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

.NET ORTAMININ TEMEL ÖZELLİKLERİ – ARAKOD SİSTEMİ

(07.06.2007 Perşembe)

— Diller Arası Entegrasyon

.Net ortamında, bu ortam için yazılmış derleyiciler aynı ortak arakodu üretmektedir. Örneğin C# derleyicisi de, Visual Basic.Net derleyicisi de aynı arakodu üretmektedir.Bu durumda C# da yazılmış plan kodun Visual Basic.Net'te ya da Visual Basic.Net'te yazılmış olan bir kodun C# ta kullanılması tamamen doğal yöntemlerle sağlanmış olur.

Bir dilde yazılmış olan kodların başka bir dilde kullanılması önemli bir sorundur. Microsoft, .Net öncesinde COM denilen bir ara yüzle bu sorunu çözmeye çalışmıştır. Fakat bu COM sisteminin önemli dezavantajları söz konusudur. Fakat .Net'te diller arası entegrasyon zaten doğal olarak arakod düzeyinde çözülmüştür. Microsoft, .Net ortamı için 4 programlama diline ait derleyiciyi kendi yazmıştır. Bunlar C#, Visual Basic.Net, C++.Net, J# derleyicileridir. Ancak C# dışında diğer diller bir takım syntax değişiklikleri ile bu ortama zorla adapte edilmiştir. Bu nedenle C# bu ortamın birincil dilidir.

Bir programlama dilinin .Net uyumlu olması için bazı özellikleri barındırması gerekir. Bu özelliklerin neler olduğu ECMA 335 CLI standartlarında belirtilmiştir.

— Hızlı Uygulama Geliştirme

.Net ortamı ve C# programlama dili hızlı uygulama geliştirme sürecini destekleyecek nitelikte tasarlanmıştır. Uzun olmayan bir eğitim sürecinden sonra pek çoğu hazır olan bileşenleri kullanarak uzman olmayanlarında program yazabilmesi mümkün hale getirilmektedir. Hızlı uygulama geliştirme de görsel öğelerin kullanılması gerekmektedir.

— Ortak Sınıf Kütüphanesi

.Net ortamında tüm dillerde kullanılabilen ortak ve geniş bir sınıf kütüphanesi vardır. Pek çok işlem bu kütüphaneler sayesinde kolayca yapılır. Bu kütüphaneleri kullanma becerisi .Net programcılığında önemli bir yer tutar. .Net kütüphanesi, framework kütüphanesine yeni sınıflar eklenerek genişletilmektedir. Bu ortak sınıf kütüphanesinin bazı bölümleri özel isimlerle anılır. Örneğin veritabanı işini yapan kısmına ADO.Net, ekranlar oluşturmak için kullanılan kısmına Forms, web sayfası oluşturmak için kullanılan kısmına ASP.Net denir.

— Web Tabanlı Çalışma

.Net ortamı Microsoft'un web sunucuları ile de bütünleştirilmiştir. ASP.Net denilen sistemle etkileşimli web sayfaları .Net ortamı içerisinde oluşturulabilmektedir.

— Güvenlik

.Net ortamında arakod çalışması sayesinde güvenlik ihlalleri CLR tarafından engellenebilmektedir. .Net ortamı pek çok bakımdan daha güvenli bir çalışma ortamı sunmaktadır.

— Visual Studio IDE'si

.Net altında yazılım geliştirme faaliyetlerini kolaylaştırmak için kendi içerisinde editörü olan, çeşitli geliştirme araçları bulunan IDE'ler bulunmaktadır. Microsoft'un .Net IDE'si Visual Studio isimli IDE'dir. Şu anda kullanılan son sürümü Visual Studio 2005'tir. Bu IDE çeşitli paketler halinde para ile satılmaktadır. En kapsamlı versiyonu Team Edition'dur.

.NET ORTAMI NASIL KURULUR?

.Net ortamını kurmanın birkaç değişik yolu vardır.

1- **Windows Xp'nin son sürümleri, Windows Vista'nın son sürümleri zaten .Net ortamını işletim sisteminin bir parçası olarak içermektedir. Bu sistemlerde ayrıca .Net ortamın kurmaya gerek yoktur. Microsoft bundan sonraki Windows sürümlerinde .Net ortamının tıpkı Internet Explorer gibi bulunacağını söylemiştir.**

2- **Visual Studio Express Edition IDE'leri yüklenirken .Net ortamı da zaten yüklenmiş olur.**

3- **www.microsoft.com/downloads internet adresine girilerek .Net Framework 2.0 SDK indirilerek kurulur. Bu paket yaklaşık olarak 300mb civarındadır. .Net ortamının yanı sıra komut satırından çalışan çeşitli araçları da içermektedir.**

4- **Bir .Net programının çalışabilmesi için minimum kurulum .NET Framework Version 2.0 Redistributable Package isimli paketle yapılabilir.**

DERLEYİCİ VE IDE KAVRAMLARI

Derleyiciler genellikle komut satırından çalıştırılan basit bir arayüze sahip programlardır. Yaptıkları iş karmaşık olmasına karşın çalıştırılmaları çok basittir. Microsoft'un C# derleyicisi csc.exe programıdır. Hâlbuki IDE derleme dışındaki geliştirme faaliyetlerine yardımcı olan programlardır. IDE'lerin kendi editörleri, menüleri ve çeşitli yardımcı araçları vardır. Programı IDE'de yazdıktan sonra derleme aşamasına gelindiğinde IDE, komut satırı derleyicisini çağırarak işlevini yapar(csc.exe). Aslında en karışık programlar bile Notepad gibi bir editörle yazılarak komut satırından derlenebilir.

MERHABA DÜNYA PROGRAMI

(12.06.2007 Salı)

Ekrana "Merhaba Dünya" yazısını çıkartan iskelet C# programı aşağıdaki gibi yazılabilir:

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

Bir C# programı kabaca isim alanlarından, isim alanları sınıflardan, sınıflarda fonksiyonlardan oluşmaktadır. C#'ta fonksiyonlara method da denir. İki süslü parantez arasındaki bölüme blok denilmektedir. İsim alanlarının, sınıfların, fonksiyonların blokları vardır.

Programlama dilleri ile doğal diller yapı olarak birbirine benzemektedir. Programlama dillerine dilbiliminde biçimsel diller(Formal Languages) denir. Biçimsel diller matematiksel olarak açıklanabilir dillerdir. Doğal diller matematiksel olarak açıklamanaz.

Bir dilde 3 çeşit kural topluluğu vardır.
1-) Syntax Kuralları 2-) Semantik Kurallar 3-)Diğer Kurallar
Dilin tüm bu kurallarına gramer denilmektedir. Syntax doğru yazım kurallarıdır. Yani hangi yazımın doğru hangi yazımın yanlış olduğunu gösteren kurallardır. Semantik kurallar ise yazılan şeylerin ne anlama geldiğine dair kuralları içerir. Ancak syntax olarak doğru olan ifadeler semantik olarak yorumlanırlar.

Kursumuzda syntax anlatımında açısal parantezler <……..> olması gereken zorunlu öğeleri, köşeli parantezler […….] olup olmaması isteğe bağlı öğeleri ifade eder.

Bir isim alanı aşağıdaki gibi bildirilir:

```
namespace < isim >
{
     ……
}
```

Sınıf alanın genel bildirimi aşağıdaki gibidir:

```
class < isim >
{
     …….
}
```

ATOM KAVRAMI

Bir programlama dilinde yazılmış kendi anlamlı olan en küçük bölüme atom(token) denilmektedir. Gerçekten derleyiciler de ilk aşamada kaynak kodu atomlarına ayırmaktadır. Bu işleme Lexical Analys denir. Daha sonra derleyiciler 2. aşamada atomların diziliş sırasının geçerli bir program oluşturup oluşturmadığına bakar. Buna Syntax Analys denir. 3. aşamada programın ne yapmak istediği tespit edilir. Buna Semantic Analys denir. En son aşamada programcının istediği şeyleri yapacak kod üretilir. Buna Code Generation denir.

