System;

namespace CSD
{
        class App
        {
                public static void Main()
                {
                        Console.WriteLine ("m_a = {0} m_b = {1}", b.valA, b.valB);
                        A a;
                        a = b;
                        a.valA = 100;
                        Console.WriteLine ("m_a = {0} m_b = {1}", b.valA, b.valB);
                }
        }
class A
{
        private int m_a;

        public A (int a)
        {
                m_a = a;
        }
        public int valA
        {
                get { return m_a;}
                set { m_a = value;}
        }
}
class B : A
{
        private int m_b;

        public B (int a, int b) : base(a)
        {
                m_b = b;
        }
```

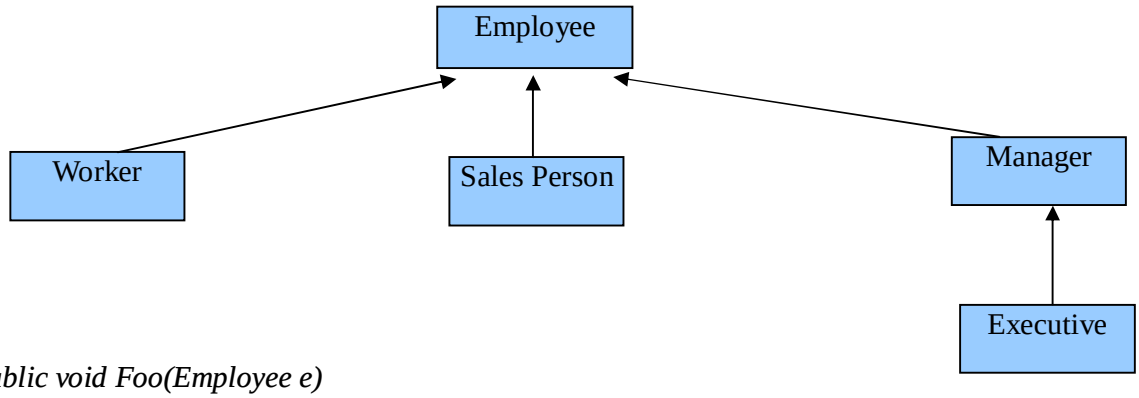
```
        public int valB
        {
                get { return m_b;}
                set { m_b = value;}
        }
}
```

Türemiş sınıf referansı taban sınıf referansına atanabilir fakat tersi doğru değildir. Eğer atababilseydi güvensiz bir durum oluşurdu.



Eğer böyle bir durum sözkonusu olsaydı b referansı ile biz gerçekte olmayan datalara erişebilirdik.

Türemiş Sınıf Referansının Taban Sınıf Referansına Atanmasının Anlamı:
Bir türetme şemasında taban sınıf türünden referans parametreli bir fonksiyon olsun. Biz bu fonksiyonu tüm türemiş sınıf nesneleriyle çağırabiliriz. Böylece bu fonksiyon tüm nesnelerin ortak özellikleri konusunda işlem yapabiliriz.




```
Public void Foo(Employee e)
{
        //...
}
Worker w = new Worker();
Foo(w);
Exacutive e = new Exacutive();
Foo(e);
```

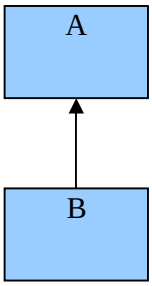
Görüldüğü gibi türtme şemasında yukarıya çıkıldıkça genelleşme aşağıya inildikçe özelleşme artmaktadır. O halde taban sınıf türünden parametreye sahip fonksiyon genel bir fonksiyondur.

Örnek:

```
using System;

namespace CSD
{
   class App
   {
      public static void Main()
      {
         Worker w = new Worker();

         w.Name = "Kaan Aslan";
         w.No = 123;
         w.Department = "Üretim";

         Manager m = new Manager();
         m.Name = "Ali Serçe";
         m.No = 345;
         m.Prim = 1000;

         Foo(w);
         Foo(m);
      }

      public static void Foo(Employee e)
      {
         Console.WriteLine("Adi Soyadi: {0}  No: {1}", e.Name, e.No);
      }
   }

   class Employee
   {
      private string m_name;
      private int m_no;

      public Employee()
      { }
      public Employee(string name, int no)
      {
         m_name = name;
         m_no = no;
      }

      public string Name
      {
         get { return m_name; }
         set { m_name = value; }
      }

      public int No
      {
         get { return m_no; }
```

```
            set { m_no = value; }
        }
    }

    class Worker : Employee
    {
        private string m_department;

        public Worker()
        { }

        public Worker(string name, int no, string department)
            : base(name, no)
        {
            m_department = department;
        }

        public string Department
        {
            get { return m_department; }
            set { m_department = value; }
        }
    }

    class Manager : Employee
    {
        private double m_prim;

        public Manager()
        { }
        public Manager(string name, int no, double prim)
            : base(name, no)
        {
            m_prim = prim;
        }

        public double Prim
        {
            get { return m_prim; }
            set { m_prim = value; }
        }


        //...
    }
}
```

Referansların Static ve Dinamic Türleri: Bir referansın statik türü bildirimde belirtilen türüdür. Örneğin:



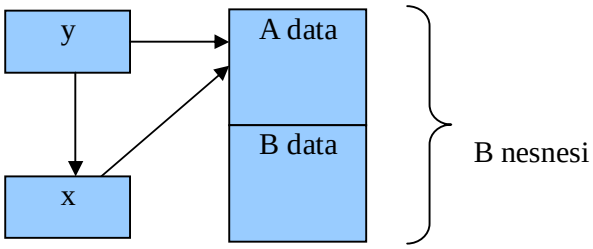
A x;

B y;

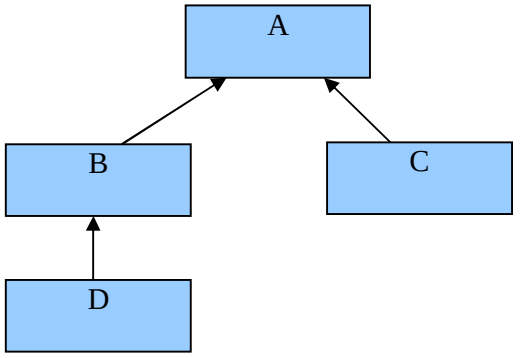
Burada x'in static türü A y'nin static türü B'dir. Bir referans daha büyük bir nesnenin bir kısmını gösteriyor olsun referansın gösterdiği yerdeki nesnenin en geniş halinin yani bütününün türüne o referansın dinamic türü denir.

Örneğin

```
A a;
B y;
y=new B();
x=y
```



Burada y nin static türü B ve dinamic türü de B dir. X'in statik türü A dinamic türü B dir.




```
B m  = new D();

A  k;
k =m;
```

Burada m referansının statik türü D dinamik türü B dir. K referansının statik tür A dinamik tür D'dir. Statik tür hiç değişmemektedir. Fakat dinamik tür değişebilir.

```
A a;
B b = new B();
C c = new C();
D d = new D();

a = b;
//...
```

a = c;
//...
a = d;
//...

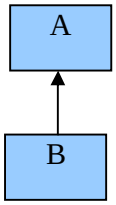
Burada a'nın statik türü hiç değişmemektedir ve A'dır. Halbuki dinamik tür değişebilir.
Referansın statik türüyle dinamik türü ancak bir türetme söz konusu olduğunda türemiş sınıf referansı taban sınıf türünden referansa atandığında farklı olur.
Madem ki her sınıf doğrudan ya da dolaylı object sınıfından türetilniştir. O halde object türünden referansa her türden referans atanabilir.

A a = new A();
B b = new B();
Object o;
o = a;
//...
o = b;
Burada o'nun statik türü object dinamik türü önce A sonra B olmaktadır.

Aşağıya Doğru Dönüştürmeler İşlemleri: Taban sınıftan türemiş sınıfa doğrudan atama yapamayız. Fakat tür dönüştürme operatörü ile dönüştürme uygulayarak atama yapabiliriz.

A a = new B();
b;

b = a; ---> error!
B = (B) a; ---> Geçerli

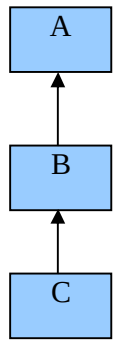


Tabandan türemişe doğru tür dönüştürme operatörü ile yapılan dönüştürmelerde her zaman derleme aşaması başarıyla gerçekleştirilir. Fakat programın çalışma zamanı sırasında ayrıca akış dönüştürme noktasına geldiğinde CLR dönüştürmenin haklı bir dönüştürme olup olmadığını da sınar. Eğer dönüştürülmek istenen referansın dinamik türü dönüştürülmek istenen türü içeriyorsa dönüştürme haklıdır. Çünkü bu durumda olmayan bir dataya erişme söz konusu olmaz. Fakat dönüştürülmek istenen referansın dinamik türü dönüştürülmek istenen türü içermiyorsa bu durum güvensiz bir durumdur. Çalışma zamanı sırasında Exception oluşur.

A a;
C c = new C();
a = c;

B x ;
x = (B) a; --> Haklı durum

C y;
y = (C) a; --> haklı

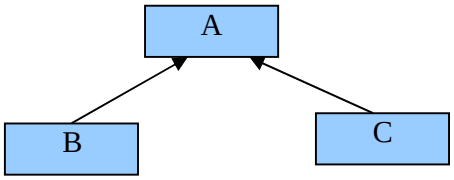


A a = new B();
B b;
b = (B) a; ---> haklı dönüştürme

//...
C c;
c = (C) a; ---> haksız dönüştürme Exception oluşur.

Aralarında türetme ilişkisi bulunmayan iki sınıf referansı tür dönüştürme operaötürü ile dönüştürülmeye çalışılırsa daha derleme aşamasında error oluşur. Çünkü böyle bir dönüştürmenin haklı olma olasılığı yoktur.

Örneğin:



B b = new B();
C c;
c = (C) b; Derleme aşamasında error oluşur.
Burada hiçbir zaman b türünden bir referansın dinamik türü C sınıfını kapsayamaz. O halde böyle bir kontrolün çalışma zamanına bırakılmasına da gerek yoktur.

Örneğin:

B b = new B();
A a = b;

C c;
C = (C)a; ---> Derleme başarılı olur. Çalışma zamanında çöker.(Exception)

Delegeler: Amacı fonksiyon tutmak ve sonra tuttuğu fonksiyonun çağrılmasını sağlamak olan özel sınıflara delegeler(delegate) denilmektedir. Fonksiyon tutmak aslında aşağı seviyeli düşünüldüğünde yalnızca fonksiyonun adresini tutmak anlamına gelmektedir. Adresi tutulan fonksiyon daha sonra çağrılabilmektedir.
Delege sınıf bildiriminin genel biçimi şöyledir.

delegate <geri dönüş değerinin türü> <isim> ([parametre bildirimi]);

Örneğin:
delegate void Proc();
delegate int Test(int a, int b);

Bildirim daha çok bir fonksiyon bildirimine benzese de aslında bir sınıf bildirimidir. Proc ve Test birer sınıf ismidir.
Delegeler aslında birer sınıf olduğuna göre kategori olarak referans türlerine ilişkindir.
Örneğin:
Proc p;
Burada p bir referanstır. Yani Proc türünden bir sınıf nesnesinin adresini tutar. Bir delege her türden fonksiyonu tutamaz. Geri dönüş değeri ve parametrik yapısı belirli türde olan fonksiyonları tutar. Yukarıdaki örnekte Proc sınıfı geri dönüş değeri void ve parametresi olmayan Test sınıfı ise geri dönüş değeri int ve parametre leri int ve int olan fonksiyonları tutar. Parametre değişkenlerinin hiçbir önemi yoktur.

Bir delege static olan ya da static olmayan fonksiyonları tutabilir. Önemli olan parametrik yapının ve geri dönüş değeri türünün uygun olmasıdır.

d bir delege referansı olmak üzere d(.....) bu delege referansının tuttuğu fonksiyon çağrılabilir. Bu işlemin sonucunda çağıran fonksiyonun geri dönüş değeri elde edilir.
Bir delege nesnesi delege sınıfının başlangıç fonksiyonuyla yaratılır. Tüm delege sınıflarının tek parametreli bir başlangıç fonksiyonları vardır. (default başlangıç fonksiyonu yoktur.) Bu tek parametreli başlangıç fonksiyonu parametre olarak delegenin tutacağı fonksiyonun ismini alır.
Örneğin:

```
delegate void Proc();
//...
Proc p = new Proc(Sample.Func);
p();
```

Yani görüldüğü gibi delegenin tutacağıfonksiyon delege nesnesi yaratılırken verilir.

```
using System;

namespace CSD
{
   delegate void Proc();

   class App
   {
      public static void Main()
      {
         Proc p = new Proc(Sample.Func);
         p();

      }
   }
   class Sample
   {
      public static void Func()
      {
         Console.WriteLine("Func");
      }
   }
}
```

Bir delege herhenagi bir sınıfın static yada static olmayan bir fonksiyonunu tutabilir. Yeterki geri dönüşdeğeri ve parametrik yapısı uyuşsun. Eğer delege sattic olmayan bir fonksiyon tutacaksa bu fonksiyon delegeye bir referansla birlikte verilmelidir. Böylece delege yoluyla fonksiyon çağrıldığında fonksiyon bu referansla çağrılacaktır.
Örneğin:

```
using System;

namespace CSD
{
   delegate void Proc();
   class App
   {
      public static void Main()
      {
         Sample s = new Sample(100);
```

```
            Proc p = new Proc(s.Disp);
            p();

        }
    }
    class Sample
    {
        private int m_a;

        public Sample(int a)
        {
            m_a = a;
        }

        public void Disp()
        {
            Console.WriteLine(m_a);
        }

        public static void Func()
        {
            Console.WriteLine("Func");
        }
    }
}
```

Eğer delege nesnesi yaratılırken verilen fonksiyon nokta operatörü olmadan niteliksiz bir biçimde belirtilmişse fonksiyon ismi sınıf faaliyet alanında aranır. Eğer static ise sanki isim sınıf ismi ile yazılmış gibi static değilse sanki this ile yazılmış gibi kabul edilir.

Örnek:

```
using System;

namespace CSD
{
    delegate int Proc(int a, int b);

    class App
    {
        public static void Main()
        {
            Proc p = new Proc(Sample.Add);

            int result;

            result = p(10, 20);

            Console.WriteLine(result);

        }
```

```
    }
    class Sample
    {
      public static int Add(int a, int b)
      {
         return a + b;
      }
    }
}
```

Bir delege nesnesi hiçbir fonksiyon tutmadan yaratılamaz. Kesinlikle nesnenin en az bir fonksiyon tutması gerekir.

Aynı türden iki delege referansı + operatörü ile toplama işlemine sokulabilir. Bu işlem sonucunda yeni bir delege nesnesi yaratılır. Yeni delege nesnesinin fonksiyon listesi iki operandın fonksiyon listesinin birleşiminden oluşur.

Örneğin:

```
delegate void Proc();
//...
Proc d1 = new Proc(Sample.Foo);
Proc d2 = new Proc(Sample.Bar);
Proc d3;

d3 = d1 + d2
```

Bir delege nesnesi başlangıçta tekbir fonksiyon tutacak biçimdedir. Fakat daha sonra + operatörü uygulanarak birden fazla delege nesnesi tutan delege nesneleri yaratılabilir. Örneğimizde d3 referansının gösterdiği delege nesnesi 2 fonksiyon tutmaktadır.

```
using System;

namespace CSD
{
   delegate void Proc();

   class App
   {
      public static void Main()
      {
         Proc d1 = new Proc(Sample.Foo);
         Proc d2 = new Proc(Sample.Bar);
         Proc d3;

         d3 = d1 + d2;

         d3();

      }
   }
   class Sample
```

```
  {
    public static void Foo()
    {
       Console.WriteLine("Foo");
    }
    public static void Bar()
    {
       Console.WriteLine("Bar");
    }
  }
}
```

Bir delege referansı fonksiyon çağırma operatörü ile kullanıldığında delege referansının tuttuğu tüm fonksiyonlar tek tek çağrılır. Çağırma işleminden elde edilen geri dönüş değeri son çağrılan fonksiyonun geri dönüş değeridir.

A + =B işlemi A = A+B anlamına geldiğine göre aşağıdaki işlemde geçerlidir.

```
Proc d1 = new Proc(Sample.Foo);
Proc d2 = new Proc(Sample.Bar);
d2 +=d1;
```

Burada d2 referansı artık d2 + d1 işleminin sonucunda yaratılan nesneyi göstermektedir.

D1 + d2 işleminde operandlardan biri NULL değerini içeriyorsa yeni bir delege nesnesi yaratılmaz. İşlem sonucunda NULL olmayan delege referfansının aynısı elde edilir. Eğer her iki operand da NULL değerde ise işlemden NULL referans elde edilir. Bu durumda **Örneğin:** (Sırayla)

```
Proc d = NULL;
d += new Proc (Sample.Foo)
d+= new Proc (Sample.Bar)
d();
```

Bu işlemlerin sonucunda Foo ve Bar fonksiyonları çağrılacaktır.

Aynı türden iki delege – operatörü ile çıkartılabilir. Bu durumda yeni bir delege nesnesi yaratılır. Yaratılan delege nesnesinin fonksiyon listesi sol taraftaki delegenin fonksiyon listesinden sağ taraftaki delegenin fonksiyon listesinin çıkartılmasıyla elde edilen nesnedir.

Örneğin:
d1'in fonksiyon listesi Func1, Func2, Func3 olsun
d2'nin fonksiyon listeside Func2 olsun.
d1-d2 işleminden elde edilen nesnenin fonksiyon listesi Func1, Func3 içerir.
Eğer d1-d2 işleminde d1'in fonksiyon listesinde d2'nin fonksiyon listesi birden fazla yerde varsa listede son görülen fonksiyonlar çıkartılır.
Örneğin:
d1'in fonksiyon listesi Func1, Func2, Func3, Func2 olsun
d2'nin fonksiyon listeside Func2 olsun.
d1-d2 işleminden yeni bir nesne yaratılır. Yaratılan nesnenin fonksiyon listesi Func1i Func2, Func3 olur.
Yani son eklenen fonksiyon çıkartılır. Eğer d1 in fonksiyon listesinde d2 i yoksa yeni bir nesne yaratılmaz. d1 referansı elde edilir.
D1 referansı NULL ise ya da her iki referansta NULL ise sonuçta NULL elde edilir. Eğer d2 NULL ise işlemler sonucunda d1 referansı elde edilir. Eğer d1-d2 işleminde d1 ve d2 aynı fonksiyon

listesine sahip ise işlemden yine NULL değeri elde edilir.

Framwork 2.0 ile birlikte fonksiyonlardan uygun delege türlerine doğrudan dönüştürme kuralı eklenmiştir. Bu kurallara göre

1. Bir delege referansına doğrudan fonksiyon atanabilir. Bu işlem yeni bir delge nesnesi yaratılıp içine ilgili fonksiyonun yerleştirilmesi ile aynı anlamdadır. Yani
   Proc d;
   d = Sample.Foo; ile
   Proc d ;
   d = new Proc(Sample.Foo); eşdeğerdir.
2. Bir delege referansı ile bir fonksiyon toplanabilir ya da çıkarılabilir. Bu durum da yeni bir delege nesnesi yaratılıp fonksiyonlar delege nesnesinin içine yerleştirilir. İşleme bu delege nesnesi sokulur. **Örneğin:**
   Proc d1, d2;
   d2 = d1 + Sample.Foo; ile
   d2 = d1 + new Proc(Sample.Foo) aynıdır.
   Ya da
   d += Sample.Foo;
   d = d + new Proc(Sample.Foo); aynı anlamdadır.

Bir delege sınıf ya da yapının veri elemanı durumunda olabilir.
Örneğin:

```
class Sample
  {
    public Proc DoSomething;
    //...
  }
Sample s = new Sample();
s.DoSomething = Foo();
s.DoSomething();
```

Sınıfların ve Yapıların EVENT Elemanları: Bir sınıfın ya da yapının delege türünden veri elemanları event eleman yapılabilir.
Örneğin:

```
delegate void Proc();
  class Sample
  {
    public event Proc  DoSomething;
    //...
  }
```

Görüldüğü gibi event anahtar sözcüğü delege türünden önce getirilmek zorundadır. İki nokta önemlidir.

1. Bir veri elemanının event türünden olabimesi için onun delege türünden olması gerekir.
2. Yerel bir değişken event yapılamaz. Ancak sınıfın ya da yapının veri elemanları event yapılabilir.

Sınıfın ya da yapının event elemanı sınıf içinde tam bir delege özelliğiyle kullanılabilir. Fakat sınıfın dışında yalnızca += ve -= operatörleriyle kullanılabilir. Yani sınıfın dışında biz event elemana atama yapamayız. Delege fonksiyonlarını çağıramayız. Event belirleyicisi delegeyi dışarıya karşı kısıtlamaktadır.

Event delege elemanının fonksiyolları dışarıdan çağrılamayacağına göre şöyle bir yöntem izlenebilir. Programcı sınıfa public bir fonksiyon yerleştirir ve bu fonksiyon içinden event delege

fonksiyonlarını çağırır ve dışarıdan da bu fonksiyon çağrılır.