Atomlar 6 gruba ayrılır:

1- **Anahtar Sözcükler(Keywords): Değişken olarak isimlendirilmesi yasaklanmış, dil için özel anlamı olan sözcüklere anahtar sözcük denir.**
2- **Değişkenler(Identifier/Variables): İsmini programcının istediği gibi verebildiği atomlara değişken diyoruz.**
3- **Sabitler(Constants/Literals): Doğrudan yazılmış olan sayılara sabit denir.**
4- **Operatörler(Operators): Bir işleme yol açan, işlem sonrasında bir değer üretilmesini sağlayan atomlara operatör denir.**
5- **Stringler(Strings): İki tırnak içindeki yazılar tırnakları ile birlikte tek bir atomdur. Bu atomlara string denir.**
6- **Ayıraçlar(Delineters): Yukarıdaki grupların dışında kalan atomlara ({, ;) ayıraç denir.**

DERLEYİCİLERİN HATA MESAJLARI

Derleyici, atomlara ayırma işlemini yaptıktan sonra programın doğru yazılıp yazılmadığını denetlerken geçersiz bir durumla karşılaşırsa hatanın kaynağını açıklayan bir hata mesajı verir. Sonuçta derleme işlemi başarısız olur. Derleyiciler genellikle tüm error mesajlarını tespit edip bir hamlede listelemektedir. Milyon satır programda bir error olsa bile program derlenemez.

Error mesajları genellikle syntax aşamasında ortaya çıktığı için bunlara "Syntax Error" denmektedir. Bazı kodlamalar syntax olarak tamamen geçerli olduğu halde mantıksal bir yanlış içerebilirler. Bu tür durumları derleyici fark ederse "Warning (Uyarı)" mesajları verir. Derleyicinin uyarı mesajı vermesi kodun derlenmesini engellemez. Fakat eğer gerçekten derleyicinin sözüne ettiği gibi bir mantık hatası varsa kod çalışırken istenileni yapamayacaktır.

C# PROGRAMININ DERLENEREK ÇALIŞTIRILMASI

Komut satırından C# derleyicisinin çalıştırılabilmesi için PATH çevre değişkeninin set edilmiş olması gerekmektedir. Komut satırı için Donatılarda değil Visual Studio 2005 Command Prompt'tan geçmek gerekmektedir. Programlar herhangi bir text editörle (Örneğin Notepad) yazılabilir. Derleme işlemi şöyle yapılmalıdır:

```
csc < proramın ismi >
```

BOŞLUK KARAKTERLERİ (WHITE SPACE)

Klavyeden boşluk oluşturmak için kullanılan Space, Tab ve Enter karakterlerine boşluk karakterleri denir.

C#'IN YAZIM KURALI

C#'ın yazım kuralı 2 madde de özetlenebilir.

1- Atomlar arasında istenildiği kadar boşluk karakteri kullanılabilir.
2- Atomlar istenildiği kadar bitişik yazılabilir. Fakat değişkenler ile anahtar sözcükler arasında en az bir boşluk karakteri eklemek gerekir.

C# PROGRAMININ ÇALIŞMASI

Bir isim alanında pek çok sınıf, bir sınıf alanında da pek çok fonksiyon bulunabilir. Fakat Main ismindeki fonksiyonun özel bir anlamı vardır. C# programı Main isimli fonksiyonla çalışmaya başlar Main fonksiyonu bitince programda biter. Main fonksiyonu herhangi bir sınıfın içinde olabilir. Fakat toplam da bir Main fonksiyonu olmalıdır.

Bir fonksiyonun bizim tarafımızdan yazılmasına fonksiyonun tanımlanması (function definetion) denir. Fonksiyonun çalıştırılmasına ise fonksiyon çağırma (function call) denir.

Fonksiyon tanımlamanın genel şekli aşağıdaki gibidir:

```
[Erişim Belirleyicisi] [Static] <Geri Dönüştürücü> <İsim>
([Parametre Bildirimi])

{
……….

}
```

Daha önce tanımlanmış bir fonksiyonun çağrılma işlemi de aşağıdaki gibi yapılır:

```
[İsim Alanı İsmi] [.] [Sınıf İsmi] [.] <Fonksiyon İsmi>
([Parametre Listesi]) ;
```

Anahtar Not

Program terimi çok eskiden söylenmiştir ve günümüzün gereksinimini tam karşılamamaktadır. Program genellikle tek bir exe'yi tanımlamak için kullanılır. Halbuki uygulama (application) bir yada birden fazla programı içeren bir projedir. Yani uygulama terimi daha geniş bir anlam ifade eder.

Windows uygulamaları GUI ve Konsol uygulamaları olarak ikiye ayrılır. Pencereli tip Windows uygulamalarına GUI uygulamalar diyoruz. Siyah ekranla çalışan uygulamalar da konsol uygulamalar diyoruz. Konsol uygulamalarında bir imleç vardır. Her zaman imleci bulunduğu yere yazı yazılır, imleçte yazılan miktar kadar ilerletilir. Consol sınıfının Write fonksiyonu imleci sonda bırakırken WriteLine fonksiyonu ekrana yazdırma işleminden sonra imleci aşağı satırın başına geçirir.

C#'TA TEMEL VERİ TÜRLERİ

Tür(Type) bir değişkenin bellekte kaç byte yer kapladığını ve onun içerisindeki bilgilerin hangi formatta bulunduğunu anlatan bir kavramdır. Temel türler bir tablo halinde aşağıdaki gibi ifade edilebilirler:

	Tür Belirten Anahtar Sözcükler	Uzunluk (Byte)	Yapı Karşılığı	Sınır Değerler
TAMSAYI TÜRLERİ	int	4	System.Int32	-2^{31} , $+2^{31}$
	Uint	4	System.UInt32	0 , +4 294 967 295
	Long	8	System.Int64	-2^{63} , $+2^{63}$
	Ulong	8	System.UInt64	
	Short	2	System.Int16	-32768 , +32767
	Ushort	2	System.UInt16	0 , +65535
	Byte	1	System.Byte	0 , +255
	sbyte	1	System.SByte	-128 , +127
GERÇEK SAYI TÜRLERİ	float	4	System.Single	Az Duyarlıklı Gerçek Sayı
	double	8	System.Double	Duyarlılık Yüksek Sayı
	decimal	16	System.Decimal	28 Digit Yuvarlama Hatasız Gerçek Sayı
	char	2	System.Char	0 , 65536
	bool	1	System.Bool	True , False

int türü bir işaretli tamsayı türüdür. int türü her sistemde 4 byte uzunluktadır. (C'de olduğu gibi derleyiciye bağlı değildir.)

uint türü int türünün işaretsiz biçimidir. Her tamsayı türünün işaretsiz bir versiyonu da vardır. İşaretsiz türlere pozitif sayılar yerleştirilir. İşaretli versiyona göre 2 kat daha büyük pozitif sayılar yerleştirilebilir.

long türü 8 byte uzunluğunda işaretli bir tamsayı türüdür.

short türü int türünün yarı uzunluğunda işaretli bir tamsayı türüdür.

ushort short türünün işaretsiz türüdür.

byte ve sbyte türleri işaretsiz ve işaretli 1 byte uzunluğunda tamsayı türleridir.

float türü 4 byte uzunluğunda noktalı bir sayı türüdür. float türünün yuvarlama hatalarına direnci zayıftır.

double türü float türünden 2 kat daha fazla duyarlılıktadır. Dolayısı ile yuvarlama hatalarına direnci daha yüksektir.

decimal türü 28 digitli, noktalı bir sayıyı yuvarlama hatası olmadan tam olarak tutar. Yapay bir türdür. Özellikle finansal hesaplamalarda tercih edilir.

Gerçek sayı türlerinin işaretli ve işaretsiz türleri yoktur. Bu türler her zaman işaretlidir.

char türü bir karakterin, karakter tablosundaki sayısal karşılığını tutmaktadır. Her sistemde 2 byte yer kaplar. Karakter tablosu Unicode tablosudur. (ASCII değil)

bool türü yalnızca doğru – yanlış bilgisini tutar.

C#'ta en fazla kullanılan tamsayı türü int türüdür. Programcı ancak başka bir gerekçe varsa bu türün dışında bir tür denemelidir. En fazla kullanılan gerçek sayı türü double türüdür. Çünkü yuvarlama hatalarına direnci yüksektir.