```
Sample s = new Sample();
        s.DoSomething += new Proc(Foo);
        s.Fire();
```

Bazı uygulamalarda sınıfın delege elemanlarına dışarıdan atama yapmak ya da delege fonksiyonlarını dışarıdan çağırmak güvensizliğe yol açmaktadır. Bu nednele delege elemenların dışarıya kısıtlanması çok kullanılan bir özelliktir.

Türemiş sınıfın taban sınıf elemanlarına eriştiğini biliyoruz. Fakat türemiş sınıf taban sınıfın event elemanlarına yine ve yalnızca += ve -= işlemlerini uygulayabilir.

Delegelerin ve Event Elemanların Uygulamadaki Kullanımları:

.net programlamada özellikle GUI kısmında delegeler yoğun olarak kullanılmaktadır. GUI programlamada çeşitli sınıfların event delege elemanları vardır. Buraya eklediğimiz delege fonksiyonları çeşitli olaylar gerçekleştiğinde FrameWork tarafından çağrılmaktadır. Biz bir olay gerçekleştiğinde bir fonksiyonun çağrılmasını istiyorsak o fonksiyonu bir delege nesnesinin içine yerleştirip o delege nesnesinde += operatörleriyle ilgili sınıfın event delege elemanına eklemeliyiz. Örneğin Button sınıfının Click isimli event elemanı EventHandler isimli bir delege türündendir. Ve düğmeye tıklandığında bu event delege elemanının fonksiyonları çağrılır. O halde örneğin biz düğmeye tıklandığında Click Handler isimli fonksiyonumuzn çağrılmasını istiyorsak bu işlemi şöyle gerçekleştirebiliriz.

```
buttonOK.Click += new EventHandler(this.ClickHandler);
```

Bir eventle karşılaşıldığında şu sorulara yanıt verilmelidir.

1. event eleman hangi delege türündendir.
2. event e girdiğimiz fonksiyonlar hangi olay gerçekleştiğinde çağrılmaktadır?
3. Event fonksiyonun da ne yapmalıyım.

EVENT üzerinde bir Uygulama: Timer Mekanizması; System.Threading isim alanındaki Timer sınıfı belirli bir periyotta belirlediğimiz bir fonksiyonu çağırmaktadır. Timer sınıfı mscorlib dll içindedir. Timer sınıfının aşağıdaki başlangıç fonksiyonu çeşitli belirlemeleri alır.

```
public Timer{
        TimerCallBack callback,
        Object state,
        int dueTime,
        int period
}
```

Fonksiyonun birinci parametresi TimerCallBack isimli bir delege referansıdır. Yani bizden TimerCallBack türünden bir delege nesnesi istemektedir. TimerCallBack delegesi şöyledir.

```
Public delegate void TimerCallback(
        Objectstate
)
```

Fonksiyonun ikinci parametresi delege fonksiyonuna geçilecek değeri belirtir. Üçüncü parametre fonksiyon ilk kez çağrılmadan önce geçecek mili saniye cinsinden zamanı belirtmektedir.
Timer sınıfının Dispose fonksiyonu Timer ı yok etmektedir.

```
using System;
using System.Threading;
```

```
namespace CSD
{
   delegate void Proc();

   class App
   {
      public static void Main()
      {
         Timer t = new Timer(new TimerCallback(TimerProc), null, 0, 1000);
         Console.ReadLine();
      }
      private static void TimerProc(object state)
      {
         DateTime dt = DateTime.Now;
         System.Console.Write("{0:D2}:{1:D2}:{2:D2}\r", dt.Hour, dt.Minute, dt.Second);
      }
   }
      //...
}
```

Bir event mekanizmasına örnek olarak DirectoryWalker sınıfı verilebilir. Bu sınıf bir dizinden başlayarak dizin ağacını dolaşmakta her bulduğu dosya için programcının belirlediği bir fonksiyonu çağırmaktadır. Programcı isterse programdan false değeri ile geri dönüp işlemi bitirebilir.

```
delegate bool DirectoryProc(string file);

   class DirectoryWalker
   {
      private string m_dirPath;
      public event DirectoryProc Proc;

      public DirectoryWalker(string dirPath)
      {
         m_dirPath = dirPath;
      }

      public void Walk()
      {
         string[] files;

         files = DirectoryWalker.GetFiles(m_dirPath, "*.*", SearchOption.AllDirectories);
         foreach (string file in files)
            if (!Proc(file))
               break;
      }
   }

using System;
using System.Threading;
using System.IO;
```

```
namespace CSD
{
   delegate void Proc();

   class App
   {
      public static void Main()
      {
         DirectoryWalker dw = new DirectoryWalker(@"c:\windows");
         dw.Proc += new DirectoryProc(Test);
         dw.Walk();
      }
      public static bool Test(string path)
      {
         Console.WriteLine(path);
         return true;
      }

   }
   delegate bool DirectoryProc(string file);

   class DirectoryWalker
   {
      private string m_dirPath;
      public event DirectoryProc Proc;

      public DirectoryWalker(string dirPath)
      {
         m_dirPath = dirPath;
      }

      public void Walk()
      {
         string[] files;

         files = Directory.GetFiles(m_dirPath, "*.*", SearchOption.AllDirectories);
         foreach (string file in files)
            if (!Proc(file))
               break;
      }
   }
}
```

Operatör Fonksiyonları: Operatör fonksiyonları sınıflar ve yapılar türünden değişkenlerin sanki temel türlerdenmiş gibi aritmetik ve mantıksal operatörler ile kullanımına olanak verir. Örneğin a + bi biçiminde bir karmaşık sayıyı tutan Complex isimli bir sınıf düşünelim. Operatör fonksiyonları sayesinde biz iki karmaşık sayısı + operatörü ile toplayabiliriz.

```
Complex z1 = new Complex(3, 2), z2 = new Complex(5, 3), z3;
z3 = z1 + z2;
```

Şüphesiz operatör fonksiyonları olmasaydı aynı işlemi fonksiyon çağırarakta yapabilirdik.

```
z3 = Complex.Add(z1, z2);
```
Java da operatör fonksiyonları yoktur. Orada bu işlemler fonksiyon çağırarak yapılmaktadır. Operatör fonksiyonları yalnızca okunabilirliği arttırmaktadır. Bu fonksiyonlar sayesinde biz çeşitli türler üzerinde sanki aritmetik ya da karşılaştırma işlemi yapıyormuş gibi o türe özgü işlemler yapabiliriz.

Operatör fonksiyonlarının genel biçimi şöyledir.
```
public static <geri dönüşdeğerinin türü>operatör <operatör sembolü>([parametre bildirimi])
{
        //...
}
```

Operatör fonksiyonları public statik olmak zorundadır. Geri dönüş değerleri herhangi bir türden olabilir. Operatör fonksiyonlarının isimleri operatör anahtar sözcüğü ile operatör sembolünden oluşur.
Örneğin:
```
public static bool operatör operator ==(sample va, sample b)
{
        //...
}
```
Eğer operatör fonksyonu iki operandlı bir operatöre ilişkinse iki parametresi tek operantlı bir operatöre ilişkinse tek parametresi bulunmak zorundadır. Operaöt fonksiyonlarının enaz bir parametresi operatör fonksiyonunun yerleştirildiği sınıf ya da yapı türünden olmak zorundadır.

Bir operatör fonksiyonu gizlice operatör işlemi ile çağrılır.
Örneğin:
```
Sample x = new Sample(10), y = new Sample(20), z;
        z = x + y;
```
Burada derleyici Sample sınıfının operatör + fonksiyonunu araştırır. Böyle bir fonksiyon bulursa onu çağırır. x + y işleminden operatör + fonksiyonunun geri dönüş değeri elde edilecektir.

Örnek:
```
using System;
using System.Threading;
using System.IO;

namespace CSD
{
   class App
   {
      public static void Main()
      {
         Sample x = new Sample(10), y = new Sample(20), z;
         z = x + y;
         Console.WriteLine(z.A);
      }
   }
   class Sample
   {
      private int m_a;

      public Sample()
```

```
        {}

        public Sample(int a)
        {
            m_a = a;
        }
        public int A
        {
            get {return m_a;}
            set {m_a = value;}
        }
        public static Sample operator +(Sample x, Sample y)
        {
            Sample result = new Sample();
            result.m_a = x.m_a + y.m_a;
            return result;
        }
    }
}
```

Bu biçimde de operatörler komnine edilebilir. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample x = new Sample(10), y = new Sample(20), z = new Sample(30), k;
            k = x + y + z;
            Console.WriteLine(k.A);
        }
    }
    class Sample
    {
        private int m_a;

        public Sample()
        {}

        public Sample(int a)
        {
            m_a = a;
        }
        public int A
        {
            get {return m_a;}
            set {m_a = value;}
        }
        public static Sample operator +(Sample x, Sample y)
        {
            Sample result = new Sample();
            result.m_a = x.m_a + y.m_a;
```

```
            return result;
        }
    }
}
```

C# derleyicisi bir operatörle karşılaştığında önce operandların türlerini araştırır. Eğer operantların her ikiside temel türlere ilişkinse küçük türü büyük türe dönüştürerek normal işlemini yapar. Eğer operandlardan en az biri bir sınıf ya da yapı türündense o sınıf ya da yapı içinde işlemi yapacak operatör fonksiyonu araştırır. Bulursa onu çağırır.

Aynı isimli farklı parametrik yapılara sahip birden fazla operatör fonksiyonu bulunabilir. Örneğin iki Sample nesnesini toplayan operatör fonksiyonun yanı sıra bir Sample ve bir int toplamını yapan bir operatör fonksiyonu bulundurulabilir.

```
public static Sample operator +(Sample x, Sample y)
{
    Sample result = new Sample();
    result.m_a = x.m_a + y.m_a;
    return result;
}
public static Sample operator +(Sample x, int y)
{
    Sample result = new Sample();
    result.m_a = x.m_a + y;
    return result;
}
public static Sample operator +(int x, Sample y)
{
    return y + x;
}
```

Sample + int işlemini gerçekleştiren operaör fonkisyonu ile int + Sample işlemini yapamaz. Bunun için yukarıdaki gibi ayrı bir operatör fonksiyonu yazmak gerekir.

Operatör fonksiyonları için Overload Resulation işlemi şu şekilde gerçekleştirilir.

1. Derleyici A + B gibi bir işlemle karşılaştığında aday fonksiyonları her iki operanda ilişkin sınıf ve yapı içerisinden oluşturur. Örneğin Foo() + Bar() işlemini yapacak bir operatör fonksiyonu Foo sınıfının da Bar sınıfının da içinde bulunabilir.

```
class Foo
{
        //...
        public static operator +(Foo f, Bar b)
        {
                //...
        }
        //...
}
class Bar
{
        //...
        public static operator +(Foo f, Bar b)
        {
                //...
        }
```

```
        //...
}
```

derleyici a<op> b işleminde her iki sınıf yapıdaki operator <op> fonksiyonlarını aday fonksiyon olarak belirler. Tabii <op> a biçiminde tek operandlı bir operatörde kullanılabilir. Bu durumda derleyici aday fonksiyonları o sınıf ya da yapıdan seçecektir.

2. Derleyici a<op>b işlemindeki operandları argüman kabul ederek parametreleri doğrudan dönüştürmeye izin veren operator <op> fonksiyonlarını uygun fonksiyon olarak seçer.
3. Uygun fonksiyonları yarışa sokarak en uygun fonksiyonları tespit etmeye çalışır.

Burada önemli olan bir nokta şudur. Her iki sınıf ya da yapıda aynıişlemi yapabilecek eşdeğer kalitede operatör fonksiyonu bulunabilir. Bu durumda en uygun fonksiyon bulunamaz. Yukarıdaki örnekte olduğu gibi Foo + Bar işlemini yapabilecek hem Foo sınıfında hemde Bar sınıfında operatör + fonksiyonu var isederleme işlemi error ile sonuçlanır.

C# da olmayan bir operatörün operatör fonksiyonu yazılamaz. Örneğin @ sembolü C# da bir operatör belirtmez. O halde biz operator @ gibi bir operatör fonksiyonu yazamayız.

Operatör fonksiyonu yazarak operaörlerin önceliklerini değiştiremeyiz. Örneğin biz komplex sayı sınıfı için hem çarpı hemde artı operatör fonksiyonu yazmış olalım.
z1 + z2 * z3 burada önce * operatörü yapılır. Prantez kullanımının dışında bunu değiştiremeyiz.

Bir sınıf ya da yapı için bir operatör fonksiyonu yazacaksak bu fonksiyona yaptıracağımız işlem operatör sembolüne uygun olmalıdır. Örneğin + operatör fonksiyonuna * işlemi yaptırmak kötü bir tekniktir. İki tarihi toplayan bir operatör fonksiyonunu bir anlamı yoktur. Fakat iki tarihi çıkaran bir operatör fonksiyonun bir anlamı var.

Bu operatör fonksiyonları iki parametre almak zorundadır. Geri dönüş değerleri herhangi bir türden olabilir. Fakat tipik olarak geri dönüş değerleri ilgili sınıf ya da yapı türünden olursa operatör kombine edilebilir.

Karşılaştırma Operatör Fonksiyonlarının Yazımı: Karşılaştırma operatör fonksiyonları çiftler halinde yazılmaktadır. Çiftler şunlardır.

```
==      !=
<       >
<=      >=
```

Çiftlerden biri için operatör fonksiyonu yazılmışsa diğeri için de yazılmalıdır. Karşılaştırma operatör fonksiyonlarının geri dönüş değerleri herhangi bir türden olabilir. Fakat şüphesiz geri dönüş değerinin BOOL türden olması anlamlıdır. İki rasyonel sayı karşılaştırma fonksiyonları

++ ve – Operatör Fonksiyonlarının Yazımı: Bu operatör fonksiyonlarının yazımında önek ve sonek etki konusu önemlidir. Programcı arttırım ya da eksiltim etkisini parametresiyle aldığı nesne üzerinde doğrudan yapmamalıdır. Bu nesnenin bir kopyasını oluşturup bu kopya üzerinde arttırım ya da eksiltim uygulayıp buna geri dönmelidir. Aksi halde önek ve sonek etki doğru oluşmaz. Örneğin rasyonel sayı sayı sınıfı için ++ operatör fonksiyonu şöyle yazılmalıdır.

```
public static Rational operator ++(Rational r)
{
        Rational Result = new Rational();
```

```
        result.m_a = r.m_a +r.m_b;
        result.m_b = r.m_b;
        return result;
}
```
-- operatör fonksiyonu da şöyle yazılabilir.

```
public static Rational operator --(Rational r)
{
        Rational Result = new Rational();
        result.m_a = r.m_a - r.m_b;
        result.m_b = r.m_b;
        return result;
}
```

Tür Dönüştürme Operatör Fonksiyonlarının Yazımı: Bazen aralarında mantıksal bağ olan fakat tamamen farklı türlerden olan iki değişkeni birbirlerine atama yapmak isteyebiliriz. Örneğin bir rasyonel sayı devirli ondalık sayının açılımı olduğuna göre bir bir rasyonel sayıyı double türden bir sayıya dönüştürüp atama yapmak isteyebiliriz. Bunu mümkün hale getirmek için tür dönüştürme operatör fonksiyonu gerekmektedir. Tür dönüştürme operatör fonksiyonu implicit ve explicit olmak üzere ikiye ayrılmaktadır. Implicit olan explicit olanı kapsar. Imlicit operatör dönüştürme fonksiyonu atama durumlarında devreye girer. Explicit dönüştürme fonksiyonları tür dönüştürme operatörleri kullanılmak istendiğinde devreye girer.
Tür dönüştürme operatör fonksiyonlarının genel biçimi şöyledir.

public static <implicit/explicit> operator <dönüştürülecek tür> (<dönüştürülecek nesne bildirimi>)
Örneğin rasyonel sayı sınıfı için double türüne inmplicit dönüştürme yapan tür dönüştürme operatör fonksiyonu şöyle yazılabilir.

```
public satic implicit operator double(Rational r)
{
        return (double)r.m_a / r.m_b;
}
```
Tür dönüştürme operatör fonksiyonunun isimi operator anahtar sözcüğü ile dönüştürülecek türden oluşmaktadır. Yukarıdaki örnekte isim operator double biçimindedir. Tür dönüştürme operatör fonksiyonları için ayrıca bir geri dönüş değeri türü yazılmaz. Dönüştürülecek tür zaten geri dönüş değerinin türü anlamına gelir. **Örneğin:**

```
        Rational a = new Rationa(1, 2);
        double d;

        d = a;

        Console.WriteLine(d);
```
Dönüştürülecek değer olacak a implicit dönüştürme fonksiyonunaparametre yapılır. Bu fonksiyondan elde edilen değer dönüştürme değeri olarak belirlenir. Implicit dönüştürme Explicit dönüştürmeyi kapsadığına göre işlem aşağıdaki gibi de yapılabilirdi.

```
Rational a = new Rationa(1, 2);
        double d;

        d = (double)a;

        Console.WriteLine(d);
```

Fakat Explicit biçim Implicit biçimi kapsamamaktadır. Aynı dönüştürmeyi yapan hem Implicit hem de Explicit dönüştürme fonksiyonları yazılamaz.

Örneğin GUI programlamada çok sık kullanılan Point ve PointF isimli yapılar bir noktanın x ve y değerlerini tutan ve üzerlerinde bazı işlemleri yapmamızı sağlayan temel iki yapıdır. Bu yapılar System.Drawing isim alanı içindedir. Ve System.Drawing.dll dosyasında bulunmaktadır. **Örneğin:**

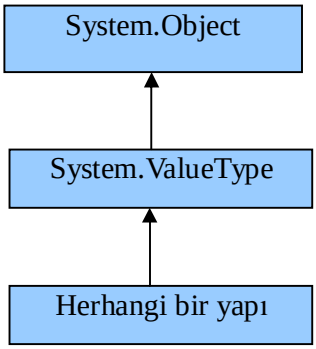
```
Point pt = new Point (2, 3);

Console.WriteLine("{0}, {1}", pt.X, pt.Y);
```

İşte nasıl int türünden float türüne atama yapabiliyorsak Ponit türünden PointF türüne atama yapabilmemizanlamlı olabilir. Çünkü Point ile PointF arasındaki tek fark Point yapısının elemanları int olarak saklaması PointF yapısının ise float olarak saklamasıdır. **Örneğin:**

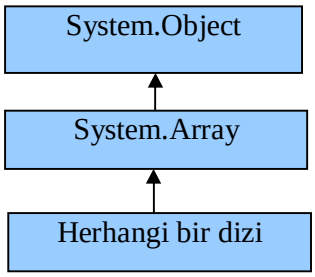
```
Point pt = new Point (2, 3);
PointF  ptf;

ptf= pt;

Console.WriteLine("{0}, {1}", ptf.X, ptf.Y);
```

Gerçektende Point yapısından PointF yapısına dönüştürme yapan Implicit bir tür dönüştürme operatör fonksiyonu vardır.

Yapıların Türetme Durumları: Bir yapıdan türetme yapılamaz. Yapı da başka bir yapı ve sınıftan türetilemez. Yani yapılar türtmeye kapalıdır. Bu nedenle yapılar protected ve protected internal elemanlara sahip olamaz. Fakat .net te tüm yapıların System.ValueType denilen bir sınıftan türetildiği varsayılmaktadır. System.ValueType sınıfı da System.Object sınıfından türetilmiştir. Programcı açıkça yapıyı System.ValueType sınıfından türetemez.

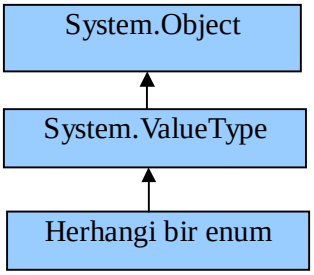


Dizilerin Türetme Durumları: Diziler de bir çeşit sınıftır fakat dizilerden de türetme yapılamaz. Diziler de bir sınıftan türetilemez. Fakat .net te tüm dizilerin System.Array isimli sınıftan türetildiği varsayılmaktadır. System.Array sınıfı da System.Object sınıfından türetilmiştir.



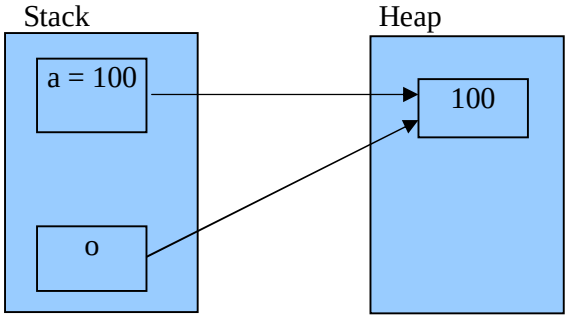
Örneğin length property si System.Array sınıfının bir elemanıdır.

Enum Türlerinin Türetme Durumu: Enum türleride türetilemez ve enum türlerinden de türetme yapılamaz. Fakat bütün enum türlerinin System.Enum isimli bir yapıdan türetildiği varsayılmaktadır. System.Enum yapısı da System.ValueType sınıfından türetilmiştir. System.ValueType sınıfıda System.Object sınıfından türetilmiştir.



Kutulama Dönüştürmesi(Boxing Conversion): Ne zaman bir yapı türünden değişken System.ValueType yada System.Object türünden bir referansa atansa o yapının heap te otomatik olarak bir kopyası çıkartılır. Ve referans yapı değişkeninin heapteki kopyasını gösterir duruma gelir. Bu işleme kutulama dönüştürmesi denilmektedir. Referanslar hiçbir zaman stack te bir yer gösteremez. Ancak heap te bir yer gösterir.

mt a = 100;
object 0;
o = a;



Kutulama dönüştürmesi sonrasında artık stackteki yapı nesnesi ile heapteki kopyası iki bağımsız değişkendir. Biri blok bittiğinde diğeri çöp toplayıcı tarafından yok edilecektir.

Kutuyu Açma Dönüştürmesi(Unboxing Conversion): System.ValueType ya da System.Object referansları yeniden tür dönüştürme operatörleriyle aşağıya doğru dönüştürülebilir. **Örneğin:**

int a = 100;
object o = a;

int b;
b = (int) o;

Aşağıya doğru dönüştürme işlemi sonucunda stack te ilgili yapı türünden geçici bir nesne yaratılır. Heapteki yapı stack teki bu geçici değişkene kopyalanır. İlgili ifade bittikten sonra bu geçici değişken yok edilecektir. Bu işleme kutuyu açma dönüştürmesi denilmektedir. Kutulama dönüştürmesi sabitlerle de yapılabilir. **Örneğin:**
object o = 123;

Burada 123 değeri int bir değer olarak heape taşınacaktır. O artık heapteki değeri gösterecektir. Kutulama ve kutuyu açma dönüştürmesi sayesinde herşey object türüne dönüştürülüp geri alınabilir.

Collection Sınıflar: Amacı birden fazla nesneyi belirli bir algoritmik yöntemle tutmak olan özel sınıflara collection sınıflar diyoruz. Collection sınıfların en ünlüsü ArrayList sınıfıdır. ArrayList dinamik büyütülen bir dizi görevini yapmaktadır. ArrayList sınıfının Add fonksiyonu vardır. Programcı eklemeyi bu Add fonksiyonuyla yapar. Sınıfın indeksleyicisi sayesinde istenilen indekse eleman elde edilebilir.

ArrayList her türden nesneyi tutabilir. Kendi içinde object türünden bir dizi vardır. Bu dizi duruma göre büyütülmektedir. Sınıfın Count ve Capacity isminde iki properties'i vardır. Capacity sınıf içindeki dizinin uzunluğunu Count ise dizinin içine yerleştirilmiş eleman sayısını verir. Her Add işlemi yapıldıkça Count değeri bir arttırılır. Caount Capacity değerine eriştiğinde Capacity değeri iki kat artar.

Bir ArrayList sınıfı aşağıdakine benzer bir biçimde yazılmıştır.

```
using System;
using System.Threading;
using System.IO;

namespace CSD
{
   class App
   {
      public static void Main()
      {
        ArrayList al = new ArrayList();
        for (int i = 0; i < 10; ++i)
           al.Add(i);
         for ( int i = 0; i< al.Count; ++i)
            Console.WriteLine((int)al[i]);
      }

   }
      class ArrayList
      {
         private object[] m_objs;
         private int m_count;
         private int m_capacity;

         public ArrayList()
         {
            //...
         }

         public int Add(object o)
         {
            if (m_count == m_capacity)
            {
               if (m_objs == null)
               {
                  m_objs = new object[1];
                  m_capacity = 1;
               }
               else
```

```
            {
                m_capacity *= 2;
                object[] temp = new object[m_capacity];
                Array.Copy(m_objs, temp, m_count);
                m_objs = temp;
            }
        }

        m_objs[m_count] = o;
        ++m_count;

        return m_count - 1;
    }

    public object this[int index]
    {
        get
        {
            return m_objs[index];
        }
        set
        {
            m_objs[index] = value;
        }
    }
    public int Count
    {
        get { return m_count; }
    }
    public int Capacity
    {
        get { return m_capacity; }
    }
    //...
  }
}
```

ArrayList sınıfı System.Collestions isim alanı içindedir. Add fonksiyonun dışında sınıfın faydalı pek çok fonksiyonu vardır.

```
using System;
using System.Threading;
using System.IO;
using System.Collections;

namespace CSD
{
   class App
   {
      public static void Main()
      {
         ArrayList al = new ArrayList();
```

```
            for (int i = 0; i < 10; ++i)
               al.Add(i);
            for (int i = 0; i < al.Count; ++i)
               Console.WriteLine((int)al[i]);
         }

      }
}
```

Array.List sınıfının isimli fonksiyonu diziyi kaydırarak belirli bir index'e insert işlemi yapar. Yani eklenen değer o index te olacak biçimde diziyi kaydırır.

```
public virtual void Insert(
      int index,
      Object value
)
```

ArrayList Ienumarable desteklediği için foreach dönüsünde kullanılabilir.

```
using System;

using System.Threading;
using System.IO;
using System.Collections;

namespace CSD
{
   class App
   {
      public static void Main()
      {
         ArrayList al = new ArrayList();
         for (int i = 0; i < 10; ++i)
         {
            al.Add(i);
         }
         al.Insert(3, 10000);
         foreach (int x in al)
            Console.WriteLine(x);
      }

   }
}
```

ArrayList sınıfın removeAt fonksiyonu diziyi sıkıştırarak belirli bir index teki elemanı siler.

```
public virtual void RemoveAt(
      int index
)

using System;

using System.Threading;
using System.IO;
using System.Collections;
```

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         ArrayList al = new ArrayList();
         for (int i = 0; i < 10; ++i)
         {
            al.Add(i);
         }
         al.RemoveAt(5);
         foreach (int x in al)
            Console.WriteLine(x);
      }

   }
}
```

Türlere İlişkin Type Nesneleri: .net te her yapı, sınıf, enum ve arayüz(interface) türleri için bir type nesnesi yaratılmaktadır. Type nesnesinin içinde türe ilişkin meta data bilgileri vardır. Type sınıfı System isim alanı içinde bulunmaktadır.

.net te çeşitli durumlarda bizden Type nesnesini gösteren referans istenmektedir. Bir türün type referansını elde etmenin üç yolu vardır.

1. typeof operatörü ile. typeof operatörü şöyle kullanılır. *typeof<sınıf ismi> Örneğin*
   Type t;
   t = typeof(Sample);
2. Object sınıfının GetType fonksiyonunu kullanarak. **Örneğin:**
   Sample s;
   Type t;

   t = s.GetType();
   GetType fonksiyonu referansın dinamik türüne ilişkin sınıfın Type nesnesini vermektedir.
   Örneğin:
   object o = new Sample();
   Type t;

   t = o.GetType();
   Örneğimizde Sample sınıfının Type nesnesi elde edilmektedir.

3. Bir Assambley içindeki tüm Type nesneleri Assambly sınıfından haraketle elde edilebilir. Bu yöntem burada ele alınmayacaktır.

Type sınıfının Name property elemanı ilgili sınıfın ismini FullName isimli property elemanı ilgili türün nitelikli ismini vermektedir.

System.Array Sınıfı: Anımsanacağı gibi Array sınıfı tüm diziler için taban sınıf durumundadır. O halde biz her türden dizinin System.Array türünden bir referansa atayabiliriz. **Örneğin:**
int[] x = new int[10];
Array a;

a = x;

istersek geri dönüşüm de uygulayabiliriz.
int[] y;
y = (int[])a;

Array sınıfının bazı statik fonksiyonları oldukça faydalı işlemler yapar:

- Copy fonksiyonları aynı türden iki diziyi birbirine kopyalamak için kullanılır.
  public static void Copy(
  Array *sourceArray,*
  Array *destinationArray,*
  int *length*
)
- Reverse fonksiyonu diziyi tersyüz eder.
  public static void Reverse(
  Array *array*
)

- Sort fonksiyonu diziyi sıraya dizer

Çok Biçimlilik(Polymorfism):

Bir dilin nesne yönelimli olabilmesi için çok biçimlilik özelliğinin bulunması gerekir. Çok biçimlilik üç farklı bakış açısıyla tanımlanabilir.

1. **Biyolojik Tanım:** Tabansınıfın bir fonksiyonunu türemiş sınıfın kendine özgü bir biçimde gerçekleştirmesidir.
2. **Aşağı Seviyeli Tanım:** Çok biçimlilik önceden yazılmış kodların sonradan yazılmış kodları çağırabilmesi özelliğidir.
3. **Yazılım Mühendisliği Tanımı:** Çok biçimlilik türden bağımsız kod parçalarının oluşturulması için bir yöntem.

C# da çok biçimlilik sanal fonksiyonlarla gerçekleştirilir.
Bir sınıfın statik olmayan fonksiyonu virtual anahtar sözcüğü getirilerek sanal bir fonksiyon yapılabilr.

```
class A
{
        //...
        public virtual void Func() //virtual ve public yer değiştirebilir. virtual public
        {
                //...
        }
}
```

virtual anahtar sözcüğü ileerişim belirleyici anahtar sözcük aynı sintaks grubu içinde olduğu için yer değiştirmeli olarak yazılabilir.

Taban sınıftaki bir sanal fonksiyon türemiş sınıfta aynı isim aynı geri dönüş değeri türü ve aynı parametrik yapıyla bidirilirse fakat virtual yerine owerride anahtar sözcüğü kullanılırsa taban sınıftaki sanal fonksiyonun türemiş sınıfta override edilmesi denir. (Şüphesiz parametre değişkeni isiminin bir önemi yoktur.)
Örneğin:

```
class A
{
	//...
	public virtual int Func(int x) //virtual ve public yer değiştirebilir. virtual public
	{
		//...
	}
}
class B:A
{
	//...
	public override int Func(int a) /
	{
		//...
	}
}
```

override etme sırasındaaynı erişim belirleyicisi kullanılmalıdır. Override etme işlemi sırasında şunlar önemlidir.

- Taban sınıfta sanal olmayan bir fonksiyon türemiş sınıfta override edilemez. Yani ancak sanal fonksiyonlar override edilebilir.
- Override etme sırasında fonksiyonun yalnızca imzası değil geri dönüş değeri de uymak zorundadır.
- Fonksiyonlar aynı erişim belirleyicisi ile override edilirler.

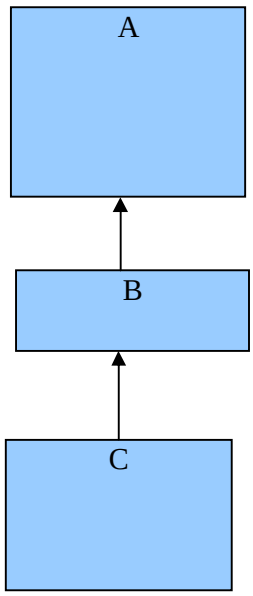
Sanal fonksiyon kavramı yalnızca virtual fonksiyonlar için değil **virtual, override** ve **abstract** fonksiyonlar içinde kullanılır. Yani sanal fonksiyon denildiğinde virtual override ve abstract fonksiyonlar anlaşılmalıdır.

Türemiş sınıfta override edilen fonksiyon yeniden türemiş sınıfta override edilebilir. Yani virtual anahtar sözcüğü sanallığı başlatmak için override anahtar sözcükleri devam ettirmek için kullanılır. Örneğin:

```
class A
{
	//...
	public virtual int Func(int x)
	{
		//...
	}
}
class B:A
{
	//...
	public override int Func(int a) /
	{
		//...
	}
}
class C:B
{
	//...
	public override int Func(int z) /
```

```
        {
                //...
        }
}
```

Taban sınıftaki sanal fonksiyon türemiş sınıfta override edilmek zorunda değil. Örneğin türemiş sınıfta override edilmediği halde türemiş sınıftan türetilen sınıfta override edilebilir.




```
class A
{
        //...
        public virtual int Func(int x)
        {
        //...
        }
}
//...
class C:B
{
        //...
        public override int Func(int z) /
        {
                //...
        }
}
```

Bir referransla bir fonksiyon çağrılmış olsun. Fonksiyon referansın statik türüne ilişki sınıfın faaliyet alanında aranır. Bulunursa fonksiyonun sanal olup olmadığına bakılır. Eğer fonksiyon sanal değilse bulunan fonksiyon çağrılır. Sanalsa referansın dinamik türüne ilşikin sınıfın override edilmiş sanal fonksiyonu çağrılır.

Örnek:

```
using System;
using System.Collections;

namespace CSD
{
   class App
   {
      public static void Main()
      {
         B b = new B();
         A a;

         a = b;

         a.Func();

      }
   }
   class A
   {
      public virtual void Func()
```

```
        {
            Console.WriteLine("A.Func");
        }
    }
    class B:A
    {
        public override void Func()
        {
            Console.WriteLine("B.Func");
        }

    }
}

using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new C();
            A a = b;

           b.Func();

        }
    }
    class A
    {
        public virtual void Func()
        {
            Console.WriteLine("A.Func");
        }
    }
    class B:A
    {
        public override void Func()
        {
            Console.WriteLine("B.Func");
        }

    }
    class C:B
    {
        public override void Func()
        {
            Console.WriteLine("C.Func");
        }
```

```
    }
}

using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new C();

            b.Func();

        }
    }
    class A
    {
        public virtual void Func()
        {
            Console.WriteLine("A.Func");
        }
    }
    class B:A
    {


    }
    class C:B
    {
        public override void Func()
        {
            Console.WriteLine("C.Func");
        }

    }
}

using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Foo(new C());
            Foo(new B());
            Foo(new A());
```

```
        }
        public static void Foo(A a)
        {
            a.Func();
        }
    }
        class A
        {
            public virtual void Func()
            {
                Console.WriteLine("A.Func");
            }
        }
        class B : A
        {
            public override void Func()
            {
                Console.WriteLine("B.Func");
            }

        }
        class C : B
        {
            public override void Func()
            {
                Console.WriteLine("C.Func");
            }

        }
}
```

Sanal Fonksiyonun Çağrılmasına ilşkin Senaryolar:
1. Türemiş sınıf türünden bir referans taban sınıf türünden bir referansa atanmıştır ve bu taban referansıyla sanal fonksiyon çağrılmıştır. Bu durumda referansın dinamik türüne ilşkin sınıfın sanal fonksiyonu çağrılır
2. Fonksiyonun parametresi taban sınıf türünden bir referanstır. Fonksiyonda türemiş sınıf türünden bir referansla çağrılır. Şimdi biz fonksiyonda bu parametre ile sanal fonksiyon çağırırsak Türemiş sınıfın sanal fonksiyonu çağrılır.
3. Taban sınıfın sanal olmayan bir fonksiyonu taban sınıfın sanal fonksiyonunu doğrudan çağırmaktadır. Türemiş sınıf türünden bir referansla taban sınıfın sanal olmayan bir fonksiyonu çağrılırsa bu sanal olmayan fonksiyonun içinde çağrılan sanal fonksiyon türemiş sınıfın fonksiyonu olur.

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new B();
```

```
            b.Bar();
        }

    }
    class A
    {
        public void Bar()
        {
            //...
            Foo();
        }
        public virtual void Foo()
        {
            Console.WriteLine("A.Foo");
        }
    }
   class B : A
   {
        public override void Foo()
        {
          Console.WriteLine("B.Foo");
        }

    }
}
```

Bar fonksiyonunun içindeki Foo çağırması this.Foo ile eşedeğerdir. Buradakli this referansının statik türü A dinamik türü B olduğu için çağrılan Foo B'nin Foo'su olacaktır. Eğer referansın dinamik türüne ilişkin sınıfta sanal fonksiyon override edilmediyse dinamik türe ilişkin sınıfta yukarıya doğru override edilmiş ilk sınıfın sanal fonksiyonu çağrılır. Nihayet hiçbir sınıfta override edilmiş fonksiyon bulunmadıysa bu durumda taban sınıftaki virtual fonksiyonu çağrılacaktır.

Çok biçimliliğin bir tanımı da önceden yazılmış kodların sonradan yazılmış kodları çağırması durumudur. Örneğin biz bir kaç sene önce aşağıdaki iki sınıfı bir DLL'e yerleştirmiş olabiliriz.

```
public class Sample
    {
        public static void Func(A a)
        {
            //...
            a.Foo();
            //...
        }
    public class A
    {
        public virtual void Foo()
        {
            //...
        }
    }
```

Biz şimdi seneler sonra bu DLL'ereferans edip DLL içindeki A sınıfından bir B sınıfı türetip A daki sanal fonksiyonu override etmiş olalım.

```
class B:A
  {
     public override void Foo()
    {
       //...
    }
  }
```
Kodu şöyle çağıralım

```
public static void Main()
    {
      B b = new B();
      Sample.Func(b);
    }
```

Artık Sample.Func fonksiyonu B sınıfının Foo fonksiyonunu çağırı duruma gelmiştir. Görüldüğü gibi: Eskiden yazılmış Func fonksiyonu sonradan yazılmış Foo fonksiyonunu çağırmaktadır.

Taban sınıf referanslarından oluşan bir dizi ya da collection sınıf oluşturulup biz bu dizi ya da collection içerisine çeşitli türemiş sınıf nesneleri yerleştirebiliriz. Sonra bu collection ı dolaşarak bu sanal fonksiyonu çağırdığımızda dinmaik türe ilişkin türün fonksiyonları çağrılmış olur.

```
using System;
using System.Collections;

namespace CSD
{
  class App
  {
    public static void Main()
    {
      ArrayList al = new ArrayList();
       al.Add(new B());
       al.Add(new A());
       al.Add(new C());
       al.Add(new B());

       foreach (object o in al)
       {
          A a = (A)o;

          a.Func();
       }
    }
  }
  class A
  {
    public virtual void Func()
    {
       Console.WriteLine("A.Func");
    }
```

```
        //...
    }
    class B:A
    {
        public override void Func()
        {
            Console.WriteLine("B.Func");
        }
    }
    class C:B
    {
        public override void Func()
        {
            Console.WriteLine("C.Func");
        }
    }
}
```

Foreach deyimi zilimin her elemanını tür dönüştürme operatörü ile döngü değişkenine atadığı için aynı işlemi kısa bir biçimde şöyle de yapabiliriz.

```
foreach (A a in al)
{
    a.Func();
}
```

Object sınıfınında sanal fonksiyonları vardır. Biz bunları herhangi bir sınıfta ve yapıda override edebiliriz. Object sınıfının sanal ToString fonksiyonunun parametrik yapısı şöyledir.

```
public virtual string ToString()
```

Örnek:

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s =new Sample();
            object o = s;

            string str = o.ToString();

            Console.WriteLine(str);
        }
    }
    class Sample
    {
        public override string ToString()
```

```
    {
        return "This is a Test";
    }
  }
}
```

Object sınıfının ToString fonksiyonu reflection işlemi uygulayarak referansın dinamik türünü belirler ve dinamik türünü belirten yazıya geri döner.

.net içindeki neredeyse tüm sınıf ve yapılardatoString fonksiyonu override edilmiştir. Sınıfların ve yapıların override edilen ToString fonksiyonları tuttukları değere ilişkin anlamlı bir yazı geri dödürürler. Örneğin DateTime sınıfının ToString fonksiyonu yapının tuttuğu tarihin yazısal karşılığı ile geri döner. Int, long, double gibi temel türlere ilişkin yapıların ToString fonksiyonları tutulan değerin yazıkarşılığını vermektedir. Örneğin biz sayıyı yazıya dönüştürmek istersek bu fonksiyondan faydalanabiliriz.

Örneğin:

```
using System;
using System.Collections;

namespace CSD
{
  class App
  {
    public static void Main()
    {
      int a = 123;
      Console.WriteLine("Number: " + a.ToString());
    }
  }
}
```

ToString sanal fonksiyon olduğuna göre aşağıdaki işlemde geçerlidir.