BİLDİRİM İŞLEMİ

Kullanılmadan önce bir değişkenin derleyiciye tanıtılması işlemine bildirim diyoruz. Bildirim yapılmadan bir değişken kullanılamaz. Bildirim işleminin genel biçimi aşağıdaki gibidir:

```
< Tür Belirten Sözcük >  < Değişken Listesi > ;
```

Örneğin:

```
int a;
int a, b, c;
double x, y, z;
int a;  double b;
```

C#'ta bildirimler 2 yerde yapılır:

1- **Fonksiyonların içinde**
2- **Fonksiyonların dışında fakat sınıf içinde**

Fonksiyonların içinde yapılan bildirimlere yerel bildirimler, sınıf bildirimi içerisindeki bildirimlere veri elemanı bildirimi denir. Benzer biçimde fonksiyon içerisindeki değişkenlere yerel değişkenler, sınıf içerisindeki değişkenlere veri elemanı diyoruz.

Bir fonksiyonun ana bloğu olmak zorundadır. Fakat iç içe veya ayrık blok açılabilir. Yerel değişkenler bloğun herhangi bir yerinden bildirilebilir. Ancak bildirilme noktasından sonra kullanılabilir.

Bildirilen bir değişken tüm programda kullanılmaz. Yalnızca belirli bir program aralığında kullanılabilir. Bir değişkenin kullanılabildiği program aralığına faaliyet alanı(scop) denir.

Bir yerel değişken hangi blokta bildirilmişse bildirim noktasından sonra, bildirimin yapıldığı blokta ve o bloğun kapsadığı bloklarda kullanılabilir. Ama o blok dışında kullanılamaz.

C#'ta bir yerel değişkenin bildirildiği blokta ve o bloğun kapsadığı bloklarda aynı isimde başka bir değişken bildirilmez.
Örneğin:

```
public static void Main()
{
     int a;
     {
          long a;        --> error
     }
}
```

Fakat ayrık bloklarda aynı isimli değişkenler bildirilebilir.
Örneğin:

```
public static void Main()
{
     {
          int a;         --> Geçerli
     }

     {
          int a;         --> Geçerli
     }
}
```

İç içe bloklarda aynı isimli değişkenler bildirilemez. Örneğin aynı isimli bildirim önce iç blokta sonra da dış blokta tanımlansa da geçersizdir.

```
public static void Main()
{
     {
```

```
        int a;
    }

    int a;          --> error
}
```

DEĞİŞKENLERE İLK DEĞER VERİLMESİ

Bir değişken bildirilir bildirilmez içerisine değer atanabilir. Bu işleme ilk değer verme denir.
Örneğin:

```
int a, b=10, c;
```

C#'ta bir değişkenin içerisine henüz değer atanmamışsa, bu değişken içerisindeki değerin kullanılacağı bir ifade kullanılamaz. Kullanılırsa error oluşur.
Örneğin:

```
int a, b;
b = a;          --> error
```

Yukarıdaki örnekte b'nin kullanımı geçerli, a'nın kullanımı geçersizdir.
Örneğin:

```
int a, b;

a = 10;         --> Geçerli
b = a;          --> Geçerli
```

Yerel değişken bildirilir bildirilmez faaliyet alanına girmiş kabul edilir.
Örneğin:

```
int a = 10, b = a;       --> Geçerli

int a, b = a;            --> Error
```

Bir değişkenin içerisindeki değer değişkenin türü ne olursa olsun Console sınıfının Write ve WriteLine fonksiyonları ile yazdırılabilir.

SABİTLER

Doğrudan yazılan sayılara sabit denir. Yalnızca değişkenlerin değil sabitlerinde bir türü vardır. İyi bir C# programcısının sayıya baktığında türünü tespit edebilmesi gerekmektedir.

1- Sayı nokta içermiyorsa ve sonunda da bir ek yoksa sayı *int, uint, long, ulong* sınırlarının hangisinin içine ilk giriyorsa sabit o türdür.

```
0 à int        1 à int        3 000 000 000 à uint
```

Sayı eğer ulong sınırları içine de sığmıyorsa error oluşur.

2- Sayı nokta içermiyorsa fakat sayının sonunda küçük ya da büyük L son eki varsa *lon* ve *ulong* türlerinin hangisinin içerisine ilk giriyorsa o türdür.

```
0L à long      1L à long
```

3- Sayı nokta içermiyorsa fakat sayının sonunda küçük ya da büyük U son eki varsa sayı *uint* ya da *ulong* türlerinin hangisinin içine ilk kez giriyorsa o türdür.

```
1U à uint      2U à uint      5 000 000 000 à ulong
```

Sayı ulong sınırlarının içine de sığmıyorsa error oluşur.

4- Sayı nokta içermiyorsa fakat sayının sonunda küçük ya da büyük UL veya LU son eki varsa sabit *ulong* türüdür. Sabit ulong sınırına da sığmıyorsa error oluşur.

```
1UL à ulong
```

5- Sayı nokta içermiyorsa fakat sonunda ek yoksa sabit double türdendir. Örneğin:

```
3,2 à Double    7,4 à Double
```

Sayı nokta içersin yada içermesin sonunda " D , d " son eki varsa sabit double türdendir.

6- Sayı nokta içersin yada içermesin sonunda " F , f " son eki varsa sabit float türdendir. Örneğin:

```
3F à Float      3,2 à Float
```

7- Bir karakter tek tırnak içerisine alınırsa bu bir sabit belirtir. O karakterin Unicode tablodaki sıra numarasını belirtmektedir.

C#'da yazılar ve karakterler Unicode olarak tutulup saklanmaktadır.

C# char türü sayısal türü sayısal bir türdür ve aritmetik işlemlere sokulabilir. Tek tırnak içerisindeki karakterler char türdendir.

Bazı karakterler ekrana basılmak istendiğinde görüntülenmez. Görüntü yerine birtakım olaylar gerçekleşir. Örneğin:

Bip sesi duyulur yada cursor hareket eder.

Tıpkı C'de olduğu gibi çok kullanılan görüntülenemeyen karakterler ters bölü biçiminde ifade edilmiştir. Tek tırnak içerisindeki ters bölü sonra özel bazı karakterler bazı görüntülenemeyen karakterlerin yerini tutmaktadır. Bunların listesi aşağıdaki gibidir:

- **' \a ' à Bip sesi çıkar (Alert)**
- **' \b ' à Cursor bir geri gider (Backspace)**
- **' \f ' à Printere gönderilmesi ile sayfa çıktısı (Form Feed)**
- **' \n ' à Cursor aşağı satırın başına geçer (New Line)**
- **' \r ' à Cursor bulunan satırın başına geçer (Curriace Return)**
- **' \t ' à Cursor bir tab ilerler**

8- Sayı nokta içersin ya da içermesin sonunda küçük yada büyük harf ' m ' varsa sabit decimal türdendir.

9- True ve false anahtar sözcükleri bool türden sabit belirtir.

10- C#'da byte, sbyte, short, ushort türden sabit kavramı yoktur.

Anahtar Not: ***Ters bölü karakterine ilişkin karakter sabiti ' \ ' biçiminde yazılamaz. ' \\ ' biçiminde yazılmak zorundadır. Benzer biçimde tek tırnak karakterine ilişkin karakter sabiti de ' ' ' biçiminde değil ' \' ' biçiminde yazılmak zorundadır.***

ATAMA İŞLEMİNDE TÜR UYUŞUMU

Atama işleminde sol taraf ve sağ taraf farklı türlerden olursa bu türlerin durumuna göre geçersiz olabilmektedir. Bu konu ileride ele alınacaktır. Fakat o ders gelene kadar tür uyuşumunu sağlayacağız.

FONKSİYONLAR

C#'da fonksiyon terimi yerine method terimi de kullanılmaktadır. Fakat kursumuzda fonksiyon terimini tercih ediyoruz.

Bir fonksiyon tanımlamanın genel şekli aşağıdaki gibidir:

```
[Erişim Belirleyicisi] [static] <geri dönüş değerinin türü>
<fonksiyon ismi> ([parametre bildirimi])
{
     //…
}
```

Erişim belirleyicisi yalnızca şu anahtar sözcülerden biri olabilir.

```
public, protected, private, internal, protected internal
```

Erişim belirleyicisi hiç yazılmayabilir. Eğer yazılmamışsa `private` yazılmış gibi işlem görür. Fakat bu konu başka bir derste ele alınana kadar biz `public` belirleyicisini kullanacağız.