```
using System;
using System.Collections;

namespace CSD
{
  class App
  {
    public static void Main()
    {
      int a = 123;

      object o = a;

      Console.WriteLine("Number: " + a.ToString());
    }
  }
}
```

Örnek:

```
using System;
using System.Collections;

namespace CSD
{
   class App
   {
      public static void Main()
      {
         ArrayList al = new ArrayList();

         al.Add(123);
         al.Add(12.3);
         al.Add(123L);

         foreach (object o in al)
            Console.WriteLine(o.ToString());
      }
   }
}
```

Console sınıfını object parametreli write ve writeLine fonksiyonları da vardır. Bu fonksiyonlar parametre üzerinde ToString fonksiyonunu çağrarak elde edilen yazıyı ekrana yazdırırlar. Bu durumda biz kendi sınıfımıza ilişkin bir referansla write ve writeline fonksiyonları yazdırabiliriz.

```
using System;
using System.Collections;

namespace CSD
{
   class App
   {
      public static void Main()
      {
         Sample s = new Sample();
         Console.WriteLine(s);
      }
      class Sample
      {
         public override string ToString()
         {
            return "test";
         }
      }
   }
}
```

C# standartlarına göre + operatöründe operandlardan biri string türünden ise diğer operanda ToString fonksiyonu uygulanır. Elde edilen iki yazı birleştirilir.

Örneğin:

```
using System;
using System.Collections;
```

```
namespace CSD
{
   class App
   {
      public static void Main()
      {
         int a = 123;
         Console.WriteLine("Number = " + a);
      }
   }
}
```

ToString fonksiyonunun yaptığı işin tam tersini temel türlere ilişkin static pars fonksiyonları yapmaktadır.

```
using System;
using System.Collections;

namespace CSD
{
   class App
   {
      public static void Main()
      {
         string s = "123";
         int val;
         val = int.Parse(s);

         Console.WriteLine(val);
      }
   }
}
```

```
val = int.Parse(Console.ReadLine()); //Klavyeden aldığı değeri parse (readline) integer a çeviriyor.
```

Çok Biçimliliğe Çeşitli Örnekler:
1.)Çok biçimlilik temel işlevleri aynı olan farklı türlere sanki aynı türmüş gibi işlem uygulamaktır. Böylece hem kolay programlama yapılmış olur hem aygılama iyileştirilir hem de kolay eklemeler yapılabilir.
Örneğin bir oyun programında bir top kullanılacak olsun. Fakat oyun programımızı top ta olacak çeşitli değişmelerden bağımsız olarak yazmak isteyelim. Yani düz top, zıplayan top gibi oyun çeşitli toplarla oyananabiliyor olsun fakat biz programımızı top kavramı üzerinde oluşturalım. Bunu için bir Ball sınıfı tasarlanır.

```
class Ball
{
        public virtual void Move()
        {
                //...
        }
```

```
        //...
}
```
Sonra bu Ball sınıfından türetilmiş çeşitli sınıflarda move sanal fonksiyonunu override edelim.
```
class SolidBall:Ball
{
        public override void Move()
        {
                //...
        }
        //...
}
class JumpingBall:Ball
{
        public override void Move()
        {
                //...
        }
        //...
}
```

Şimdi oyunun kendisini de bir sınıfla temsil edelim. Sınıfın start fonksiyonu oyunu oynatsın.

```
class Game
{
        private Ball m_ball;
        //...
        publicGame(Ball ball)
        {
                m_ball = ball;
                //...
        }
        publicvoid Start()
        {
                m_ball .move();
                //...
        }
        //..
}
```

Görüldüğü gibi game sınıfı gamin hareket ettrilimesine göre yaratılmıştır. Fakat hareketin nasıl olacağı duruma gmre değişmektedir. Biz Game nesnesini JumpingBall veya başka bir Ball nesnesi ile yaratabiliriz.
```
public static void Main()
{
        Game g = new Game(new JumpingBall());
        g.Start();
}
```

Biz bu oyuna daha sonra yeni bir topta ekleyebiliriz. Böylece oyunda hiçbir değişiklik yapmadan yalnızca Ball sınıfından yeni bir sınıf türetmekle bunu gerçekleştirebiliriz.

Bu tasarımda en az bir Ball sınıfının olması gerekir. Ball sınıfının kendisi default bir topu belirtebilir. Böylece Ball sınıfından hiçbir sınıf türetmesek bile bu oyunu oynayabiliriz.

2.) Belirli bir kaynakta bulunan metni karakter karekter alarak belirli bir amaç için parse eden bir parser sınıfı yazmak isteyelim. Amacımız Parser sınıfının kaynak ne olursa olsun hepsi için çalışacak biçimde yazılmasıdır. Yani parser sınıfı öyle bir biçimde yazılsınki hiçbir değişiklik yapılmadan bir sitringi de bir dosya içindekileri de, soketten gelen bilgileri de va daha pek çok kaynaktan gelen bilgileri parse edebilsin. Bunun için source isimli bir taban sınıf ve bu sınıftan türetilmiş sınıflar kullanılabilir.

```
class Source
{
   public virtual char GetChar()
   {
      //...
   }
   //...
}
class FileSource : Source
{
   private string m_fileName;

   public FileSource(string fileName)
   {
      m_fileName = fileName;
      //...
   }
   public override char GetChar()
   {
      //...

   }
   //...
}
class StringSource : Source
{
   private string m_text;

   public StringSource(string text)
   {
      m_text = text;
      //...
   }
   public override char GetChar()
   {
      //...
   }
   //...
}
```

Parser sınıfı bir source referansını alarak bu referans üzerinde sanla GetChar fonksiyonunu kullanarak kaynaktan bilgiyi alır.
Parser sınıfı:

```
class Parser
{
   private Sorce m_source;

   public Parser(Source source)
   {
      m_source = source;
   }
   public void Parser()
   {
      //...
      char = m_source.GetChar();
      //...
   }
   //...
}
```

Örneğin biz bir dosyadaki metni parse etmek isteyelim.

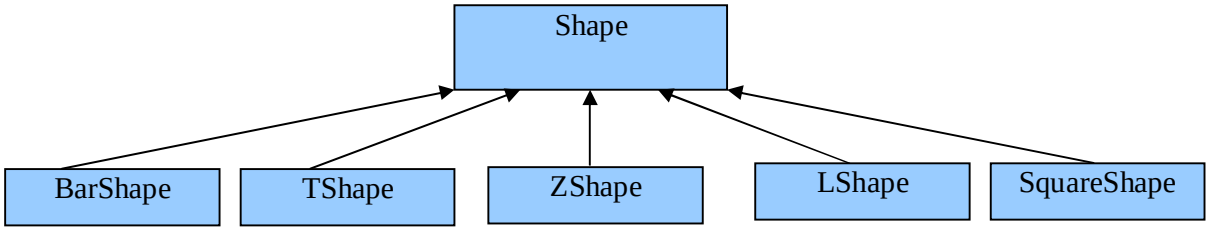
```
   public static void Main()
   {
      FileSource fs = new FileSource("test.txt");
      Parser p = new Parser(fs)

      p.Parser();

   }
```

Burada Parser sınıfının çağıracağı GetChar fonksiyonu FileSource sınıfının GetChar fonksiyonudur. Bu fonksiyon ise Text.txt dosyasından karakter verecektir.
Şimdi Parser sınıfının bir stringden çalışmasını sağlayalım:

```
   public static void Main()
   {
      StringSource ss = new StringSource("this is a test");
      Parser p = new Parser(ss)

      p.Parser();

   }
```

Burada Parser sınıfı StringSource sınıfını kullanmaktadır. Dolayısıyla çağrılacak GetChar fonksiyonu bu sınıfın GetChar fonksiyonudur.
Parser sınıfının başka kaynaklardan çalışması sağlanabilir Tek yapılacak şey Source sınıfından bir sınıf türetip GetChar fonksiyonunu override etmektir.

3.) Bir tetris oyunu yazacak olalım. Oyunun kendisi bir sınıfla temsil edilebilir. Düşen şekiller çok biçimli olarak ifade edilebilir. Düşen şekiller aslında birbirinden farklıdır fakat hepsi düşme, sola ve sağa öteleme ve döndürme özelliğine sahiptir. Tetris oyununu tüm şekilleri sanki tek bir şekil varmış gibi tasarlayabiliriz. Bunun için tabanbir Shape sınıfı alınıp sonra o sınıftan türetilmiş gerçek şekil sınıfları oluşturulur.



Shape sınıfı çeşitli sanal fonksiyonlar dan oluşur.

Shape sınıfınıdan türetilmiş her sınıfta bu fonksiyonlar override edilir. Oyunun knedisi tetris isimli sınıfla temsil edilebilir. Sınıfın Run fonksiyonu oyunu çalıştırsın. Bu durumda oyun şöyle oynatılacaktır.

```
public static void Main()
{
    Tetris tetris = new Tetris();
    tetris.Run();
}
```

Rasgele şekil üreten aşağıdaki gibi bir fonksiyon yazabiliriz.

```
private Shape GetRandomShape()
{
    Shape shape = null;
    Random r = new Random();

    switch ((Shapes)r.Next(5))
    {
        case Shapes.BarShape:
            shape = new BarShape();
            break;
        case Shapes.LShape:
            shape = new LShape();
            break;
        case Shapes.SquareShape:
            shape = new SquareShape();
            break;
        case Shapes.TShape:
            shape = new TShape();
            break;
        case Shapes.ZShape:
            shape = new ZShape();
            break;
    }
    return Shapes;
}
```

Şimdi Run fonksiyonunun çatısını tasarlayalım. Fonksiyon içerisinde rastgele bir şekil düşecek ve o şekil klavyedeki tuşlara göre hareket ettirilecektir.

Run fonksiyonunu çatısı şöyle olabilir.

```
public void Run()
    {
        Shape shape;

        for(; ;)
```

```
        {
            shape = GetRandomShape();

            for(int i = 0; i < 20; ++i)
            {
               shape.MoveDown();
               if (Console.KeyAvailable)
               {
                  switch (Console.ReadKey().Key)
                  {
                     case ConsoleKey.LeftArrow:
                        shape.MoveLeft();
                        break;
                     case ConsoleKey.RightArrow:
                        shape.MoveRight();
                        break;
                     case ConsoleKey.Spacebar:
                        shape.Rotate();
                        break;
                     case ConsoleKey.Q:
                        goto EXIT;
                  }
               }
               System.Threading.Thread.Sleep(500);
            }
         }
         EXIT:
         ;

using System;
using System.Collections;

namespace CSD
{
   class App
   {
      public static void Main()
      {
         Tetris tetris = new Tetris();
         tetris.Run();
      }
   }
   enum Shapes
   {
      BarShape, TShape, ZShape, LShape, SquareShape
   }
   class Tetris
   {
      public void Run()
      {
         Shape shape;
```

```
        for (; ; )
        {
            shape = GetRandomShape();

            for (int i = 0; i < 20; ++i)
            {
                shape.MoveDown();
                if (Console.KeyAvailable)
                {
                    switch (Console.ReadKey().Key)
                    {
                        case ConsoleKey.LeftArrow:
                            shape.MoveLeft();
                            break;
                        case ConsoleKey.RightArrow:
                            shape.MoveRight();
                            break;
                        case ConsoleKey.Spacebar:
                            shape.Rotate();
                            break;
                        case ConsoleKey.Q:
                            goto EXIT;
                    }
                }
                System.Threading.Thread.Sleep(500);
            }
        }
    EXIT:
        ;

}
    private Shape GetRandomShape()
    {
        Shape shape = null;
        Random r = new Random();

        switch ((Shapes)r.Next(5))
        {
            case Shapes.BarShape:
                shape = new BarShape();
                break;
            case Shapes.LShape:
                shape = new LShape();
                break;
            case Shapes.SquareShape:
                shape = new SquareShape();
                break;
            case Shapes.TShape:
                shape = new TShape();
                break;
            case Shapes.ZShape:
                shape = new ZShape();
```

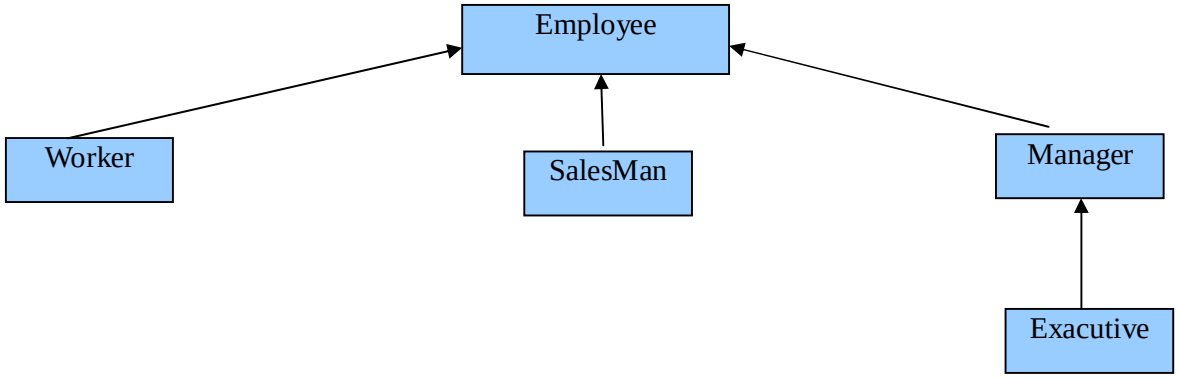
```
            break;
        }
        return shape;
    }
}
class Shape
{
    public virtual void MoveDown()
    { }
    public virtual void MoveLeft()
    { }
    public virtual void MoveRight()
    { }
    public virtual void Rotate()
    { }
}
class BarShape:Shape
{
    public override void MoveDown()
    {
        Console.WriteLine("BarShape.Down");
    }
    public override void MoveLeft()
    {
        Console.WriteLine("BarShape.Left");
    }
    public override void MoveRight()
    {
        Console.WriteLine("BarShape.Right");
    }
    public override void Rotate()
    {
        Console.WriteLine("BarShape.Rotate");
    }
}
class LShape:Shape
{
    public override void MoveDown()
    {
        Console.WriteLine("LShape.Down");
    }
    public override void MoveLeft()
    {
        Console.WriteLine("LShape.Left");
    }
    public override void MoveRight()
    {
        Console.WriteLine("LShape.Right");
    }
    public override void Rotate()
    {
        Console.WriteLine("LShape.Rotate");
```

```
    }
}
class ZShape:Shape
{
    public override void MoveDown()
    {
        Console.WriteLine("ZShape.Down");
    }
    public override void MoveLeft()
    {
        Console.WriteLine("ZShape.Left");
    }
    public override void MoveRight()
    {
        Console.WriteLine("ZShape.Right");
    }
    public override void Rotate()
    {
        Console.WriteLine("ZShape.Rotate");
    }
}
class TShape:Shape
{
    public override void MoveDown()
    {
        Console.WriteLine("TShape.Down");
    }
    public override void MoveLeft()
    {
        Console.WriteLine("TShape.Left");
    }
    public override void MoveRight()
    {
        Console.WriteLine("TShape.Right");
    }
    public override void Rotate()
    {
        Console.WriteLine("TShape.Rotate");
    }
}
class SquareShape:Shape
{
    public override void MoveDown()
    {
        Console.WriteLine("SquareShape.Down");
    }
    public override void MoveLeft()
    {
        Console.WriteLine("SquareShape.Left");
    }
    public override void MoveRight()
    {
```

```
            Console.WriteLine("SquareShape.Right");
        }
        public override void Rotate()
        {
            Console.WriteLine("SquareShape.Rotate");
        }
    }
}
```

Görüldüğü gibi Run fonksiyonu bir şekil kavramını kullanmaktadır. GetRandomShape fonksiyonu hangi shape nesnesini verirse gerçekte o shape sınıfını fonksiyonları çağrılır. Oyuna bir şekil eklemek oldukça kolaydır. Bunun için yeni bir şekil türetiriz. Shape sınıfındaki fonksiyonları override ederiz. Ve GetRandomShape fonksiyonunun bu şekli de hesaba katmasını sağlarız. Örneğin Run fonksiyonunda bir değişiklik yapmayız.

4.)Bir işyerinde çalışanları Employee sınıfından türetilmiş çeşitli sınıflarla temsil edelim.



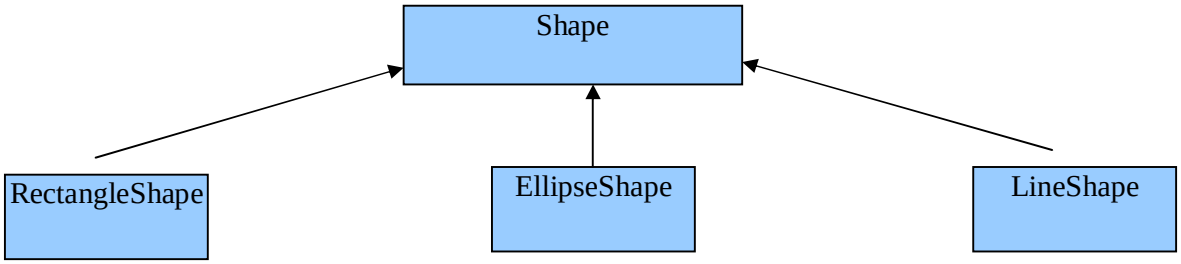
Burada Employee sınıfı tüm çalışanların ortak özelliklerini tutuyor olabilir. Her çalışanın bir maaşı vardır. Fakat Bu maaş herkesde farklı biçimlerde hesaplanıyor olabilir. Employee sınıfının CalcSalary isimli sanal bir fonksiyonu olsun. Bu fonksiyon türemiş sınıfların her birinde o çalışan sınıfının maaşını hesaplayacak biçimde override edilmiş olsun. Şimdi tüm çalışanların bir ArrayList içinde toplandığını düşünelim. Çalışan tüm kişilerin maaş toplamını aşağıdaki gibi bulabiliriz.

```
Double total = 0;
foeeach (Employee e in al)
        total += a.CalcSalary();
```

Burada her al ArrayList nesnesini temsil etmektedir. ArrayList içinde farklı çalışanlar bir arada tutulmaktadır. Çağrılan CalcSalery nesnenin dinamik türü ne ise ona ilikin CalcSalary fonksiyonudur.

5.) PowerPoint benzeri bir program yazacak olalım. Bu programda kabaca bir takım şekiller çizilmekte sonra bu şekillerin içine tıklandığında hangi şekil tıklanmışsa o şekil seçilmektedir. Bu tür programlarda mecburen tüm şekilleri bir collection içinde tutmamız gerekir. Sonra tıklanan noktanın tek tek bu şekillerin içinde olup olmadığına bakmak gerekir. Bir noktanın bir şeklin içinde olup olmadığı o şekle göre değişik bir biçimde test edilir. Her şeklin seçilmesi sırasında küçük kutucukların basılması şekle özgü olarak farklı bir biçimde yapılmaktadır. İşte bu davranışlar çok biçimli davranışlardır.

PowerPoint programında tüm şekilleri aynı şekilmiş gibi işleme sokabiliriz. Bunun için şekiller taban bir shape sınıfından türetilebilir.



Taban Shape sınıfının en azından aşağıdaki gibi iki sanal fonksiyonu bulunmalıdır.

Anahtar Notlar: Uygulamalarda öncelikle çokbiçimli davranışların belirlenmesi gerekir. Eğer birtakım nesnelerin bazı davranışları birbirine benziyorsa fakat onların ana nitelikleri aynı işi yapış fraklılıkları varsa burda çok biçimlilik vardır. Nasıl her hayvanın kulağı duyma gibi temel bir görevi yerine getiriyor fakat az çok birbirlerinden farklıdır.

Ne zaman bir şekil çizilse o şeklin kordinatları kullanılarak shape sınıfından türetilmiş bir nesne yaratılabilir. Sonra bu bütün şekil nesneleri bir arrayList içinde toplanabilir. Shape sınıfının aşağıdaki sanal fonksiyonları olsun.