Static anahtar sözcüğü belirtilebilir ya da belirtilmeyebilir. Static anahtar sözcüğü, erişim belirleyicisi anahtar sözcükle aynı grupta yer aldığı için yer değiştirmeli olarak yazılabilir. Static anahtar sözcüğü ile tanımlanmış fonksiyonlara (static method) static, static anahtar sözcüğü kullanılmadan bildirilmiş fonksiyonlara static olmayan (instance method) fonksiyonlar denilmektedir.

Main fonksiyonu özel bir sınıfta bulunmak zorunda değildir. Herhangi bir sınıfta bulunabilir. Fakat Main fonksiyonu static olmak zorundadır. Erişim belirleyicisi public olmak zorunda değildir.

C#'da isim alanları (namespace) iç içe bulunabilir. Hiçbir isim alanı içinde bulunmayan bölümde bir isim alanıdır. Buna da global isim alanı denmektedir. Global isim alanına da doğrudan sınıf alanı tanımlanabilir. Fakat iyi bir teknik değildir.

Global isim alanı içerisindeki system isim alanı .Net'in kendi sınıflarının bulunduğu isim alanıdır. .Net'in tüm sınıfları ya doğrudan system isim alanı içerisindedir ya da system isim alanı içerisindeki isim alanları içerisindedir.

Anahtar Not:

Foo ve Bar isimleri son 10 yıldır yazılımında örneklerde uydurma fonksiyon isimleri olarak kullanılmaktadır. Bu sözcüklerin bu bağlamda anlamları yoktur.

Bir static fonksiyon en genel olarak önce isim alanı ismi sonra sınıf ismi belirtilerek çağrılır. Çağırma işlemi genel biçimi aşağıdaki gibidir:

```
[isim alanı ismi] [.] [sınıf ismi] [.] <fonksiyon ismi>
([parametre listesi]) ;
```

Eğer çağrılacak fonksiyon aynı sınıfın static fonksiyonu ise isim alanı ismi ve sınıf isminin belirtilmesine gerek yoktur. Sadece fonksiyon ismi belirtilebilir. Çağrılacak static fonksiyon aynı isim alanı içerisindeki başka bir sınıf içerisinde ise isim alanı isminin belirtilmesine gerek yoktur. Nihayet fonksiyon başka bir isim alanı içerisindeki sınıfın fonksiyonu ise bu durumda tüm nitelikleri yapmak gerekmektedir.

FONKSİYONLARIN GERİ DÖNÜŞ DEĞERLERİ (RETURN VALUE)

Fonksiyon çalıştırıldıktan sonra onu çağıran fonksiyona ilettiği değere geri dönüş değeri denir. Geri dönüş değeri bir değişkene atanabilir, aritmetik işlemlerde kullanılabilir. Fonksiyonun geri dönüş türü herhangi bir türden olabilir. Eğer geri dönüş türünün yerine Void anahtar sözcüğü yazılırsa bu durum fonksiyonun geri dönüş değerine sahip olmadığı anlamına gelir.

Fonksiyonun geri dönüş değeri ' return ' anahtar sözcüğü ile belirtilir. Return anahtar sözcüğünün genel biçimi aşağıdaki gibidir:

```
return [ifade] ;
```

Return anahtar sözcüğünün iki işlevi vardır:
1- Fonksiyonu sonlandırır. 2- Geri dönüş değerini oluşturur.

Fonksiyonun geri dönüş değerinde return anahtar sözcüğünün kullanılması zorunludur. Programın her olası akışında derleyici return anahtar sözcüğü ile karşılaşmak zorundadır.

Void fonksiyonlarda return kullanılabilir. Fakat return anahtar sözcüğüne bir değer yazılamaz. Bu durumda return, fonksiyonu sonlandırmak için kullanılmıştır.

Fonksiyonun geri dönüş değeri önce derleyici tarafından yaratılan geçici bir değişkene atanır, oradan alınarak kullanılır. Zaten fonksiyonun geri dönüş değerinin türü de bu geçici değişkenin türüdür. Örneğin:

```
x = Func();
```

Burada aslında aşağıdaki gibi bir işlem söz konusudur:

```
temp = return ifadesi;
```

```
x = temp;
```

Yaratılan bu geçici değişken yine derleyicinin kendisi tarafından ilgili ifade bittiğinde yok edilir.

O halde aslında return işlemi bir atama işlemidir. Geçici değişkene atama yapılmaktadır.

Fonksiyonun geri dönüş değerine sahip olması onu kullanmayı zorunlu hale getirmez. Yani geri dönüş değeri extra bir değerdir.

İFADE KAVRAMI

Değişkenlerin, operatörlerin ve sabitlerin her bir birleşimine ifade denir. Örneğin:

```
x , x + 1, 10, x + 1 + 5, x + y birer ifadedir.
```

YEREL DEĞİŞKENLERİN ÖMÜRLERİ

Bir değişkenin bellekte kaldığı zamana ömür(duration) denir.

Anahtar Not:

C#'da /* ve */ arasındaki yazılar derleyici tarafından dikkate alınmaz. Programcı kaynak koda ilişkin açıklamaları bu şekilde yazabilir. Benzer biçimde //'dan satır sonuna kadarki yazıda dikkate alınmaz.

Bir yerel değişken, programın akışı tanımlama noktasına geldiğinde oluşturulur. Akış değişkenin tanımlandığı bloktan çıktığında otomatik yok edilir. Yani bir programın yerel değişkenlerinin hepsi aynı anda bellekte değildir. Yerel değişkenlerin bu şekilde oluşturulup yok edilmeleri çok hızlı yapılmaktadır. Yerel değişkenler belleğin stock denilen bölümünde oluşturulur.

Yerel bir değişken tanımlandığı blok dışında kullanamamızın nedeni zaten o değişkenin blok dışına çıkıldığında bellekte yaşamıyor olmasından kaynaklanmaktadır.

OPERATÖRLER

Bir işleme yol açan işlem sonrasında bir değer üretilmesini sağlayan atomlara operatör denir. Operatörleri üç biçimde sınıflandırabiliriz:

1. **İşlevlerine göre**
2. **Operant sayılarına göre**
3. **Operatörün konumuna göre**

1. **İşlevlerine Göre Sınıflandırma: Operatörlerin hangi amaçla kullanıldığına yönelik sınıflandırmadır.**
   a. **Aritmetik operatörler + - * / operatörleridir.**

b. Karşılaştırma operatörleri (Comparison veya relational operators) < > gibi operatörlerdir.
c. Mantıksal operatörler(Logical operators) Ve veya not tarzı operatörlerdir.
d. Bit operatörleri (Bit seviyesinde operatörlerdir)
e. Özel amaçlı operatörler (Special purposes)

2. Operant Sayılarına Göre Sınıflandırma: Operatörlerin işleme soktuğu ifadelere operant denir. a+b
   a. Tek operantlı operatörler(Unary operators) a!
   b. İki operantlı operatörler (Binary operatorler) a+b
   c. Üç operantlı operatörler (Ternary Operators)

3. Operatörün Konumuna Göre Sınıflandırma:
   a. Önek operatörler - Prefix
   b. Araek operatörler - Infix
   c. Son ek operatörler – Postix

C sharpta tüm iki operantlı operatörler araek- infix durumundadır.

Bir operatörü teknik olarak açıklayabilmek için üç sınıflandırmada da operatörün nereye düştüğünü belirtmek gerekir. Örneğin "+ operatörü iki operantlı, araek bir aritmetik operatördür."

OPERATÖRLER ARASINDAKİ ÖNCELİK İLİŞKİSİ

Aynı ifade içerisinde birden fazla operatör kullanıldığında bu operatörler belirli bir sırada yapılır. Operatörlerin arasındaki öncelik ilişkisi öncelik tablosu denilen bir tablo ile gösterilir. Öncelik tablosu satırlarda oluşur. Üst satırlar alt satırlardan daha önceliklidir.