```
class shape
{
        public virtual bool IsInside(int x, int y)
        {}
        public virtual void Select(graphics g)
        {}
        public virtual void DeSelect(graphics g))
        {}
}
```

Şimdi bütün şekillerin aşağıdaki gibi bir collection nesnesi içinde saklanmış olduğunu düşünelim

```
ArrayList m_shapes;
```

Fareye her tıklandığında biz tek tek tıklanan noktanın bir şekil içinde olup olmadığına bakmalıyız. Windows bize tıklanan yerin kordinatlarını vermektedir. ArrayList içinde çok farklı şekillere ilişkin nesneler bir arada tutulmaktadır. Noktanın hangi şeklin içinde olup olmadığını anlamak şekle göre değişen bir işlemdir. Bu işlem şöyle gerçekletirilebilir.

```
foreach(Shape shape in m_shapes)
        if(shape.IsInside(x, y)
                shape.select(graphics g)
                        break;
```

Görüldüğü gibi tüm şekiller için isInside fonksiyonu çağrılmıştır. Fonksiyon true ile döndüğünde o şekil seçilmiştir. Şeklin seçilmesi yine çok biçimli olarak yapılmıştır.

Abstract Fonksiyonlar ve Sınıflar: Pek çok uygulamada türetme şemasının tepesindeki taban sınıf türden bağımsızlığı temsil etmek için kulanılır. Bu sınıf türünden nesne yaratılmaz. Referans tanımlanarak genellik oluşturulur. Tetris örneğinde tepedeki shape sınıfı powerPoint örneğinde yine tepedeki shape sınıfı buna örnek verilebilir. Böylesi durumlarda tepedeki sınıfların sanal fonksiyonları aslında hiç çağrılmamaktadır. Çünkü bu sınıf türünden bir nesne yaratılmamaktadır.

İşte bu tür durumlarda bu fonksiyonlara boşuna gövde yazmamak için abstract belirleyicisi kullanılır.

Abstract belirleyicisine sahip bir fonksiyon gövde içermez. Fonksiyon bildirimi ";" ile kapatılır. En az bir abstract elemana sahip olan sınıf için abstract anahtar sözcüğü sınıf bildiriminin önüne getrilirek vurgulama yapılmak zorundadır.
Örneğin:

```
abstract class Shape
{
        public abstract void MoveDown();
        public abstract void MoveLeft();
}
```

abstract bir sınıf türünden referanslar tanımlanabilir fakat new operatörüyle nesneler yaratılamaz.

```
shape s;
s = new shape(); //error
```

Abstract sınıftan türetilen bir sınıf taban abstract sınıftaki tüm abstract elemanları override etmelidir. Aksi taktirde türemiş sınıfta abstract olur. Türemiş sınıf bildiriminin başına abstract yazmak gerekir. Tabi bu durumda türemiş sınıf türünden de new operaötrü ile nesne yaratılamaz.

Abstract anahtar sözcüğü virtual anahtar sözcüünüde kapsamaktadır. Abstract, virtual ve override belirleyicileri bir arada kullanılamaz. Bunlardan yalnızca biri kullanılabilir.

```
using System;
using System.Collections;

namespace CSD
{
   class App
   {
      public static void Main()
      {
         Tetris tetris = new Tetris();
         tetris.Run();
      }
   }
   enum Shapes
   {
      BarShape, TShape, ZShape, LShape, SquareShape
   }
   class Tetris
   {
      public void Run()
      {
         Shape shape;

         for (; ; )
         {
            shape = GetRandomShape();

            for (int i = 0; i < 20; ++i)
            {
               shape.MoveDown();
               if (Console.KeyAvailable)
```

```
            {
                switch (Console.ReadKey().Key)
                {
                    case ConsoleKey.LeftArrow:
                        shape.MoveLeft();
                        break;
                    case ConsoleKey.RightArrow:
                        shape.MoveRight();
                        break;
                    case ConsoleKey.Spacebar:
                        shape.Rotate();
                        break;
                    case ConsoleKey.Q:
                        goto EXIT;
                }
            }
            System.Threading.Thread.Sleep(500);
        }
    }
    EXIT:
        ;

}
    private Shape GetRandomShape()
    {
        Shape shape = null;
        Random r = new Random();

        switch ((Shapes)r.Next(5))
        {
            case Shapes.BarShape:
                shape = new BarShape();
                break;
            case Shapes.LShape:
                shape = new LShape();
                break;
            case Shapes.SquareShape:
                shape = new SquareShape();
                break;
            case Shapes.TShape:
                shape = new TShape();
                break;
            case Shapes.ZShape:
                shape = new ZShape();
                break;
        }
        return shape;
    }
}
abstract class Shape
{
    public abstract void MoveDown();
    public abstract void MoveLeft();
    public abstract void MoveRight();
    public abstract void Rotate();
}
class BarShape:Shape
```

```
{
    public override void MoveDown()
    {
        Console.WriteLine("BarShape.Down");
    }
    public override void MoveLeft()
    {
        Console.WriteLine("BarShape.Left");
    }
    public override void MoveRight()
    {
        Console.WriteLine("BarShape.Right");
    }
    public override void Rotate()
    {
        Console.WriteLine("BarShape.Rotate");
    }
}
class LShape:Shape
{
    public override void MoveDown()
    {
        Console.WriteLine("LShape.Down");
    }
    public override void MoveLeft()
    {
        Console.WriteLine("LShape.Left");
    }
    public override void MoveRight()
    {
        Console.WriteLine("LShape.Right");
    }
    public override void Rotate()
    {
        Console.WriteLine("LShape.Rotate");
    }
}
class ZShape:Shape
{
    public override void MoveDown()
    {
        Console.WriteLine("ZShape.Down");
    }
    public override void MoveLeft()
    {
        Console.WriteLine("ZShape.Left");
    }
    public override void MoveRight()
    {
        Console.WriteLine("ZShape.Right");
    }
    public override void Rotate()
    {
        Console.WriteLine("ZShape.Rotate");
    }
}
class TShape:Shape
```

```
    {
        public override void MoveDown()
        {
            Console.WriteLine("TShape.Down");
        }
        public override void MoveLeft()
        {
            Console.WriteLine("TShape.Left");
        }
        public override void MoveRight()
        {
            Console.WriteLine("TShape.Right");
        }
        public override void Rotate()
        {
            Console.WriteLine("TShape.Rotate");
        }
    }
    class SquareShape:Shape
    {
        public override void MoveDown()
        {
            Console.WriteLine("SquareShape.Down");
        }
        public override void MoveLeft()
        {
            Console.WriteLine("SquareShape.Left");
        }
        public override void MoveRight()
        {
            Console.WriteLine("SquareShape.Right");
        }
        public override void Rotate()
        {
            Console.WriteLine("SquareShape.Rotate");
        }
    }
}
```

Abstract bir sınıf anstract olmayan elemanlara veri elemanlarınave normal fonksiyonlara sahip olabilir. Bir sınıfın abstract olabilmesi için en az bir elemanının abstract olması yeterlidir. Fakat bir sınıf hiç abstract elemana sahip olmadığı halde yine de sınıf bildiriminin başına abstract anahtar sözcüğü ekleyerek abstract duruma getirilebilir. Bu durumda yine osınıf türünden nesne yaratılamaz.

Mademki yapılar türetilememektedir. O halde abstract bir yapı sözkonusu olamaz.

Sealed Sınıflar: Sealed belirleyicisi abstract belirleyicisinin adeta zıt anlamlısıdır. Bir sınıf sealed belirleyicisi ile bildirilmişse o sınıftan türetme yapılamaz.

Örnek:

```
sealed class Sample
{
        //...
}
```

“abstract türetme yapılmadıktan sonra işe yaramaz anlamına gelirken sealed bu sınıftan türetme yapmak anlamsızdır anlamına gelir.”

Sealed Override Fonksiyonlar: Bir override fonksiyon aynı zamanda sealed yapılabilir. Fakat abstract veya virtual fonksiyon sealed yapılamaz. Sealed override bir fonksiyon o sınıftan bir türetme yapıldığında artık daha fazla override edilemez.

Örneğin:

```
class A
{
        public virtual void Foo()
        {
                //...
        }
        //...
}
class B:A
{
        public sealed override void Foo()
        {
                //...
        }
        //...
}
```

Burada B sınıfından türetme yapılabilir. Fakat Foo fonksiyonu override edilemez.

Static Sınıflar: Statik sınıflar Framework 2.0 (C# language spasification 3.0) ile eklenmiştir. Static sınıflar statik olmayan elemanlara sahip olamaz. Dolayısıyla static sınıf türünden nesneler ve referanslar da yaratılamaz.

Örneğin:

```
static class Sample
{
        public static void Foo()
        {
                //...
        }
        //...
}
```

Örneğin Math sınıfı statik bir sınıftır. Statik sınıf kavramı okunabilirliği arttırmaktadır. Bir sınıfın static olduğunu gördüğümüzde o sınıftan nesne ve referans yaratmanın anlamı olmadığını sınıfın sadece statik elemanlara sahip olduğunu anlarız. Static sınıflardan türetme de yapılamaz.

Sanal ve Abstract Property Elemanları: Property lerde virtual olabilir ve türemiş sınıflarda override edilebilir. Çünkü propertyler aslında get ve set fonksiyonlarıdır.

Örneğin:

```
class A
{
        private int m_val;

        public virtual int val
        {
                get
                {
                        //...
                        return m_val;
```

```
		}
		set
		{
			//...
			m_val=value;
		}
		//...
	}
}

class B:A
{
	private int m_val2;

	public override int val
	{
		get
		{
			//...
			return m_val2;
		}
		set
		{
			//...
			m_val2=value;
		}
		//...
	}
}
public stativ void Main()
{
	//...
	A.a = new B();
	a.Val=10; //B'nin property sinin set bölümü çağrılacak.
}
```

Bir property abstract olabilir bu durumda get ve set bölümleri ";" ile kapatılmak zorundadır.

```
abstract class A
{
	public abstract int val
	{
	get;
	set;
	}
	//...
}
```

Şüphesiz abstract property read only, write only ya da read write olabilir.

Eğer taban sınıftaki virtual propert get ve set bölümünü içeriyorsa türemiş sınıfta bunlardan yalnızca biri override edilebilir. Fakat taban sınıfıta property abstract sa bildirimde belirtilen bölümleri hepsi türemiş sınıfta override edilmelidir. Aksi taktirde tgüremiş sınıfda abstract olur.

Birden Fazla Kaynak Kodla Derleme Yapmak: CSC derleyicisiyle komut satırından derleme yaparken birden fazla kaynak dosya belirtilebilir. Bu durumda bu dosyalar derlenerek tek bir exe ya da dll oluşturulur.
Örneğin:
csc a.cs b.cs Enter Bu durumda .exe hedef dosyanın ismi eğer exe derlemesi yapılıyorsa main fonksiyonunun bulunduğu dosyanın ismi dll derlemesi yapılıyorsa birinci kaynak dosyanın ismi olacaktır. Yani a.exe(main içindeyse) Fakat exe ya da dll dosyasının ismi */out:<dosya isimi>* ile değiştirilebilir
csc /out:test.exe a.cs b.cs ***ya da örneğin***
csc /target:library /out:test.dll a.cs b.s Enter

IDE de birden fazla kaynak dosya projeye eklenirse IDE bu kaynak dosyaları yukarıda açıklandığı gibi csc derleyicisi ile tek hamlede derler. Bu durumda hedef dosyanın ismi proje ismi ile aynı olur.

Projedeki kaynak dosyalar aynı assambly nin parçaları olur. Dolayısıyla bir kaynak dosyadaki sınıf diğer kaynak dosyadan doğrudan kullanılabilir. Fakat using direktifleri dosyaya özgüdür. Yani bir kaynak dosya diğer kaynak dosyanın using direktiflerini görmez.

Partial Sınıflar: Normal olarak bir sınıf tek bir defada ve tek bir kaynak dosyada bildirilmek zorundadır. Fakat framework 2.0 ile birlikte bir sınıfın parçalı bir biçimde ayrık olarak bildirilebilmesi sağlanmıştır. Eğer bir sınıf partial anahtar sözcüğü ile bildirilirse aynı isim alanında bu sınıf yeniden partial anahtar sözcüğü ile bildirildiğinde birleştirme anlamına gelir.
Örneğin:

```
using System;
using System.Collections;

namespace CSD
{
   class App
   {
      public static void Main()
      {

      }
      partial class Sample
      {
         public void Foo()
         {
            //...
         }
      }
      partial class Sample
      {
         public void Bar()
         {
            //...
         }
      }
   }
}
```

Bir sınıf partial olarak farklı kaynak dosyalarda da bildirilebilir. Zaten partial sınıf kavramı bunu sağlamak için düşünülmüştür. Örneğin Visual Studio 2005 IDE sinde ide nin yazdığı kodlar ile

programcının aynı sınıf için yazdığı kodlar partial sınıf bildirim sayesinde farklı dosyalara bölünebilmiştir.

Sınıf partial olarak bildirilecekse her parçasında partial anahtar sözcüğü belirtilmelidir. Bir sınıf partial anahtarsözcüğü ile bildirildi diye onun birden fazla parçadan oluşması zorunlu değildir. Partial sınıf bildirimlerinde taban sınıf yalnızca bir tek partial bildirimde belirtilir. Bu bildirihangi parçadan yapıldığının bir önemi yoktur.

Exception İşlemleri: Exception terimi yazılımda umulmadık bir biçimde ortaya çıkmış olan problemli durumları anlatmaktadır. Exception mekanizması pek çok nesne yönelimli programlama dilinde benzer biçimde bulunmaktadır.

C# da exeption mekanizması try, catch, finall, throw anahtar sözcükleriyle gerçekleştirilir.

try anahtar sözcüğünden sonra blok açılmak zorundadır. Buna try bloğun denir. try bloğunu bir yada birden fazla catch bölümü ya da finally bölümü takip etmelidir. try bölümü tek başına bulunamaz catch anahtar sözcüğünden sonra parantezler içinde catch paramtre bildirimi yapılır. catch parametresi birden fazla olamaz. catch parentezinden sonra catch bölümü bir blok içermek zorundadır. Bunlara catch blokları denir. Tipik olarak try ve catch blokları şöyle oluşturulur.

```
try
{
        //...
}
catch(<tür> [deyim isimi])
{
        //...
}
catch(<tür> [deyim isimi])
{
        //...
}
```

try bloğu ile catch bloğu arasına ve catch blokları arasına herhangi bir deyim yerleştirilemez. Ayrıca catch parametresinin türü yazılmak zorundadır fakat parametre ismi yazılmak zorunda değildir. catch parametresi System.Exception isimli sınıf türünden ya da bu sınıftan türetilmiş bir sınıf ismi türünden olmak zorundadır. System.Exception sınıfı ya da bu sınıftan türetilmiş sınıflara Exception sınıfları denir.

Exception işlemini tetikleyen anahtar sözcük throw anahtar sözcüğüdür.

Throw işleminin genel biçimi şöyledir
throw[ifade]
throw ifadesindeki ifade System.Exception sınıfı türünden ya da bu sınıftan türemiş bir sınıf türünden olmak zorundadır.

Programın akışı throw deyimini gördüğünde akış sanki bir goto işlemi gibi son girilen try bloğunun uygun catch bloğuna aktarılır. Akış oradan bir daha geri dönmez tüm catch blokları atlanarak akış catch bloklarının sonundaki ilk deyimle devam eder.
Throw işlemiyle akış throw ifadesiyle aynı türden parametreye sahip catch bloğuna aktarılmaktadır. Catch blokları adeta break li case bölümleri gibidir. Eğer akış try bloğuna girdikten sonra hiçbir throw işlemi oluşmazsa akış tüm catch blokları atlanarak catch bloklarından sonraki ilk deyimle devam eder.

Throw işlemiyle bir exception oluştuğunda eğer exception hiçbir catch bloğu tarafında yakalanmazsa ya da akış hiç bir try bloğuna girmemişse bu durumda CLR exceptionun oluştuğu thread i sonlandırır. Tek tread li uygulamalarda bu durum tüm programın sonlanacağı anlamına gelir.

Örnek:

```
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         try
         {
            Foo(-2);
            Console.WriteLine("...");
         }
         catch (MyException e)
         {
            Console.WriteLine("MyException");
         }
         catch (YourException e)
         {
            Console.WriteLine("YourException");
         }

         Console.WriteLine("Main ends...");


      }

      public static void Foo(int a)
      {
         Console.WriteLine("Foo begins...");

         if (a < 0)
            throw new MyException();

         Console.WriteLine("Foo ends...");

      }
   }


   class MyException : Exception
   {
      //...
   }

   class YourException : Exception
   {
      //...
   }

}
```

Eğer throw işlemini takip eden uygun bir catch bloğuda(aynı parametreli) yoksa program yine çöker.

Örnek:

```
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         try
         {
            Foo(-2);
            Console.WriteLine("...");
         }

         catch (YourException e)
         {
            Console.WriteLine("YourException");
         }

         Console.WriteLine("Main ends...");


      }

      public static void Foo(int a)
      {
         Console.WriteLine("Foo begins...");

         if (a < 0)
            throw new MyException();

         Console.WriteLine("Foo ends...");

      }
   }


   class MyException : Exception
   {
      //...
   }

   class YourException : Exception
   {
      //...
   }

}
```

Kütüphane içinde pek çok exception sınıfı da vardır. Kütüphane içindeki bazı fonksiyonlar çeşitli olumsuzlıklarda yine kütüphane içindeki bazı excepiton sınıflarıyla throw etmektedir.

Bir kütüphane fonksiyonu çağrılırken o kütüphane fonksiyonunun dökümantasyonuna bakarak onun hangi türlerle hangi durumlarla throw edilebilecği göz önüne alınmalıdır. Fonksiyon programın çökmesi istenmiyorsa try catch içinde çağrılmalıdır. MSDN dökümanlarında her fonksiyon için "Exception" başlığı altında o fonksiyonun hangi türlerle hangi durumlar da throw ewdileceği belirlenmiştir.

Örnek:

```
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         try
         {
            int.Parse(Console.ReadLine());
         }
         catch (FormatException e)
         {
            Console.WriteLine("Sayinin formati bozuk");
         }
         catch (OverflowException e)
         {
            Console.WriteLine("Sayi çok büyük ya da çok küçük");
         }

         Console.WriteLine("Main ends...");


      }

   }

   class MyException : Exception
   {
      //...
   }
   class YourException : Exception
   {
      //...
   }

}
```

Örneğin temel türlere ilişkin Parse fonksiyonları eğer parametreleriyle belirtilen yazı sayısal karakterlerden farklı karakterler içeriyorsa FormatException sınıfıyla eğer yazı sınır taşması durumundaysa OverflowException sınıfıyla exception edilir.

Bir exception oluştuğunda programcı hatayı bildirip işlemi yinelemek isteyebilir. **Örneğin:**

```
public static void Main()
      {
         bool flag = true;

         int val;
```

```
        do
        {

            try
            {
                val = int.Parse(Console.ReadLine());
                flag = false;
            }
            catch (FormatException e)
            {
                //...
            }
            catch (OverflowException e)
            {
                //...
            }

        } while (flag);


    }
```

Program akışı iç içe try bloklarına girmiş olabilir. Örneğin try bloğu içinde biz bir fonksiyon çağırmış olalım o fonksiyon içinde yine try catch işlemi yapılmış olsun. İç try bloğunda bir throw işlemi oluştuğunda CLR önce son girilen try bloklarının catch bloklarını tarar. Eğer orada uygun bir catch bloğu bulunursa akış oraya aktarılır. Bulunumazsa içeriden dışarıya doğru tek tek try bloklarının catch blokları taranır. Eğer sonuna kadar hiçbir catch bloğu Exception u yakalamazsa thread sonlandırılır.
Örneğin:

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         try
         {
            Foo(-2);
         }
         catch (MyException e)
         {

         }
      }

      public static void Foo(int a)
      {
```

```
            try
            {
                Bar(a);
            }

            catch (YourException e)
            {
            }

        }

        public static void Bar(int a)
        {
            if (a < 0)
            {
                throw new HerException();
            }
        }

    }
    class MyException : Exception
    {
        //...
    }

    class YourException : Exception
    {
        //...
    }

    class HerException : Exception
    {
        //...
    }

}
```

Bir fonksiyon pek çok farklı türle throw ediyor olsun biz bu fonksiyonu güvenle çağırabilmek için tüm türlere ilişkin cacht bloğu yerleştirmek zorundamıyız. İşte türemiş sınıf taban sınıfa atanabildiğine göre türemiş sınıf türünden bir throw işlemi ile taban sınıf catch bloklarıyla yakalanabilir. Bu durumda örneğin biz exception parametreli bir catch bloğu ile tüm excfepitonları yakalayabiliriz.
Catch bloklarına sırasıyla bakılmaktadır. Bu nedenle hem taban sınıfa hem de türemiş sınıfa ilişkin bir catch bloğu bir arada bulundurulabilir. Fakat taban sınıfa ilişkin catch bloğunun daha yukarıya yerleştirilmesi saçma olduğu için yasaklanmıştır. Bu tür durumlarda her zaman türemiş sınıfa ilişkin cach blokları daha yukarıya yerleştirilir. Böylece türemiş sınıfa ilişkin exception oluştuğunda bunu türemiş sınıf yakalar geri kalanları taban sınıf catch blokları tarafından yakalanır.

```
using System.IO;
using System;
using System.Collections;

namespace CSD
```

```
{
   sealed class App
   {
      public static void Main()
      {
         int val;

         try
         {
            val = int.Parse(Console.ReadLine());
            Console.WriteLine(val);
         }
         catch (OverflowException e)
         {
            //...
         }
         catch (Exception e)
         {
            //...
         }


      }

   }

   class MyException : Exception
   {
      //...
   }

   class YourException : Exception
   {
      //...
   }

   class HerException : Exception
   {
      //...
   }
}
```

Try bloğundan sonra isteğe bağlı olarak finally bloğu oluşturulabilir. Finally bloğu parametre içermez eğer bulundurulacaksa catch bloklarının sonuna yerleştirilmek zorundadır. Finally bloğu exception oluşsada oluşmasada çalıştırılan bir bloktur. Örneğin try bloğuna girdikten sonra bir excepiton oluşmuş olsun ve exception bir catch bloğu tarafından yakalanmış olsun ilgili catch bloğu çalıştırıldıktan sonra finally bloğu da çalıştırılır ve akış finally bloğunun sonundaki ilk deyimle devam eder. Try bloğuna girdikten sonra hiç true işlemi oluşmamış olsun. Bu durumda catch blokları atlanır fakat finally bloğu yine de çalıştırılır.
using System.IO;

```
using System;
```

```
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         try
         {
            Func();
         }
         catch (OverflowException e)
         {
            //...
         }
         catch (FormatException e)
         {
            //...
         }

         finally
         {
            //...
         }


      }

      public static void Func()
      {
         int val;


         try
         {
            val = int.Parse(Console.ReadLine());
            System.Console.WriteLine(val);
         }
         catch (OverflowException e)
         {
            //...
         }

         finally
         {
            //...
         }

      }

   }
```

Madem ki finally blogu her durumda çalistirilmaktadir o halde bir deyimin finally bloguna yerlestirilmesi ile hiç finally blogu olmadan catch blogunun sonuna yerlestirilmesi arasinda ne fark vardır?

Arada ki fark tipik olarak iç içe try blokları söz konusu oldugunda belirginleşmektedir. İç try blogunda throw olustuğunda bu exception iç try bloğunun catch blokları tarafindan yakalanmasa bile iç try bloğunun finally blogu yine çalıstırılır. Finally bloğunun her zaman çalıstırılması geri alma işlemlerinin kesinlikle yapılabilmesini saglamaktadır.

Örnegin:

```
public static void Func()
{
    FileStream fs = null;

    try
    {
       fs = new FileStream();
       //...
    }

    finally
    {
       if (fs != null)
          fs.Close();
    }

}
```

Bu örnekte FileStream nesnesi yaratıldıgında dosya açılmıstır. Daha sonra dosya üzerinde çesitli islemlerin yapildigini varsayalim. Programci bu islemler sirasinda umulmadik bir Exception olusup akisin baska bir try blogun catch blogu tarafindan yakalanacagini dikkate almistir. Bu durumda akis hangi catch bloguna geçerse geçsin bu geçisten önce finally blogu çalistirilacagi içindosya kapatilmistir.

Yukarıdaki kod tipik bir kalıptır. Geri alma işlemi yani close işlemi koşullu olarak yapılmıştır. Çünkü excepiton fliestream fonksiyonun başlangıç fonksiyonunda da oluşabilir. Bu durumda fs ye henüz değer atanmadığına göre içinde null olacaktır. Eğer finally bloğunda kontrol yapılmamış olsaydı bu özel durumda close null ile çağrılmış olurdu.

Try catch durumlarında derleyici exception oluşma ihtimalini değerlendirmektedir. Yukarıdaki kod aşağıdaki gibi oluşturulsaydı bu durum yukarıda anlatılan olumsuzlukların dışında zaten derleme zamanında errore yol açardı.

```
    Filestream fs;

    try
    {
            fs = new FieStream(...);
            //...
    }
    finally
    {
```

```
            fs.close();
        }
```

Burada derleyici FileStream sınıfının başlangıç fonksiyonu içinde exception oluşabileceği olasılığını dikkate almaktadır.

Görüldüğü gibi finally bloğu garantili geri alma işlemleri için düşünülmüştür.

Parametresiz catch bloğu özel bir catch bloğudur. Parametresiz catch bloğu yerleştirilecekse tüm catch bloklarının sonuna fakat varsa finally bloğundan önce yerleştirilmelidir.

```
        try
        {
           Func();
        }
        catch (OverflowException e)
        {
           //...
        }
        catch (FormatException e)
        {
           //...
        }
        catch
        {
                //...
        }
 ****************************************
        try
        {
           Func();
        }
        catch (OverflowException e)
        {
           //...
        }
        catch (FormatException e)
        {
           //...
        }
        catch
        {
           //...
         }

        finally
        {
           //...
        }
```

Parametresiz catch bloğu tüm exceptionları yakalamaktadır. Yani yukarıdaki catch blokları eğer excepitonları yakalamışsa parametresiz catch bloğu bunu kesinlikle yakalayacaktır. **Peki excepiton parametreli catch bloğu ile parametresiz catch bloğu arasında ne fark vardır?** Throw işleminin excepiton sınıfı yada ondan türetilmiş bir sınıfla yapılma zorunluluğu .net geneline ilişkin bir

sorunluluk değil C# a ilişkin bir zorunluluktur. Yani C++.Net te herhangi türle throw işlemi yapabiliriz. İşte böyle C++ kodunun C# dan çağrıldığını düşünürsek böyle bir exceptionı excepiton parametreli catch bloğuyla yakalayamayız fakat parametresiz catch bloğuyla yakalayabiliriz. Tabi projemizde yalnızca C# kullanacaksak gerçekten bir fark yoktur.

Exception Sınıflarının Anlamı: Bir hata oluştuğunda hatanın nedeni ve hata nedeniyle oluşan çeşitli bilgiler exception nesnesine doldurularak o nesne ile throw yapılır. Böylece exceptionı yakalayacak kişi oluşan excepiton ile oluşan bilgileri bu nesnenin niçinden alabilir. **Örneğin:**

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         try
         {
            Foo(-2);
         }

         catch (MyException e)
         {
            Console.WriteLine(e.Message);
         }
      }

      public static void Foo(int a)
      {
         if (a < 0)
            throw new MyException("Value cannot be negative!..");

      }
   }


   class MyException : Exception
   {
      private string m_msg;
      //...

      public MyException(string msg)
      {
         m_msg = msg;
         //...
      }

      public string Message
      {
         get { return m_msg; }
```

```
        }
    }
}
```

System.Exception Sınıfı: System.Exception sınıfı tüm exception sınıflarının taban sınıfı durumundadır. Bu sınıfın en önemli elemanlarından biri sanal message properties'dir.
Public virtual string Message{get;}
Kütüphane içindeki pek çok exception sınıfı bu property i override etmiştir. Böylece biz tüm exceptionları exception sınıfıyla yakalayıp çok biçimli message property si yoluyla gerçek hata mesajına erişebiliriz. **Örneğin:**

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
    sealed class App
    {
        public static void Main()
        {
            int val;

            try
            {
                val = int.Parse(Console.ReadLine());
                Console.WriteLine(val);
                //...
            }

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

Burada catch parametresi olan e nin dinamik türü aslında başka bir sınıftır ve duruma göre dinamik tür hangi sınıfa ilişkinse onun message property si işleme sokulacaktır.

Ayrıca message sınıfının sanal bir data elemanı vardır. Bu data elemanında excepiton a ilişkin diğer önemli bilgiler bulunur.

Arayüzler(Interface): Arayüzler konusu C# da ve Java da benzer biçimde bulunmaktadır fakat C++ da yoktur. Arayüzler veri elemanına sahip olamayan tüm elemanları abstract fonksiyonlardan oluşan abstract sınıflara benzemektedir. Arayüzler Interface anahtar sözcüğüyle oluşturulur. Arayüz fonksiyonları erişim belirleyici anahtar sözcüğe sahip olamaz. Fakat public erişim belirtecine sahip olduğu varsayılır.
Örneğin:

```
interface Isample
{
        void Foo(int a);
```

```
        int Bar();
}
```
Arayüz isimleri geleneksel olarak I harfiyle başlayarak isimlendirilmektedir. Bir yapı ya da sınıf : dan sonra taban listesinde bir yada birden fazla arayüz ismini geçirebilir. Bu duruma ilgili sınıf yada yapının o arayüzleri desteklemesi(implimente etmesi) denilmektedir. Örneğin:
```
class Test:ISample
{
        //...
}
```
Burada Test sınıfı ISample arayüzünü desteklemektedir. Anımsanacağı gibi yapılar türetilememektedir. Fakat bir yapı bir ya da birden fazla arayüzü destekleyebilir. Bir sınıf sadece tek bir sınıftan türetilebilir. Yani bir sınıfın bir tane taban sınıfı olabilir. Fakat bir sınıf aynı zamannda bir ya da birden fazla arayüzü destekleyebilir. Eğer":" den sonra taban sınıf ismi yazılmamış sadece arayüz yazılmışsa sınıfın yine System.Object sınıfından türetildiği varsayılır. Yani yukarıdaki örnekte Test sınıfı Object sınıfından türetilmiştir.

Eğer bir sınıf için hem taban sınıf hemde bir takım arayüzler taban listesinde belirtilecekse listede önce taban sınıfın belirtilmesi zorunludur. Fakat arayüzler herhangi bir sırada belirtilebilir.
```
interface IX
{
        void Foo();
}
interface IY
{
        int Bar();
}
class A
{
        //...
}

class B: A, IX, IY
{
        //...
} yazabiliriz.
```
Burada önce A sınıfının belirtilmesi zorunludur.

Bir snıf yada yapı bir arayüzü destekliyorsa o arayüzün tüm fonksiyonlarını tanımlamak zorundadır. Bu duruma o fonkisyonları o sınıf ya da yapı da imlimente edilmesi denilmektedir. Implimente edilme işlemi adeta override edilme işlemi gibidir. Fakat override anahtar sözcüğü kullanılamaz. Fonksiyon tanımlanırken public olmak zorundadır. Geri dönüş değeri ismi ve parametrik yapısı arayüzdeki ile aynı olmak zorundadır. Ancak parametre değişkeninin isminin bir önemi yoktur.
Örneğin:
```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
```

```
        {

        }
    }

    interface IX
    {
        void Foo(int a);
    }

    class Sample : IX
    {
        public void Foo()
        {
            //...
        }
        //...
    }
}
```

Bir arayüz türünden referanslar tanımlanabilir fakat new operatörüyle nesneler yaratılamaz. Arayüzler çok biçimliliğe olanak sağlayan türlerdir. Bir sınıf türünden referans ya da yapı türünden değişken o sınıf ya da yapının desteklediği arayüzü referansına doğrudan atanabilir.

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
    sealed class App
    {
        public static void Main()
        {

        }
    }

    interface IX
    {
        void Foo(int a);
    }

    class Sample : IX
    {
        public void Foo()
        {
            //...
        }
        //...
        Sample s = new Sample();
        IX i;
```

```
        i = s; ---> geçerli
    }
}
```

Bir arayüzreferansı ile arayüzün fonkisyonları çağrılırsa referansın dinemik türüne ilişkin sınıfın ya da yapının fonksiyonu çağrılır. Yani arayüzler çok biçimlidir. Arayüz fonksiyonunun sınıf yada yapıda implimente edilmesi override işlemi gibidir.

```
 Sample s = new Sample();
        IX i;

        i = s; ---> geçerli
        i.Foo();

using System.IO;
using System;
using System.Collections;

namespace CSD
{
    sealed class App
    {
        public static void Main()
        {
            IX r;
            Sample s = new Sample();

            r = s;

            r.Foo();
        }
    }

    interface IX
    {
        void Foo();

    }

    class Sample : IX
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
        //...
    }
}
```

Arayüzler Neden Kullanılmaktadır? Arayüzlerin abstract sınıflardan önemli bazı farklılıkları vardır. 1. Bir sınıf bir tek sınıftan türetilebilir fakat birden fazla arayüzü destekleyebilir. Yani C# da çoklu türetme yoktur. Fakat çoklu türetmenin bazı avantajlarından arayüzler yoluyla faydalanılabilir.

2. Bir sınıf yada yapı bir arayüzü destekliyorsa programcı kesinlikle o sınıf yada yapıda o arayüz

fonksiyonlarının bulunacağını bilir. Bu da öneml ibir bilgidir. Aynı zamanda bir garanti oluşturmaktadır.
3. Yapılar çok biçimliliğe kapalı olmakla birlikte arayüzler sayesinde bu amaçla kullanılabilir.
4. Arayüzler arayüz sınıf kavramını daha açık ve daha resmi biçimde oluşturmak için kullanılabilir. Örneğin biz parser isimli sınıfın kaynaktan bağımsız pars işlemi yapmasını daha önce sınıf yoluyla sağladık. Fakat bu tür durumlarda tipik olarak arayüzler kullanılabilir. **Örneğin:**

```
interface Isource
{
        char GetChar();
        //...
}
class Parser : Isource
{
        private Isource m_source;

        public Parser(ISource source)
        {
                m_source = source;
                //...
        }
        public void Parse()
        {
                //...
                ch = m_source.GetChar();
                //...
        }
        //...
}
```

Şimdi biz parser türünden bir nesneyi Isource arayüzünü destekleyen bir sınıf yada yapıyla destekleyebiliriz. Örneğin:

```
class Filesource :I source
{
        //...
}

public static void Main()
{
        //...
        FileSource fs = new FileSource(....)
        Parser p = new Parser(fs);
}
```

Arayüz konusuna ilişkin Ayrıntılar: Eğer taban sınıf bir arayüzü destekliyorsa türemiş sınıf o arayüzü desteklemese bile türemiş sınıfında o arayüzü desteklediği kabul edilir. Yani biz türemiş sınıf türünden bir referansı da taban sınıfın desteklediği arayüz referansına doğrudan atayabiliriz.

```
interface IX
  {
    void Foo();
```

```
    }

    class A : IX
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
        //...
    }
  class B : A
    {
        //...
    }

  public static void Main()
{
            IX r;
            B b = new B();
            r = b;      // geçerli!
            r.Foo();    // A.Foo çagrilir

}
```

Hepsi:

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
    sealed class App
    {
        public static void Main()
        {
            IX r;
            B b = new B();
            r = b;      // geçerli!
            r.Foo();    // A.Foo çagrilir

        }
    }

    interface IX
    {
        void Foo();
    }

    class A : IX
    {
        public void Foo()
        {
```

```
        //...
      }
      //...
   }
   class B : A
   {
      //...
   }

}
```

Burada önemli olan noktalardan biride biz B sınıfında artık Foo yu yine tanımlasak bunu çok biçimliliğe hiçbir etkisi olmaz. Ancak türemiş sınıfta yine taban listesinde belirterek aynı arayüzü bir daha destekleyebilir. Bu durumda çok biçimli etki türemiş sınıfa da yansır. Buna interface reimplimentation denir.
Örneğin:

```
 interface IX
   {
      void Foo();
   }

   class A : IX
   {
      public void Foo()
      {
         //...
      }
      //...
   }
   class B : A, IX
   {
      //...
   }
public static void Main()
      {
         IX r;
         B b = new B();
         r = b;      // geçerli!
         r.Foo();    // B.Foo çagrilir

      }
```

Yukarıdaki Main'e göre B'nin Foo su çağrılır.

Bir arayüz fonksiyonu bir sınıfta implimente edilirken virtual anahtar sözcüğüyle sanallıkta başlatılabilir. Bu surumda bu fonkisyon türemiş sınıflarda override edilebilir. Böylece çok biçimli etki aşağılara yansıtılabilir.
Örneğin:

```
 interface IX
   {
      void Foo();
   }

   class A : IX
```

```
    {
        public virtual void Foo()
        {
            //...
        }
        //...
    }
    class B : A
    {
        public override void Foo()
        {
            //...
        }
        //...
    }
public static void Main()
{
        IX r;
        B b = new B();
        r = b;      // geçerli!
        r.Foo();    // B.Foo çagrilir
}
```

Hatta bir arayüzü bir abstract sınıfta destekeleyebilir. Ve arayüz fonkisyonu o abstrac sınıfta abstract olarak impliment edilebilir.

```
 interface IX
{
      void Foo();
}

abstract class A : IX
{
      public abstract void Foo()
      {
          //...
      }
      //...
}
```

İki farklı arayüzde tamamen aynısimili aynı parametrik yapıaya sahip ve aynı geri dönüş değerine sahip fonksiyonlar bulunabilir. (Bu durum birbirlerini tanımayan şirketlerin arayüzlerinde oluşabilir.) Bir sınıf yada yapı bu arayüzlerin her iksinide desteklediğinde tek bir tanımlamayla iki arayüzdeki fonksiyonuda tek hamlede implimente etmiş olur.

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
```

```
    }
}

interface IX
{
    void Foo();
    //...
}

interface IY
{
    void Foo();
    //...
}

class A : IX, IY
{
    public void Foo()
    {
        //...
    }
    //...
}

}
```

Burada biz A sınıfı türünden bir referansı hem IX türünden bir referansa hem de IY türünden bir referansa doğrudan atayabiliriz. Sonra bu referanslarla Foo fonkisyonunu çağırdığımızda aynı fonksiyon çağrılacaktır.

Bir arayüzün fonksiyonu hem normal olarak hem de açıkça(explicit) implimente edilebilir. Açıkça implimente etmede erişim belirleyicisi yazılmaz. Fonksiyon arayüz ismi ve nokta operatörü ile belirtilir. Açıkça implimente etmenin bazı önemli sonuçları vardır. Örneğin biz açıkça implimente etme yoluyla yukarıdaki örnekte IX ve IY arayüzleri için ayrı ayrı yani iki farklı Foo fonksiyonu implimente edebiliriz.
Örneğin:

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
    sealed class App
    {
        public static void Main()
        {

        }
    }

    interface IX
    {
        void Foo();
```

```
    //...
  }

  interface IY
  {
    void Foo();
    //...
  }

  class A : IX, IY
  {
    void IX.Foo()
    {
      //...
    }

    void IY.Foo()
    {
      //...
    }
    //...
  }

}
```

Arayüz fonksiyon açıkça impilimente edildiyse sınıfı içinde yada o sınıf türünden referans yoluyla fonksiyon çağrıldığında isim araması sırasında açıkça implimente edillmiş olan fonksiyonlar görülmez.

```
 public static void Main()
    {

    }

interface IX
  {
    void Foo();
    //...
  }

  interface A:IX
  {
    void IX.Foo();
    //...
  }
```

Açıkça tanımlanmış fonksiyonlar ancak çok biçimli olarak çağrılabilir. **Örneğin:**

```
public static void Main();
{
        A a =new A();

        a.Foo(); //error!