Öncelik Tablosu:

()	Soldan Sağa
* /	Soldan Sağa
+ -	Soldan Sağa
=	Sağdan Sola

Öncelik tablosunda her satırın sağına Soldan sağa veya sağdan sola ifadesi yazılıdır. Soldan sağa demek, o satırda bulunan operatörlerin "ifade içerisinde hangisi soldaysa o önce yapılır" demektir. Aynı satırdaki operatörler eşit önceliklidir. Soldan sağa veya sağdan sola bu eşitlikteki önceliği bildirir. Yani öncelik tablosunda, aynı satırdaki operatörler farklı sırada yazılsa da aynı anlama gelir.

*** / + - Operatörleri:** Bu operatörler, iki operantlı araek aritmetik operatörlerdir. Klasik dört işlem yaparlar.

% Operatörü: Bu operatör de, iki operantlı araek bir aritmetik operatördür. 10 % 3 şeklinde gösterilir. Bu operatör soldaki operantın sağdaki operanda bölümünden elde edilen kalanı verir.

Anahtar Notlar : C ve C++'da % operatörünün operantları tam sayı türlerine ilişkin olmak zorundadır ama C# da böyle bir zorunluluk yoktur. Yani 20.2 % 3 veya -20 % 3 geçerli bir mod kullanımıdır.

Öncelik Tablosu:

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

% operatörü * ve / operatörleri ile aynı düzeyde soldan sağa bulunur.

İşaret + ve İşaret – operatörleri: Tamamen farklı operatörler aynı sembolle ifade ediliyor olabilir. Örneğin; - a ya da + a derken kullanılan – ve + operatörleri toplama ve çıkartma operatörleri değildir. Bunlara işaret + ve işaret – operatörleri denir. Bu operatörler tek operantlı, önek operatörlerdir. – operatörü operantın negatif değerini elde eder. İşaret + operatörü, sırf işaret eksi operatörü olduğu için bütünlük sağlamak amacıyla bulundurulmuştur ve bir işlevi yoktur. Yani, operandın değerinin aynısını üretir. Bu operatörler, öncelik tablosunun ikinci düzeyinde sağdan sola grupta bulunur. İşlevine göre aritmetik bir operatördür.

+	**-**		**Sağdan Sola(işaret + ve işaret - operatörleri)**
()			**Soldan Sağa**
*****	**/**	**%**	**Soldan Sağa**
+	**-**		**Soldan Sağa**
=			**Sağdan Sola**

```
{
    int a;

    a = -3;

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

Ekran Çıktısı: 3

Anahtar Notlar

Atomlara ayırma işlemi sırasında tıpkı C/C++ daki gibi atom olmaya aday en uzun karakter kümesi tek bir atom olarak ele alınmaktadır. Bu nedenle, iki sembolden oluşan bazı operatörler vardır. Bu durumda bu iki sembolün bitişik yazılması gerekir. Eğer bu iki sembol ayrı yazılırsa, derleyici iki ayrı atom olduğu fikrine kapılır ve error oluşur. Örneğin;

a > = b ifadesi a > = b olarak yazılırsa error oluşur.

Hiç şüphesiz derleyici, toplama ve çıkarma operatörlerini ifade içerisindeki konumlarından hareketle işaret + ve işaret – operatörlerinden ayırmaktadır. Örneğin;

```
z = x ----y;

 z =x     -          - - -        y;
          Çıkartma   İşaret -
          Operatörü  Operatörü
```

X'ten sonraki ilk – işareti çıkartma olmakla beraber diğer – operatörleri işaret – operatörleridir.

Anahtar Notlar

Öncelik parantezinin içerisinde ifade tanımına uyan bir atom grubunun bulunması zorunludur. Mademki, tek başına bir operatör ifade değildir, o halde tek başına bir operatör öncelik parantezi içerisine alınamaz.

++ ve -- operatörleri : Bu operatörler, tek operantlı hem önek hem de sonek biçiminde kullanılabilen operatörlerdir. ++ artırma (increment) operatörü, -- operatörüne eksiltme (decrement) operatörü denir. ++, operandı olan değişkenin değerini bir artırır, -- ise bir eksiltir. Bu operatörlerin önek ve sonek kullanımları arasında bir fark vardır. Bu operatörler, ikinci düzeyde sağdan sola bulunurlar.

()				**Soldan Sağa**
+	**-**	**++**	**--**	**Sağdan Sola (işaret + ve işaret - operatörleri)**
*****	**/**	**%**		**Soldan Sağa**
+	**-**			**Soldan Sağa**
=				**Sağdan Sola**

Önek kullanımında, arttırma ve eksiltme tablodaki öncelikte yapılır ve geri kalan işleme değişkeni arttırılmış veya eksiltilmiş hali sokulur.

```
a = 3
b = ++a * 2
i1 = ++a => 4
i2 = i1 * 2 => 8
i3 => b = i2 = 8
```

Hâlbuki son ek kullanımda yine artırma ve eksiltme yapılır ama geri kalan işlemlere değişkenin *artırılmamış* veya *eksiltilmemiş* hali sokulur.

```
a = 3
b = ++a * 2
i1 = ++a => 4
i2 = i1 * 2 => 6
i3 => b = i2 = 6
```

Örneğin; aşağıdaki ifade için

```
int a = 3, b = 2, c;
c = ++a * b -- * 2 ; //c=16 b=1 a=4 değerlerini alır.

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

Örneğin:

```
int a = 3, b;
b = ++a; //a =4 b = 4
```

Örneğin;

```
int a = 3, b;
b = a++; //a =4 b = 3
```

Şüphesiz, bu operatörler tek başlarına kullanıldığında ön ek ya da sonek biçimler arasında bir fark oluşmaz. Örneğin;

++a;
ile
a++;
arasında bir fark yoktur. Çünkü tek başlarına kullanılmışlardır.

++ ve – operatörlerinin operantlarının değişken olması gerekir. Örneğin;

```
++3; //error verir.
```

Anahtar Notlar (C'ciler için)

C/C++ da bir değişken bir ifadede ++ ya da -- ile kullanılmışsa o değişkenin o ifadede bir daha gözükmemesi gerekir. Aksi halde, tanımsız davranış oluşur. Hâlbuki C#' da böyle bir durum söz konusu değildir. Aşağıdaki ifade tamamen geçerlidir:

```
int a=3, b;
b=++a + ++a; //b = 9 a=5
```

Karşılaştırma operatörleri: C# da altı tane karşılaştırma operatörü vardır. <, >, <=, >=, ==, != operatörleri aritmetik operatörlerden daha düşük önceliklidir.

()	**Soldan Sağa**
+ - ++ --	**Sağdan Sola(işaret + ve işaret - operatörleri)**

* / %	Soldan Sağa
+ -	Soldan Sağa
=	Sağdan Sola
<, >, <=, >=, ==, !=	Soldan Sağa
=	Sağdan Sola

Bu operatörlerde bool türden değer üretilir. Yani sonuç yalnızca bool türden değişkenlere atanabilir. Önerme doğruysa true değerini, yanlışsa false değerini üretirler.

```
int a = 3;
bool b;
b = a > 1; //b=True
```

```
bool b;
b = 1 + 1 < 2 + 3;
System.Console.WriteLine(b);

Ekran Çıktısı: False
```

Mantıksal Operatörler: Üç adet mantıksal operatör vardır. ! not operatörüdür. Tek operantlı ve önek bir operatördür. && yani and operatörü, || yani or operatörü'dürler. Bu operatörler iki operantlı araek operatörlerdir.

! öncelik tablosunun ikinci düzeyinde sağdan sola bulunur. && ve || karşılaştırma operatörlerinden daha düşük önceliklidir.

()	Soldan Sağa
+ - ++ --	Sağdan Sola(işaret + ve işaret - operatörleri)
* / %	Soldan Sağa
+ -	Soldan Sağa
=	Sağdan Sola
<, >, <=, >=, ==, !=	Soldan Sağa
&&(and işareti)	Soldan Sağa
||(or işareti)	Soldan Sağa
=	Sağdan Sola

Mantıksal operatörün operantlarının bool türden olması zorunludur. Bu operatörler, ilgili mantıksal işlemi yaparlar ve sonuç olarak bool türden bir ürün üretirler. Örneğin:

```
e = a > b && c > d;
```

Yukarıdaki ifade de önce a > b sonra c > d yapılır. Bu işler bool türden ürün verdiğine göre && operatörünün kullanımı doğrudur. Tabi, e'nin de bool türden olması gerekir. Örneğin:

```
bool b;
b = 3 > 2 && 2 < 8 ;
System.Console.WriteLine(b);
```

Ekran Çıktısı: True

Örneğin:

```
bool b;
b = !Func();
System.Console.WriteLine(b);
public static bool Func()
{
     return true;
}
```

Ekran Çıktısı: False

&& ve || operatörlerinin kısa devre özelliği vardır. Bu operatörler klasik öncelik kuralına uymazlar. || operatörünün sağında ne olursa olsun önce sol tarafı tamamen yapılıp bitirilir, eğer sol tarafındaki ifade true ise sağ taraf hiç yapılmaz ve sonuç true olarak tespit edilir. && operatörünün de önce sol tarafı tamamen yapılıp bitirilir. Eğer sol taraf false ise sağ taraf hiç yapılmaz ve sonuç false olarak tespit edilir.