        IX r;
        r =a;
```

```
        r.Foo // Geçerli Ancak implimente edilmiş olarak çağrılacak
}
```

Bir arayüz fonkisyonu hem normal olarak hemde açıkça implimente edilebilir. Bu durumda sınıf türünden referansla çağırma yapıldığında normal implimente edilmiş olan arayüz referansı ile çok biçimli çağırma yapıldığında açıkça implimente edilmiş olan çağrılır. **Örneğin:**

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         A a = new A();

         a.Foo();       // Normal implimente edilmiş olan çağrılır.

         IX r;
         r = a;
         r.Foo();       // Açıkça implimente edilmiş olan çağrılır.

      }
   }

   interface IX
   {
      void Foo();
      //...
   }

   class A : IX
   {
      void IX.Foo()
      {
         //...
      }

      public void Foo()
      {
         //...
      }
      //...
   }

}
```

Bir arayüzden başka bir arayüz türetilebilir ve burada türetme terimi kullanılır.

```
interface IX
{
        void Foo();
```

```
}
interface IY:IX
{
        void Bar();
}
```

Eğer bir sınıf yada yapı türemiş arayüzü destekliyorsa o sınıf yada yapıda hem türemiş arayüzün hemde taban arayüzün tüm fonksiyonları tanımlanmazk zorundadır. Yani Sanki taban arayüzün fonksiyonları türemiş arayüzün içindeymiş gibi işlem söz konusudur.

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {



      }
   }

   interface IX
   {
      void Foo();
   }

   interface IY : IX
   {
      void Bar();
   }

   class A : IY
   {
      public void Foo()
      {
         //...
      }

      public void Bar()
      {
         //...
      }
   }

}
```

Bazen taban listede hem taban hem de türemiş arayüzünisimleri bulunabilir.

Örneğin:

```
class A: IY, IX
{
        public void Foo()
        {
                //...
        }
        public void Bar()
        {
                //...
        }
        //...
}
```
Burada taban arayüzün ayrıca belirtilmesinin hiçbir özel anlamı ve faydası yoktur. Yalnızca okunabilirliği arttırmak için tercih edilmektedir.

Çok Kullanılan Bazı Arayüzler

Sınıfların Bitiş Fonksiyonları ve IDisposable Arayüzü:C# daçöp toplayıcı nesne seçilebilir duruma geldikten sonra nesneyi silmeden az önce nesne için sınıfın bitiş fonksiyonu denilen bir fonksiyonu çağırır. Bitiş fonksiyonu(destractor) ismi sınıfın ismi ile aynı olan başına ~ getirilmiş bir fonksiyondur. Bitiş fonksiyonu erişim belirleyicisine sahip olamaz. Bunların geri dönüş değeri kavramları da yoktur. **Örneğin:**

```
class Sample
{
        public Sample() //Constructor
        {
                Console.WriteLine("Constractor");
        }
        ~Sample() // destractor
        {
                Console.WriteLine("Destractor");
        }
        //...
}

using System.IO;
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         Sample s = new Sample();

         //...

         s = null;

         Console.WriteLine("ends...");
```

```
    }
  }

  class Sample
  {
    public Sample()     // constructor
    {
      Console.WriteLine("Constructor");
    }

    ~Sample()          // destructor
    {
      Console.WriteLine("detsructor");
    }

    //...
  }


}
```

Aslında çöp toplayıcı object sınıfının sanal finalize fonksiyonunu çağırmaktadır. Derleyici bitiş fonksiyonu yazıldığında onu finalize fonksiyonuna dönüştürmektedir. Yani diğer .net dilerinde bitiş fonksiyonu kavramı yerine finalize kavramı kullanılmaktadır. C# daki bitiş fonksiyonu C++ da olduğu gibi önemli ve kritik değildir. C# daki bitiş fonksiyonu deterministik değildir. Yani çöp toplayıcı tarafından ne zaman çağrılacağı belli değildir. Çünkü çöp toplayıcının nesneyi ne zaman sileceği belli değildir. Gerçi program sonlanınca çöp toplayıcı seçilebilir durumdaki tüm nesneleri silmektedir. Fakat bazı durumlarda özellikle anormal sonlanma durumlarında çöp toplayıcı bu silme işlemini dahi yapamayabilir. Ne zaman yapacağı kesin belli olmayan bir fonksiyona önemli işler yüklemek anlamlı değildir.

Normal olarak destractorlar başlangıç fonksiyonunda tahsis edilen çeşitli kaynakların otomatik geri bırakılmasını sağlamak için düşünülmüştür. Fakat deterministik olmadıkları için bu amaçla kullanılmaları da çok uygun gözükmemektedir. Ayrıca .nette seçilebilir duruma gelmiş nesnelerin silinme sırasıda kesin olarak belli değildir. Bir sınıfın başka bir sınıf türünden referans veri elemanına sahip olma durumunda elemana sahip sınıf nesnesinin seçilebilir duruma gelmesiyle eleman olan referansın gösterdiği nesne de seçilebilir duruma gelir. İşte çöp toplayıcının bu silme işlemini hangi sırada yapacağı belli değildir. Bu durumda biz elemana sahip sınıfın bitiş fonksiyonunda referans veri elemanını kullanamayız çünkü o daha önce silinmiş olabilir. Örneğin:

```
class Sample
  {
    public FileStream m_fs;

    public Sample()
    {
      m_fs = new FileStream(...);
      //...
    }
```

```
    ~Sample()
    {
        m_fs.Close(); //--->Hata
        //...
    }

    //...
  }
```

Tam Örnek:

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
  sealed class App
  {
    public static void Main()
    {
      Sample s = new Sample();

      //...

      s = null;

      Console.WriteLine("ends...");


    }
  }

  class Sample
  {
    private FileStream m_fs;

    public Sample()
    {
      m_fs = new FileStream(....);
      //...
    }

    ~Sample()
    {
      m_fs.Close();      //--> Hata!
      //...
    }
    //...
  }


}
```

Burada açılan dosya nesne çöp toplayıcı tarafından silinirken bitiş fonksiyonunda kapatılmak istenmiştir. Fakat yapılan işlem hatalıdır. Çünkü önce FileStream nesnesi silinmiş olabilir ve m_fs silinmiş bir nesneyi gösteriyor durumda olabilir.

Belirli kaynakların çöp toplayıcının insafına kalmadan belirli noktalarda boşaltılması gerekebilir. İşte Idisposable arayüzü resmi bir prosodür oluşturmak için oluşturulmuştur. System alanındaki Idisposable arayüzü tek bir fonksiyona sahiptir.

```
Interface IDisposable
{
        public void Dispose();
}
```

Görüldüğü gibi bir sınıf yada yapı IDisposable arayüzünü destekliyorsa bu sınıf ya da yapının Dispose isimli fonksiyonu bulunmak zorundadır. İşte dispose fonksiyonu boşaltım yapan bir fonksiyon olarak yazılmaktadır. O halde programcı istediği noktada Dispose fonkisyonunu kendi çağırarak boşaltımı yapmalıdır.

Örneğin:

```
 class Sample : IDisposable
   {
      private FileStream m_fs;

      public Sample()
      {
         m_fs = new FileStream(....);
         //...
      }

     public void Dispose()
      {
         m_fs.Close();
         //...
      }
      //...
   }
```

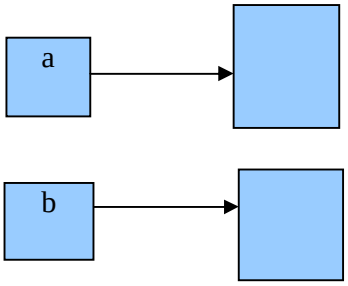
Sınıfı kulanan kişi Dispose fonksiyonunu çağırarak boşaltımı kendisi yapmalıdır.
Örneğin:

```
 public static void Main()
{
        Sample s = new Sample();

        //...

        s.Dispose();
}
```

Bu tür durumlarda programcı Dispose fonksiyonunu çağırmasa ne olur? Enellikle bu tür durumlarda sınıfı tasarlayan kişi ayrıca sınıfın bitiş fonksiyonunda da boşaltım yapmaktadır. Bu nedenle en kötü olasılıkla sınıfın bitiş fonksiyonunda boşaltım gerçekleşecektir. Bunun bazı sorunları olabilir. Fakat hiç boşaltım yapılmamaktansa nesne silinirken boşaltım yapılması daha iyidir. Fakat iyi bir teknik bakımından Idisposable arayüzünü destekleyen sınıf ya da yapılar için programcı kendisi dispose fonksiyonunu çağırmalıdır.

IColoneabla Arayüzü: Icloneable arayüzü Clone isimli tek bir fonksiyonu vardır. Clone fonksiyonu yeni bir nesne yaratarak eski nesnenin özdeş bir kopyasını oluşturur. Clone fonksiyonunun geri dönüş değeri object türündendir fakat aslında dinamik türü kopyası çıkarılacak nesne türündendir.

```
interface Icloneable
{
    public object Clone();
}
```



Bir sınıf ya da yapı için clone fonksiyonunu yazmak zor değildir. Aslındao bir nesneyi klonlamak demek o nesnenin veri elemanlarını değer olarak birebir içeren yen bir nesnenin oluşturulmasıfır.

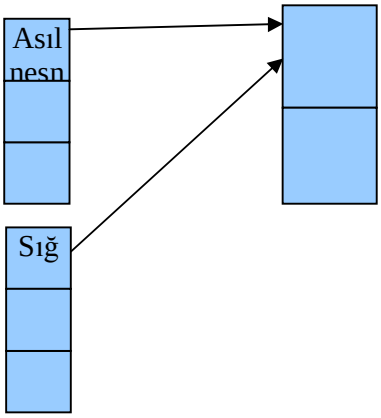
Örneğin:

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         Sample x = new Sample();   //Kullanımıböyle olabilir.

         x.A = 100;

         Sample y;

         y = (Sample)x.Clone();

         Console.WriteLine(y.A);

      }
   }

   class Sample : ICloneable
   {
      private int m_a;

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

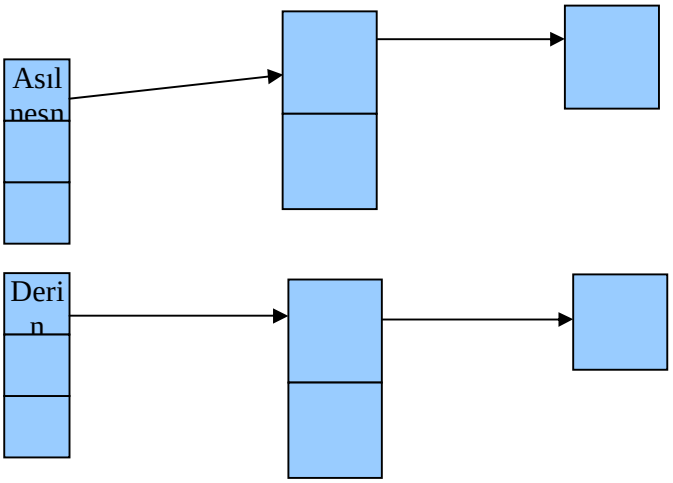
```
        }

        public object Clone()
        {
            Sample s = new Sample();

            s.m_a = m_a;

            return s;
        }
    }
}
```

Klonlama işlemi derin ya da sığ biçimde yapılabilir.

Sığ kopyalama da yalnızca ana nesnenin veri elemanlarının kopyası alınır. Böylece eğer nesnenin referans türünden veri elemanları varsa bunların gösterdikleri yerdeki nesnelerin kopyaları çıkarılmaz.



Halbuki derin kopyalamada sonuna kadar referansların gösterdiği nesnelerin de kopyasından çıkartılır.



Peki derin kopya nasıl çıkartılır? Derin kopyanın çıkartılabilmesi için eleman olarak kullanılan sınıflarından Iclonaebla arayüzünü desteklemeleri gerekir.

Örnek:

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         Sample x = new Sample();

         x.A = 100;

         Sample y;

         y = (Sample)x.Clone();

         Console.WriteLine(y.A);

      }
   }


   class A : ICloneable
   {
      //...
   }

   class B : ICloneable
   {
      //...
   }

   class C : ICloneable
   {
      private A m_a;
      private B m_b;

      public object Clone()
      {
         C c = new C();

         c.m_a = (A)m_a.Clone();
         c.m_b = (B)m_b.Clone();
         //...
         return c;
      }
      //...
   }

}
```

Arayüzlerden Sınıflardan Yapılara Aşağıya Doğru Dönüştürme: Bir sınıf türünden referans ya da bir yapı türünden değişken doğrudan bu sınıfın ya da yapının desteklediği arayüz referanslarına doğrudan atanabilir. Bu durum yukarıya doğru dönüştürmedir. Fakat aynı zamanda taban sınıftan arayüzlere tür dönüştürme operatörleriyle de dönüştürme yapılabilmektedir. Bu işlem derleme zamanında kabul edilir. Fakat çalışma zamanı sırasında referansın dinmaik türünün o arayüzü destekleyip desteklemediğine bakılır. Eğer referansın dinmaik türü ilgili arayüzü desteklemiyorsa invalid cast exception türü excepiton oluşur. Anımsanacağı gibi türemiş sınıfın taban sınıfı bir referansı destekliyorsa türemiş sınıfın da o referansı desteklediği kabul edilir.
Örneğin:

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         A a = new A();
         object o = a;

         IDisposable id;

         id = (IDisposable)o;


      }
   }

   class A
   {
      public void Dispose()
      {
         //...
      }
      //...
   }

}
```

Burada derleme aşamasından başarıyla geçilecektir. Ayrıca runtime da "o" nun dinamik türünün Idisposable arayüzünü destekleyip desteklemediğine bakılacaktır. Örneğimizde A sınıfı Idisposable arayüzünü desteklediğine göre sorun oluşmayacaktır.

Özetle herhangi bir sınıf referansı tür dönştürme operatörüyle herhangi bir türe dönüştürülebilir. Bu durumda derleme aşaması başarıyla tamamlanır. Fakat programın çalışma zamanı sırasında ayrıca referansın dinamik türünün arayüzü destekleyip desteklemediğine ayrıca bakılacaktır. Bu durumda örneğin eğer bir ArrayList içindeki tüm nersnelerin dinmaik türleri Idisposable arayüzünü destekliyorsa biz tek hamlede foreach döngüsüyle tüm nesneleri boşaltabiliriz.

```
ArrayList al = new ArrayList();
//...
```

```
foreach (Idisposable id in al)
    id.Dispose();
```
Bir arayüzden o arayüzün dinamik türüne ilişkin bir sınıfa da dönüştürme yapılabilir.
```
IDisposable id = new A();
A a = (A)id;
```
Ayrıca çalışma zamanı sırasındada arayüz referansının dinamik türünün ilgili arayüzü destekleyip desteklemediği kontrol edilmektedir.

Indeksleyiciler

Sınıfların ve Yapıların Indeksleyici Elemanları: Bir sınıf türünden referansı ya da yapı türünden değişkeni köşeli parantez içinde kullanabilmek için sınıfın yapının indeksleyici denilen elemana sahip olması gerekir. Örneğin ArrayList sınıfı ya da string sınıfı indeksleyici elemanlara sahiptir. Bu nedenle biz onları köşeli parantez operatörü ile kullanabiliriz.

Bir indeksleyivi elemanın genel biçimi şöyledir.

```
[erişim belirleyicisi] < tür> this[<parametre bildirim>]
{
   get
   {
        //...
   }
    set
   {
        //...
   }
}
```
Örneğin:
```
class Sample
{
    //...
    public int this[int index]
    }

                                    get
                                    {
                                          //...
                                    }
                                    set
                                    {
                                          //...
                                    }
     }
}
```
Görüldüğü gibi indeksleyici bildirimi adeta bir property bildirimi gibidir. Property ismi yerine this anahtar sözcüğü gelmiştir. [ ] içinde indeks bildirimi yer almaktadır. Şimdi biz Sample sınıfı türünden bir referansı artık köşeli parantez ile kullanabiliriz.

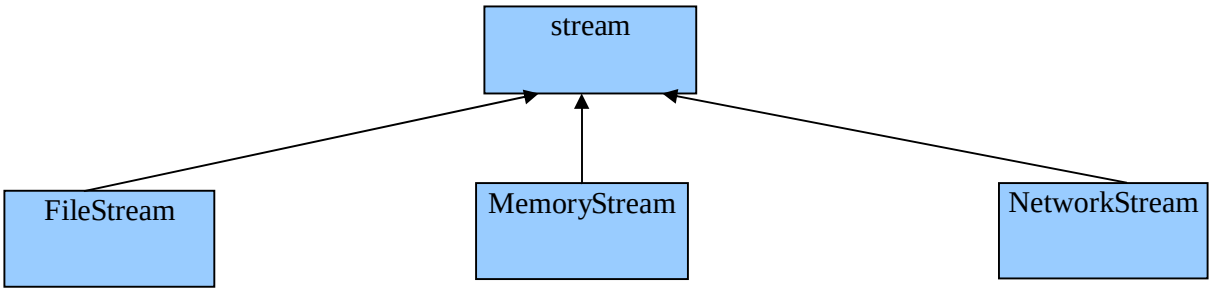
```
S[3] = 10;
```
[ ] içindeki ifade indeks parametresine argüman olarak aktarılmaktadır. Yukarıdaki örnekte indeksleyicinin set bölümü çalıştırılır ve indeks parametredi üç olarak geçirilir. S[] inin türü intir.

Şüphesiz her sınıf için indeksleyici yazmaya gerek yoktur. Eğer sınıf bir collection özelliğine sahipse yani birden fazla nesneyi tutuyorsa indeksleyici anlamlı olabilir. **Örneğin:**

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         MyArray a = new MyArray(10);

         for (int i = 0; i < 10; ++i)
            a[i] = i;



      }
   }

   class MyArray
   {
      private int [] m_array;

      public MyArray(int size)
      {
         m_array = new int[size];
      }

      public int this[int index]
      {
         get
         {
            return m_array[index];
         }

         set
         {
            m_array[index] = value;
         }
      }
      //...
   }
}
```

Dosya İşlemleri: Dosyalar üzerinde okuma/yazma gibi temel işlemleri yapabilmek için .net sınıf sisteminde birbirleriyle ilişkili bir grup sınıf kullanılmaktadır. Bu sınıfların hepsi **System.IO isim** alanı içindedirler. Stream isimli sınıf dosya gibi işlem gören tüm kavramların taban sınıfıdır. Gerçekte bir disk dosyası olmayıp dosya gibi kullanılabilen tüm kavramlar stream sınıfıyla temsil edilmektedir. Stream sınıfı Abstract bir sınıftır. Yani stream sınıfı tek başına kullanılmaz. Stream

Sınıfından türetilmiş bir takım sınıflar vardır.



FileStream sınıfı disk tabanlı dosyaları MemoryStream sınıfı bellek tabanlı dosyaları NetworkStream sınıfı socket sistemini temsil etmektedir.

Bir dosya üzerinde işlem yapabilmek için FileStream sınıfı kullanılmaktadır.
FileStream fs = new FileStream();

Dosyaların Yaratılması ve Açılması: Bir dosya üzerinde işlem yapabilmek için öncelikle dosyanın açılması gerekir. Dosya açmak için FileStream sınıfının başlangıç fonksiyonları kullanılmaktadır.

Filestream sınıfının pek çok başlangıç fonksiyonu vardır. Fakat en çok kullanılanı ve genel olanı aşağıdaki fonksiyondur.
public FileStream(string path, FileMode mode,FileAccess access)
Fonksiyonun birinci parametresi dosyanın yol ifadesidir. Ikinci parametre FileMode isimli bir enum türündendir. Bu parametre açış modunu belirtmektedir. Bu enum türünün elemanları şunlardır.
Open: Bu seçenek olan dosyayı açmak için kullanılır. Eğer birinci parametrede belirtilen dosya yoksa bu durumda FileNotFound exception oluşur.

Create: Bu seçenek dosya yoksa onu yaratır ve açar dosya varsa sıfırlar ve açar.

OpenOrCreate: Bu seçenekte dosya varsa açılır. Yoksa yaratılır ve açılır.

CreateNew: Dosya yoksa yaratılır ve açılır. Fakat dosya varsa işlem başarısız olur. IOExcepiton isimli exception fırlatılır.

Truncate: Bu seçenek olan bir dosyayı sıfırlayıp açmak için kullanılır. Fakat dosya yoksa FileNotFound exception oluşur.

Append: Bu seçenekte dosya yoksa yaratılır ve açılır. Varsa doğrudan açılır(OpenOrCreate gibi) Bu modda dosyaya yapılan her yazma işlemi sona ekleme anlamına gelir.

Fonksiyonun üçüncü parametresi dosyaya hangi işlemin yapılacağını gösterir. FileAccess isimli fonksiyonun 3 adet enum türü vardır.
Read: Dosyadan yalnızca okuma yapılır. Yazma yapılırsa excfeption oluşur.
Write: Dosyadaya yalnızca yazma yapılır okuma yapılırsa excepiton oluşur.
ReadWrite: Dosyadan hem okuma hem yazma yapabiliriz.

FileStream sınıfının string ve FileMode parametreli başlangıç fonksiyonu da vardır. Bu default olarak dosyayı ReadWrite modda açar.
Açılan her dosya işlem bittikten sonra kapatılmalıdır. Kapatma işlemi için stream sınıfının close fonksiyonu kullanılır. Dosya açıldıktan sonra excepiton oluştuğunda otomatik kapatmayı sağlamak

için close işleminin finally bölümünde yapılması tavsiye edilmektedir. O halde try-catch içinde güvenli bir biçimde dosya açabilmek için tipik kalıp aşağıdaki gibidir.

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         FileStream fs = null;

         try
         {
            fs = new FileStream("x.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);
            //...
         }
         catch (Exception e)
         {
            //...
         }
         finally
         {
            if (fs != null)
               fs.Close();
         }
      }
   }
}
```

Dosya Göstericisi Kavramı: Bir dosya aşağı seviyeli ele alındığında bir byte topluluğundan başka bir şey değildir. Dosyanın içinde ne olursa olsun uzantısı ne olursa olsun dosya bizim için byte topluluğundan oluşmaktadır. Dosya içindeki her byte ın ilk byte sıfır olmak üzere bir offset numarası vardır. Dosya göstericisi okuma ve yazma işlemlerinin dosyanın neresinden yapılacağını belirten bir ofset değeridir. Dosya açıldığında dosya göstericisi 0. offsettedir. Okuma ve yazma fonksiyonları dosyanın neresinden okunup yazılacağı konusunda parametre almamaktadır. Çünlü bütün okuma ve yazma işlemleri dosya göstericisinin gösterdiği yerden yapılır. Yani dosya göstericisi cursor görevi görür. Örneğin dosya göstericisinin değeri 100 olsun. Şimdi biz 10 byte okursak ya da 10 byte yazarsak bu işlemler 100. offsetten itibaran yapılacaktır. Okuma ve yazma işlemleri sonlandığında dosya göstericisi otomatik olarak okunan ya da yazılan byte miktarı kadar ilerletilir. Dosya göstrericisinin dosyanın son byte ından sonraki byte ı göstermesi durumuna(ki böyle bir byte yoktur) EOF (end Of File) durumu denir. EOF konumundan okuma yapılamaz. Fakat EOF konumundan yazma yapılabilir. Bu durum dosyaya ekleme yapmak anlamına gelir.