Tek | ve tek &, bit düzeyindeki or ve and operatörleridir. Her ne kadar bu operatörler bit operatörleri olsa da bool türlerle de kullanılabilir. Fakat bu operatörlerin kısa devre özelliği yoktur. Yani operatörün her iki tarafı da yapılmaktadır.

Atama operatörü: Atama operatörleri de iki operantlı araek operatörlerdir. Atama operatörünün solunda bir değişken bulunmalıdır. Atama operatörü de bir değer üretmektedir. Atama operatörünün ürettiği değer, değişkene atanmış olan değerdir. Biz bu ürünü, başka işlemlerde kullanabiliriz. Örnek:

```
a = b = 0
i1 = b = 0;
a = i1 => 0
```

Örnek:

```
a = (b = 10) + 20;
```

Burada önce b'ye 10 atanır. Elde edilen 10 , 20 ile toplanıp a'ya atanır.

Noktalı Virgül'ün İşlevi:

a = 10+20;

```
b = 30;
```

; ifadeleri sonlandırmak için kullanılır. İki; arasındaki ifade diğerlerinden farklı bir biçimde önceliklendirilir ve yapılır. Eğer programcı ;'ü unutursa önceki ifade ile sonraki ifade tek bir ifade olarak değerlendirilir ve error oluşur. Örneğin;

```
a = 10 + 20
b = 30; // bu ifade iki satırdan oluşmaktadır. Çünkü üst satırda; yok
```

Programlama dillerinde, ifadeleri sonlandırmak için kullanılan bu tür atomlara sonlandırıcı (terminator) denilmektedir.

Basic gibi bazı dillerde sonlandırıcı olarak \n (alt satırın başına geç) karakteri kullanılmaktadır. Bu nedenle bu dillerde her satıra tek bir ifade yazılmaktadır.

C#'da bir işlemin sonucunun, gizli ya da açık bir değişkene atanması gerekir. Örneğin;

a + b; işlemi anlamsızdır ve error oluşturur. Fakat fonksiyon çağırmaları (fonksiyon zaten bir yan etkiye yol açtığı için) bu biçimde değerlendirilmez. Örneğin;

```
Func();
```

Fonksiyonun geri dönüş değeri kullanılmak zorunda değildir. Bu durum geçerlidir. Hâlbuki Func()+1; ifadesi geçersizdir ve error verir.

Anahtar Notlar

C/C++ da bir işlemin sonucunun bir yere atanması zorunlu değildir. Yani yukarıdaki örneklerde C/C++ da geçerlidir ve derleyici en fazla uyarı verir.

VISUAL STUDIO IDE'SİNİN TEMEL KULLANIMI

Visual Studio IDE'sinde bir C# programını derleyip çalıştırmak için önce bir proje oluşturmak gerekir. Projeler ise solution denilen bir ortamda bulunmaktadır. Bir solution, bir ya da birden fazla projeyi içerebilir. O halde programcının bir solution yaratıp içine bir proje yerleştirmesi gerekir. Bunun için file/new/project menüsü seçilir. Karşımıza new project isimli bir diyalog penceresi çıkar. Bu diyalog penceresinden proje türü olarak Visual C#/windows/empty project seçilir.

Normal olarak hem solution için hem de project için ayrı bir klasör yaratılır. Yani, normal olarak yaratım işleminin sonunda bir solution klasörü bir de onun içinde project klasörü oluşturur.

Solution klasörünün yaratılacağı taban klasör location kısmında belirtilir. Sonra projeye bir isim verilir. İstenirse solutiona da ayrı bir isim verilebilir. Bu isimler yaratılacak klasörlerin isimleri olur.

Bir solution yaratıldığında "solution explorer" isimli bir pencere de yaratılır. Bu pencere kapatılırsa view menüsünden ya da araç çubuklarından yine görüntülenir.

Proje yaratıldıktan sonra .cs uzantılı bir kaynak dosyayı projenin içine yerleştirmek gerekir. Kaynak dosya eklemek için project/add new item seçilebilir. Ya da solution explorer'da proje isminin üstüne sağ tıklanarak çıkan bağlam menüsünden de seçim yapılabilir. Daha sonra çıkan add new item diyalog penceresinden "code file" seçilir. Kaynak dosyaya isim verilerek yaratılır. Daha sonra program dosyaya yazılır.

Bu işlemlerden sonra derleme için built/built solution seçilir. Artık exe oluşmuştur. Çalıştırma işlemi komut satırından yapılabilir. Ya da hiç komut satırına çıkmadan debug/start without debugging (ctrl+f5) ile yapılabilir. Eğer ctrl+f5 yapılırsa zaten hem derleme hem çalıştırma işlemi otomatik olarak gerçekleştirilir. Daha önce yaratılmış olan bir solution file/open/project-solution yoluyla açılabilir. Programcı, sln uzantılı dosyayı seçerek solution'ı açabilir.

PARAMETRE DEĞİŞKENLERİ

Bir fonksiyonun parametre değişkenleri fonksiyon tanımlanırken parametre parantezinin içinde bildirilir. Parametre parantezinin içine aralarına virgül konularak tür ve değişken isimleri verilir. Parametre değişkenlerine ilk değer verilemez.

Örneğin;

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

Parametre değişkenlerinin türleri aynı olsa bile tür belirten anahtar sözcük her defasında yazılmak zorundadır. Örneğin;

```
public static void func (int a , b) Error verir.
{
     //....
     //....
}
```

Parametreli bir fonksiyon çağırılırken parametre değişkeni sayısı kadar ifade yazılmak zorundadır.

Anahtar Notlar

Pek çok programlama dilinin standartlarında parametre ve argüman terimleri farklı anlamlarda kullanılmaktadır. Fonksiyon çağırılırken yazılan ifadelere argüman (argument), fonksiyonun parametre değişkenlerine parametre (parameter) denilmektedir. Halbuki kursumuzda argüman yerine parametre, parametre yerine parametre değişkeni kullanılmaktadır.

```
public static void Func(int a, int b)
{
     //…
}
//…
Func(100 + 300 , 200 + 500);
```

Parametreli bir fonksiyon çağırıldığında sırasıyla şunlar olur;

1. **Parametrelerin değerleri hesaplanır. (100+300 gibi)**
2. **Fonksiyonun parametre değişkenleri yaratılır.**
3. **Parametrelerden parametre değişkenlerine karşılıklı atama yapılır.**
4. **Akış fonksiyona geçirilir.**

Parametre değişkenleri, fonksiyon çağırıldığında yaratılmakta ve fonksiyon sonlandığında da yok edilmektedir.

```
public static void Main()
{
     int a = 10 , b = 20 , result;
     result = Add(a + 1 , b + 1);
     System.Console.WriteLine(result);
}
public static int Add(int a, int b)
{
     return a + b;
}
```

Görüldüğü gibi parametreli fonksiyonların çağırılması da gizli bir atama işlemine yol açmaktadır.

System.Math Sınıfı: Math sınıfının temel matematiksel işlemleri yapan pek çok faydalı fonksiyonu vardır.