Stream Sınıfının Temel Fonksiyonları:

Dosyadan N byte okumak için stream sınıfının read fonksiyonu kullanılır. Bu fonksiyon absract bir fonksiyondur. Asıl işi FileStream sınıfının fonksiyonu yapmaktadır.

Public abstract int Read(byte[] buffer, int offset, int count)

Fonksiyon dosya göstericisinin gösterdiği yerden count kadar byte birinci parametresiyle belirtilen diziye okur. İkinci parametresi dizide offset belirtmektedir. Fonksiyon okumayı dizinin belirli bir indeksinden itibaren yapabilmektedir.

```
using System.IO;
using System;
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         FileStream fs = null;

         try
         {
            fs = new FileStream("x.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);

            byte[] b = new byte[10];
            //...
            fs.Read(b, 0, 10);

            //...
         }
         catch (Exception e)
         {
            //...
         }
         finally
         {
            if (fs != null)
               fs.Close();
         }
      }
   }
}
```

Fonksiyon dosya göstericisinin gösterdiği yerden itibaren 10 byte ı dizinin sıfırıncı indeksinden itibaren diziye okur. Fonksiyonun geri dönüş değeri okunabilen byte miktarıdır. Fonksiyon talep edilen miktardan daha az sayıda byte ı okuyabilir. Örneğin dosya göstericisinin gösterdiği yerden itibaren dosyanın sonuna kadar 5 byte vardır. Fakat biz 10 byte okumak istemiş olabiliriz. Bu durumda Fonksiyon okuyabildiği kadar byet ı (5 byte) okur ve 5 değeriyle geri döner. Fonksiyonun EOF durumunda olduğundan dolayı hiç okuma yapamazsa exception oluşmaz sıfırla geri döner.

Fonksiyon IO hatasından dolayı okuma yapamazsa IO excepiton fırlatılır.

Stream sınıfının Write isimli fonksiyonu tam tersi bir işlemi yapar bir byte dizisi içinde bulunan değerleri dosya göstericisinin gösterdiği yerden itibaren dosyaya yazar.
public abstract void Write(byte[] buffer, int offset, int count)
Fonksiyonun parametreleri tamamen Read fonksiyonundaki gibidir. Fakat yönü terstir. Yani Fonksiyon dizideki bilgileri dosyaya yazar. IO hatası durumunda yine IO Exception oluşur. Örneğin bir kaynak dosyadan belirli bir miktar okumalar yapıp onu hedef dosyaya yazarak bir dosya kopyalaması yapabiliriz.

```
using System.IO;
using System;
```

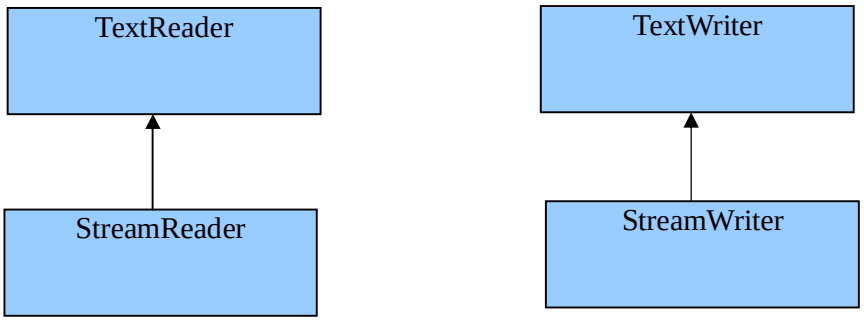
```
using System.Collections;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         FileStream fsSource = null, fsDest=null;

         try
         {
           fsSource = new FileStream(@"E:\CS-SAPER\Sample\Sample.cs", FileMode.Open, FileAccess.Read);
           fsDest = new FileStream(@"E:\CS-SAPER\Sample\CopySample.cs", FileMode.CreateNew, FileAccess.Write;

            int n;
            byte[] buf = new byte[512];

            while ((n = fsSource.Read(buf, 0i 512))>0)
               fsDest.Write(buf, 0, n);

            //...
         }
         catch (Exception e)
         {
           Console.WriteLine(e.Message);
         }
         finally
         {
            if (fsSource != null)
               fsSource.Close();
            if (fsDest != null)
               fsDest.Close();
         }
      }
   }
}
```

Aslında dosya kopylamak için file isimli sınıfın copy fonksiyonu kullanılabilir.

Stream sınıfının ayrıca bir byte okuyup yazan ReadByte ve WriteByte isimli fonksiyonlarıda vardır. Bu fonksiyonlar aslında read ve write fonksiyonları çağrılarak yazılmıştır. Stream ve FileStream sınıflarında bir stream i yazan ya da okuyan int, long gibi sayıları okuyan ya da yazan fonksiyonlar yoktur. Yalnızca byte okuyuıp yazan fonksiyonlar vardır. Fakat aslında her türlü işlemler byte okuyup yazan fonksiyonlarla dolayllı bir biçimde yürütülür. Yazıların byte dizisine dönüştürülüp geri alınması encoding konusu ile ilgilidr. :Örneğin bir byte dizisinin içinde bir yazının bytları nın olduğunu düşünelim. Bu yazı her bir karekteri 2 byte olan Unicode veya 1 byte olan ASCII yazısı olabilir. Ya da örneğin sıkıştırılmış UTF modda da olabilir. O halde bize bir byte dizisi verilse biz o dizi içindeki bytlerın nasıl kodlanarak bir yazı oluşturuluduğunu bilmedikten sonra dönüştürme yapamayız. Fakat int, long gibi türler için bir encoding bilgisi söz konusu değildir.

Temel türleri byte dizisine dönüştüren byte dizisini de temel türlere dönüştüren BitConverter isimli temel bir sınıf vardır.

StreamReader ve StreamWriter Sınıfları: Bu sınıflar bir stream nesnesini alarak onun read ve write fonksiyonlarını çağırmak yoluyla yetenekli text işlemler yapar. StreamReader sınıfı TextReader sınıfından StreamWriter sınıfı TextWriter sınıfından türemiştir.



TextReader ve TextWriter Abstract sınıflardır. sınıfındanbir nesne sınıfın stream parametreli başlangıç fonksiyonuyla yaratılabilir.

public StreamReader(Stream stream)

o halde tipik olarak StreamReader nesnesi şöyle oluşturulabilir.

```
public static void Main()
{
    FileStream fs = null;
    StreamReader = null;

     try
    {
       fs = new FileStream("test.txt", FileMode.Open, FileAccess.Read);
       sr = new StreamReader(fs);
      //...
     }
    finally
    {
       if (sr !=null)
       sr.Close();

      if (fr !=null)
     fr.Close();
}
```

Aslında StreamReader sınıfının close fonksiyonunu kullandığımızda aslında bu close fonksiyonu ttuutuğustream nesnesi için close fonksiyonunu çağırmaktadır. Dolaysıyla yalnızca streamReader için Close da çağrılabilir.

StreamReader sınıfının ReadtoEnd isimli fonksiyonu tekhamlede tüm dosyayı okuyarak bunu bir String nesnesi olarak vermektedir.
public override string ReadToEnd()

```
using System;
using System.IO;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         FileStream fs = null;
         StreamReader = null;

         try
         {
            fs = new FileStream(@"E:\CS-SAPER\Sample\Sample.txt", FileMode.Open, FileAccess.Read);
            sr = new StreamReader(fs);

            string str = sr.ReadToEnd();

            Console.Write(str);

            //...
         }

         catch (Exception e)
         {
            Console.WriteLine(e.Message);
         }
         finally
         {
            if (sr != null)
               sr.Close();

            }
      }
   }
```

StreamReader sınıfının satır satır okuma yapmak için kullanılan ReadLine fonksiyonu da vardır.

```
public override string ReadLine();
```

Eğer dosyanın sonuna gelinmişse null referans elde edilir. O halde bir dosyayı satır satır şöyle okuyabiliriz.

```
using System;
using System.IO;

namespace CSD
{
   sealed class App
   {
      public static void Main()
```

```
    {
        FileStream fs = null;
        StreamReader = null;

        try
        {
            fs = new FileStream(@"E:\CS-SAPER\Sample\Sample.txt", FileMode.Open, FileAccess.Read);
            sr = new StreamReader(fs);

            string s;
            while ((s = sr.ReadLine()) != null)
                Console.WriteLine(s);
            //...
        }

        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        finally
        {
            if (sr != null)
                sr.Close();

        }
    }
}
```

StreamReader sınıfının diğer elemanları MSDN dökümanlarından izlenebilir.

StreamWriter Sınıfı text yazımlar için kolaylık sağlamaktadır. Sınıfın her türden parametreye sahip Write ve WriteLine fonksiyonları dosya göstericisinin gösterdiği yerden itibaren ilgili değerleri yazıya dönüştürüp dosyaya yazmaktadır. Böylece biz önceki örneklerde olduğu gibi sayıları yazıya dönüştürüp onları byte dizisine çevirerek ilkel fonksiyonlarla yazma yapmayız.

```
using System;
using System.IO;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         FileStream fs = null;
         StreamWriter sw = null;

         try
         {
            fs = new FileStream(@"test.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);
            sw = new StreamWriter(fs);
```

```
            string s;
            for (int i = 0; i < 100; ++i)
                sw.WriteLine(i);
            //...
        }

        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        finally
        {
            if (sw != null)
                sw.Close();

        }
    }
  }
}
```

Ayrıca tıpkı Console sınıfında olduğu gibi formatlı yazdırmak da mümkündür.

```
 for (int i = 0; i < 100; ++i)
                sw.WriteLine("Number = {0}", i);
```

```
using System;
using System.IO;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         FileStream fs = null;
         StreamWriter sw = null;

         try
         {
            fs = new FileStream(@"test.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite);
            sw = new StreamWriter(fs);

           for (int i = 0; i < 100; ++i)
                sw.WriteLine("Number = {0}", i);
            //...
         }

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

```
        {
            if (sw != null)
                sw.Close();

        }
    }
  }
}
```

Ayrıca File sınıfının bazı statik fonksiyonları konuyla ilgili bazı faydalı işler yapmaktadır.

- File Sınıfının ReadAlText yol ifadesiyle verilen dosyadaki tüm yazıları okuyarak bir string biçiminde vermektedir. Şüphesi bu fonksiyon kendi içinde aslında yukarıdaki gibi tüm yazıyı almaktadır. *string str = File.ReadAllText("test.txt");*
- Sınıfın WriteAllText fonksiyonu tam ters bir işlem yapmaktadır.
  public static void WriteAllText(string path, string contents)
- File sınıfının OpenText isimli fonksiyonu doğrudan belirtilen dosyayı açarak bize StreamReader nesnesi biçiminde verir.
  *public static StreamReader OpenText(string path)*Böylece önce FileStream i kullanıp daha sonra StreamReader nesnesini elde etme gibi bir işleme gerek yoktur. Örneğin:
  StreamReader sr = File.OpenText(test.txt);
- Benzer biçimde sınıfın OpenWrite ve Open fonksiyonları da dosyayı açarak FileStream nesnesi vermektedir.

Binary Reader ve BinaryWriter Sınıfları: Bu sınıflarda bir stream referansını alıp onun Read/Write fonksiyonlarını kullanarak okuma ve yazma işlemlerini yaparlar. Kullanımları StreamReader ve StreamWriter sınıflarına oldukça benzemektedir. Fakat bu sınıflar yazma ve okumalarını text biçiminde değil byte düzeyinde yapmaktadır. **Örneğin**:*int a = 1234567* Biz bu sayıyı StreamWriter sınıfıyla yazdırırsak bu sayının karakterleri Unicode olarak yazı biçiminde dosyaya yazdırılır. Fakat BinaryWriter sınıfıyla yazdırırsak a yı oluşturan 4 byte dosyaya yazdırılacaktır.

```
using System;
using System.IO;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {
         FileStream fs = null;
         BinaryWriter bw = null;

         try
         {
            fs = new FileStream(@"test.txt", FileMode.Create, FileAccess.Write);
            bw = new BinaryWriter(fs);

            for (int i = 0; i < 100; ++i)
               bw.Write(i);

            //...
```

```
        }

        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        finally
        {
            if (bw != null)
                bw.Close();

        }
    }
  }
}
```

Tür İçinde Tür Bildirimleri: Bir sınıfın içinde bir sınıf, yapı ya da enum türleri bildirilebilir. Benzer biçimde yapıların içinde de diğer türler bildirilebilmektedir. **Örneğin:**

```
class A
{
        //...
        class B
        {
                //...
        }
        //...
}
```

Bir tür bir sınıf ya da yapının içinde bildirilmişse erişim belirleyicisi olarak sınıf elemanlarına getirilen belirleyicilerden biri getirilebilir. Erşim belirleyicisi kullanılmazsa yine default private algılanır.

Bir tür başka bir türün içinde bildirildiği zaman bir içerme ilişkisi söz konusu olmaz. Yukarı daki örnekte A ve B sınıfları tamamen bağımsız sınıflardır. Bu sınıflar sanki ayrık bildirilmiş gibi düşünülmelidir.

İçteki sınıf dıştaki sınıfın bildirimi içinde ve fonksiyonlarında isim olarak doğrudan kullanılabilir. Fakat en dışarıdan içteki sınıf ancak dıştakisınıf ismiyle beraber kullanılabilir. Yani biz doğrudan dışarıdan B yi kullanamayız. Onu A.B biçiminde kullanmalıyız. Şüphesiz içteki sınıfı dışarıdan kullanabilmemiz için public belirleyicisine sahip olması gerekir.

Örneğin:

```
using System;
using System.IO;

class A
{
   private int m_a;
   //...
   public class B
   {
      private int b;
      //...
```

```
    }
    //...
}
namespace CSD
{
    sealed class App
    {
        public static void Main()
        {
            //...
            A a = new A();
        }
    }
}
```

Burada bir kapsama ilişkisi söz konusu değildir. A türünden bir nesne için m_a kadar yararlanılmıştır.

Fakat

```
using System;
using System.IO;



class A
{
    private int m_a;
    //...
    public class B
    {
        private int b;
        //...
    }
    //...
}
namespace CSD
{
    sealed class App
    {
        public static void Main()
        {
            //...
            B b = new B(); //error
        }
    }
}
```

şöyle olmalıydı.

```
namespace CSD
{
    sealed class App
    {
        public static void Main()
        {
            //...
```

```
            A.B b = new A.B();
        }
    }
}
```

Görüldüğü gibi data bakımından bir içerme sözkonusu değildir. İsimsel bir içerme sözkonusudur.

Peki neden bir sınıf dışarıda bağımsız bildirilmez de başka bir sınıfın içinde bildirilir. İşte eğer bir B sınıfının kendi başına kullanımı anlamlı değilse ancak A sınıfı ile birlikte kulanımı ya da A sınıfının içinde kullanımı anlamlı ise bu durumda B sınıfını A sınıfı içinde bildirmek iyi bir tekniktir. Çünkü bu mantıksal ilişki daha iyi ifade edilmektedir. Bir sınıf başka bir sınıfın private bölümünde bildirilmişse sınıfın dışarıdan kullanımı mümkün değildir. Ancak sınıf kapsayan sınıfın içinde bildirilir.

```
using System;
using System.IO;

namespace CSD
{
    sealed class App
    {
        public static void Main()
        {

        }
        class MyCollection : IEnumerable
        {
            public IEnumerator GetEnumerator()
            {
                //...
                return new MyEnumerator(...);
            }
            private class MyEnumerator
            {
                //...
            }

        }
    }
}
```

Bir enum bir sınıf ya da yapının içine de yerleştirilebilir. Bu durumda bu enum private ise yalnızca o sınıf ya da yapının içinde kullanılabilir. Bir A sınıfının içinde private bir B sınıfının bulunduğunu düşünelim. Biz bu B sınıfını dışarıdan değil yalnızca A nın içinde kullanabiliriz. Fakat A sınıfının bir fonksiyonunun geri dönüş değeri B türünden olsun. Burada bir tuhaflık söz konusudur.

```
using System;
using System.IO;

namespace CSD
{
    sealed class App
    {
        public static void Main()
```

```
        {

        }
        class A
        {
            public B Func() ---->error
            {
                return new B;
            }
            private class B
            {
                //...
            }
            //...
        }
}
```

Eğer bu fonksiyon public is kod anlamsızdır ve bu nedenle error ile sonuçlanır. Çünkü fonksiyonu dışarıdan çağırabiliriz ve bu durumda fonksiyon bize dışarıdan kullanamayacağımız bir şey vermektedir. Burada fonksiyon private olsaydı bir sorun oluşmazdı.

```
using System;
using System.IO;

namespace CSD
{
    sealed class App
    {
        public static void Main()
        {

        }
        class A
        {
            private B Func() --->error değil
            {
                return new B;
            }
            private class B
            {
                //...
            }
            //...
        }
}
```

Aynı durum parametre değişkeni olarak kullanımda da ortaya çıkmaktadır.

```
using System;
using System.IO;

namespace CSD
{
    sealed class App
    {
        public static void Main()
```

```
    {

    }
    class A
    {
        private void Func(B b) //error
        {
            //...
        }
        private class B
        {
            //...
        }
        //...
    }
}
```

İç sınıf bildiriminin C# daki diğer önemli farklılığı da iç sınıfın kapsayan sınıfın private elemanlarına erişebilmesidir. Tabi bu erişim doğrudan değil dış sınıf türünden referans yoluyla yapılmalıdır. **Örneğin:**

```
using System;
using System.IO;

namespace CSD
{
   sealed class App
   {
      public static void Main()
      {

      }
      class A
      {
         private int m_a;

         class B
         {
            public void Func()
            {
               A a = new A();
               a.m_a = 10; //Geçerli
            //...
         }


      }
}
```

fakat dış sınıfın benzer biçimde iç sınıfın private bölümüne erişmesi geçerli değildir.