- **sqrt fonksiyonu double bir parametre alır ve geri dönüş değeri de double türdendir. Parametresi ile aldığı değerin karekökünü geri döner.**

```
     public static void Main()
     {
          Double a = 100 , result;
          Result = System.Math.Sqrt(a);
```

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

- **Pow fonksiyonu birinci parametresi ile belirtilen değerin ikinci parametresi ile belirtilen kuvvetini alır ve bu değeri geri döner.**
- **Sin, cos, tan, asin, acos, atan fonksiyonları trigonometrik işlemler yapar. Parametre olan açılar radyan cinsindendir.**
- **Abs fonksiyonları (absolute value) mutlak değer almakta kullanılır.**

Deyimler(statement): Bir C# programı kabaca isim alanlarından , isim alanları sınıflardan, sınıflar veri elemanları ve fonksiyonlardan, fonksiyonlar da deyimlerden oluşur. Fonksiyonlar deyimlerden oluşur, deyimler doğal dillerdeki cümleler gibidir.

Deyimler, aşağıdaki gibi sınıflandırılabilir;

1. **Yalın Deyimler (Simple Statement): Bir ifadenin sonuna ; konulursa bu ifade deyim olur. Böyle deyimlere basit deyim denir. örneğin; a=b+c; görüldüğü gibi ifade kavramı ; içermez.**

2. **Bileşik Deyimler (Compound statement): bir blok içerisine yerleştirilmiş 0 ya da daha fazla deyime bileşik deyim denir. Yani bloklar bileşik deyim belirtmektedir. Örnek:**

```
{
X=10;
   {
   Y=20;
   Z=30;
   }
}
```

Burada bloğun tamamı tek bir deyimdir. Bu birleşik deyim tek bir deyim içermektedir. Görüldüğü gibi, bir deyim başka deyimleri içerebilir.

3. **Bildirim Deyimleri (Declaration statement): Bildirim yapmakta kullandığımız ifadeler de bir deyim oluşturur. Örneğin; int a, b;**
4. **Kontrol Deyimleri(Control Statement) : Programın akışı üzerinde etkili olan if, for, while gibi anahtar sözcüklerle oluşturulan cümleler de birer deyimdir.**
5. **Boş deyim (null statement) : Sonunda bir ifade olmadan kullanılan ;'lere boş deyim denir. örneğin: x=10;;; (en sondaki iki adet ; boş deyimdir) boş deyimin karşısında derleyici bir şey yapmaz. Fakat boş deyim de bir deyim olarak değerlendirilir.**

IF Deyimi : If deyiminin genel biçimi şöyledir;

```
if (<bool türden ifade>
      <deyim>
   [else <deyim>]
```

Derleyici, if anahtar sözcüğünden sonra parantezler içerisinde bool türden bir ifade bekler. If deyiminin doğruysa ve yanlışsa kısmında tek bir deyim bulunmak zorundadır.

Eğer doğruysa ve yanlışsa kısmında birden fazla deyim varsa programcı bloklama yaparak bunları tek deyim haline getirmek zorundadır.

Anahtar Notlar

C/C++'da if parantezi içerisindeki ifade herhangi bir türden olabilir.

If deyimi şöyle çalışır: **Derleyici if parantezi içerisindeki ifadenin değerini hesaplar. Bu değer true ise if deyiminin doğruysa kısmındaki deyimi çalıştırır, yanlışsa kısmını atlar. False ise yanlışsa kısmındaki değeri çalıştırır.**

Anahtar Notlar

Klavyeden T türünden bir bilgi okumak için aşağıdaki ifade kullanılabilir:

```
T.Parse(System.Console.Readline())
```

```
puclic static void Main()
{
     int val;
     val = int.Parse(System.Console.ReadLine());
     System.Console.WriteLine(val);
}
```

```
puclic static void Main()
{
     int val;
     val = int.Parse(System.Console.ReadLine());

     if(val > 10)
          System.Console.WriteLine("Evet");
     else
          System.Console.WriteLine("Hayır");

}
```

If deyiminin else kısmı bulunmak zorunda değildir. Derleyici, if deyiminin doğruysa kısmından sonra else anahtar sözcüğünün gelip gelmediğine bakar. Eğer else anahtar sözcüğü, gelmemişse if deyiminin bittiğini düşünür.

If deyiminin yanlışlıkla ; ile kapatılması hatası ile karşılaşılmaktadır. Örneğin;

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

Burada artık if deyiminin doğruysa kısmında boş deyim vardır. Dolayısıyla, "ifade2;" if deyiminin dışındadır.

İç içe if deyimleri: İf deyiminin tamamı, doğruysa ve yanlışsa kısmı ile birlikte tek bir deyimdir. Örneğin;

```
if ifade1
     if ifade2
          {
          ifade1;
          ifade 2;
          }
     else
          {
          ifade3;
          ifade 4;
          }
else
     ifade7;
     ifade8;
```

İki if'e karşılık tek bir else'in bulunduğu aşağıdaki durumda else'in içteki if'e ilişkin olduğu kabul edilmektedir.

İf(ifade1) if(ifade2) ifade3; else ifade4;

Yani bu if cümlesi, şu anlama gelmektedir.

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

Eğer else kısmının dıştaki if'e ilişkin olması isteniyorsa bilinçli olarak bloklama yapılması gerekir.

```
if(ifade1)
     {
     if (ifade2)
```

```
          ifade3;
     }
else
     ifade4;
```

Bir karşılaştırma doğruyken diğerinin doğru olma olasılığı yoksa bu iki karşılaştırmaya ayrık (discreet) karşılaştırmalar denir. Örneğin;

```
a > 10
a < 0
```

Karşılaştırmaları ayrıktır.

Örneğin;

```
a == 1
a == 2
```

Karşılaştırmaları ayrıktır.

Fakat

```
a > 0
a > 10
```

Karşılaştırmaları ayrık değildir.

Ayrık karşılaştırmaların ayrık if deyimleri ile ifade edilmesi kötü bir tekniktir. Örneğin;

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

Ayrık karşılaştırmaların else – if biçiminde ifade edilmesi gerekir.

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

Bazen ayrık karşılaştırmaların sayısı fazla olduğu için bir else-if merdiveni söz konusu olabilir. Örneğin;

```
if(a == 1)
     ifade1;

else if (a == 2)
     ifade2;
     else if(a == 3)
          ifade3;
          else if (a == 4)
```

```
                ifade4;
```

System.Char isimli sınıfın char bir değeri parametre olarak alıp onu test eden çeşitli isXXX (is ile başlayan) fonksiyonları vardır. Bu fonksiyonların geri dönüş değeri bool türündendir.

IsUpper; büyük harf mi testini,
IsLower; küçük harf mi testini,
IsDigit; sayısal bir karakter mi testini,
IsLetter; harf mi testini,
IsWhiteSpace; boşluk karakterlerinden biri mi testini yapar.

```
public static void Main()
{
     char ch;
     ch = char.Parse(System.Console.ReadLine());

     if(System.Char.IsUpper(ch))
          System.Console.WriteLine("Büyük Harf");
     else if(System.Char.IsLower(ch))
          System.Console.WriteLine("Küçük Harf");
     else if(System.Char.IsDigit(ch))
          System.Console.WriteLine("Digit");
     else if(System.Char.IsWhiteSpace(ch))
          System.Console.WriteLine("Boşluk Karakteri");
     else
          System.Console.WriteLine("Diğer");
}
```

If deyiminin doğruysa kısmı olmak zorundadır. Yalnızca yanlışsa kısmı olan bir if yapay olarak şöyle sağlanabilir;

```
if (ifade)
     ;
else
     ifade1;
```

Bunun yerine, soru ifadesini aşağıdaki gibi değiştirmek daha uygundur.

```
if (!ifade)
ifade1;
```

Döngü Deyimleri: Bir program parçasının yinelemeli (literatif) olarak çalıştırılmasını sağlayan deyimlere döngü deyimleri denir.

C#'da üç tür deyim vardır.

1- While döngüleri

a- Kontrolün başta yapıldığı while döngüleri
b- Kontrolün sonda yapıldığı while döngüleri

2- For döngüleri

3- For Each döngüleri

Anahtar Notlar
C/C++'da for each döngüsü yoktur. For each döngüleri diziler konusundan sonra ele alınacaktır.

Kontrolün Başta Yapıldığı While Döngüleri:

While döngüleri bir koşulun doğru olduğu sürece yinelenen döngülerdir. Genel biçimi şöyledir;

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

while anahtar sözcüğünden sonra parantezlerin içinde bool türden bir ifade bulunmak zorundadır. Döngünün içinde tek bir deyim vardır. Eğer birden fazla deyimin döngü içerisinde bulundurulması isteniyorsa bloklanması lazımdır.

while döngüsü şöyle çalışır: **while parantezi içerisindeki ifadenin değeri hesaplanır eğer bu değer true ise döngü deyimi çalıştırılır ve başa dönülür, false ise programın akışı döngü dışındaki ilk deyimle devam eder.**

```
public static void Main()
{
     int i = 0;

     while (i < 10)
     {

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

Ekran Çıktısı:

```
0
1
2
3
4
5
6
7
8
9
```

While parantezinin içerisinde son ek olarak bir ++ ya da -- varsa değişkenin artırılmamış ya da eksiltilmemiş hali test işlemine girer.

```
while (i++ < 10)
{
     System.Console.WriteLine
(i);

}
```

```
while (i >= 0)
{
     System.Console.WriteLine(i);
     --i;
}
```

Döngülerin yanlışlıkla boş deyimle kapatılması durumuyla sık karşılaşılmaktadır. Örneğin;

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

Burada döngü boş deyimle oluşmaktadır ve dolayısıyla sonsuz döngü oluşur. Kırmızı ile işaretli olan boş deyimdir ve o gerçekleştirilir. Sonsuz döngü şöyle kurulabilir;

```
while (true)
{
     // .....
}
```

Kontrolün sonda Yapıldığı While Döngüleri: Bu döngülere do-while döngüleri de denilmektedir.

Genel biçimi:

```
Do
     <deyim>

While (<bool türden bir ifade>);
```

Buradaki ; işareti bir boş deyim değil syntax'ın bir parçasıdır.

Örnek:

```
int=0;

do
{
     System.Console.WriteLine(i);
     ++i;
}
while (i<10);
```

Kontrolün sonda yapıldığı while döngüleri çok seyrek kullanılmaktadır.

```
char ch;

do
{
     System.Console.WriteLine(" (E)vet / (H)ayır ");
     ch = char.Parse(System.Console.ReadLine());
}
while (ch != 'e' && ch != 'h');
```

For Döngüleri: For döngüleri işlevsel olarak while döngülerini kapsamaktadır. En sık kullanılan döngülerdir.

Genel biçimleri:

```
For  ([ifade1];[ifade2(bool)];[ifade3])
     <deyim>
```

Derleyici for anahtar sözcüğünden sonra ";;" bekler. Bu iki noktalı virgül for döngüsünü üç kısma ayırır. Bir ve üçüncü kısımdaki ifadeler herhangi bir türden olabilir fakat ikinci kısımdaki ifade bool türden olmak zorundadır.

For döngüsü şöyle çalışır:

```
int i;

for (i = 0 ; i < 10 ; ++i;)
```

Birinci kısımdaki ifade döngüye girişte yalnız bir kez yapılır ve bir daha hiç yapılmaz. Döngü, ikinci kısımdaki ifade true olduğu sürece devam eder. Üçüncü kısımdaki ifade, döngü deyimi çalıştırıldıktan sonra her yinelemede başa dönerken çalıştırılmaktadır.

```
public static void Main()
{
     int i;

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

Asal sayı bulmak için en basit fakat etkin yöntemlerden biri şöyledir: For döngüsünün birinci kısmı döngünün yukarısına alınırsa eşdeğerlik bozulmaz. Örneğin:

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

Döngünün üçüncü kısmı yazılmayıp döngü deyiminden sonraya alınabilir:

```
i = 0;

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

Döngü için iki noktalı virgül mutlaka lazımdır. Boş bile olsa ; konulur.

```
for (; i>10; ++i)
{

}
```

Birinci ve üçüncü kısmı olmayan for döngüsü while döngüsü ile eşdeğerdir.

Örneğin;

```
for (; ifade;)
{
     //...
}
```

ile

```
while (ifade)
{
     //...
}
```

eşdeğerdir.

For döngüsünün ikinci kısmı yazılmazsa koşulun her zaman sağlandığı kabul edilir. Yani sonsuz döngü oluşur. Örneğin;

```
for (i = 0 ; ; ++i) //sonsuz döngü
{
     //...
}
```

ile

```
for (i = 0 ; true ; ++i) //sonsuz döngü
{
     //...
}
```

aynıdır.

Nihayet for döngüsünün hiçbir kısmı olmayabilir. Fakat ; 'ün her zaman bulunması gerekir. Örneğin;

```
for ( ; ; ) // sonsuz döngü
{
     //...
}
```

For döngüsünün birinci kısmında (fakat diğer kısımlarında değil) bildirim yapılabilir. Örneğin;

```
for (int i = 0 ; i < 100;++i)
{
     //..
}
```

For döngüsünün birinci kısmında belirtilen değişkenlerin faaliyet alanı sadece döngü bloğunun içerisini kapsar. Yani;

```
for (int i = 0 ; i < 100 ; ++i)
{
     //…
}
```

İş**leminin eşdeğeri;**

```
int i;
for (i = 0 ; i < 100 ; ++i)
{
     //..
}
```

biçimindedir.

Bu eşdeğerlikten hareketle, aynı blok içerisinde aşağıdaki gibi iki for döngüsünün bulunması soruna yol açmaz.

```
for (int i = 0 ; i < 100 ; ++i)
{
     //...
}
```

```
for (int i = 0 ; i < 100 ; ++i)
{
     //...
}
```

For döngüsünün en çok kullanıldığı kalıp şöyledir;

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

Fakat döngünün bölümleri bu biçimde bir kalıp izlemek zorunda değildir.

```
for (System.Console.Writeline("Bir") ; i<10 ;
System.Console.Writeline("üç"))
{
     //..
}
```

gibi de olabilir

Anahtar Notlar

Bir değeri bir değişkene atayıp sonucu karşılaştırmak için atama işlemine öncelik vermek amacıyla parantez kullanmak gerekir. Örneğin;

```
while ((val = func()) !=0)
{
//..
}
```

Burada func fonksiyonunun geri dönüş eğeri val değişkenine atanıp 0 ile karşılaştırılmıştır. Eğer parantezler olmasaydı karşılaştırmanın sonucu olan bool değer val değişkenine atanırdı.

Bir döngünün içerisinde başka bir döngü olabilir. Örneğin;

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

Anahtar notlar:
Console sınıfının write ve writeline fonksiyonlarının daha kullanışlı bir versiyonu vardır.

Bu fonksiyonlar iki tırnak içindeki karakteri ekrana yazar. Fakat küme parantezlerini gördüklerinde {n} karakterlerini ekrana yazmazlar. Burada n bir sayıdır ve parametre indeksi belirtmektedir. Bu Küme parantezleri bir yer tutucudur. Bu karakterler yerine n'inci indeksteki parametrenin değeri yazılır. İki tırnaktan sonraki ilk parametrenin indeksi sıfırdır.
System.Console.WriteLine("((0) , (1))",i ,k);

Break deyimi: Break deyiminin kullanılabilmesi için bir döngünün içerisinde ya da switch deyiminin içerisinde olmak gerekir. Kullanımı şöyledir:

```
break;
```

Programın akışı break anahtar sözcüğünü gördüğünde akış döngünün dışındaki ilk deyimle devam eder.

Bazen döngüden çıkış pek çok koşula bağlı olabilir. Tüm koşulları while ya da for parantezi içerisinde değerlendirmek iyi bir teknik değildir. Belki en önemli koşul while ya da for parantezinde belirtilebilir ve diğer koşullar döngünün içerisinde if ve break deyimleri ile sorgulanabilir.

continue DEYİMİ

Continue deyimi yalnızca döngüler içerisinde kullanılabilir. Kullanım biçimi şöyledir:

```
continue;
```

Akış continue anahtar sözcüğünü gördüğünde sanki döngü deyimi bitmiş de yeni bir yinelemeye geçiyormuş gibi bir etki söz konusudur. Yani while döngülerin de continue kullanılırsa akış hemen döngünün başına döndürülerek yeni bir yinelemeye geçilir, for döngüsünde continue kullanılırsa döngünün üçüncü kısmı yapılarak başa dönülür.(üçüncü kısım arttırma veya azaltmanın yapıldığı kısım)

```
public static void Main()
{
     for (int i = 0 ; i < 10 ; ++i)
     {
          if (i % == 0)
               continue;
          System.Console.WriteLine(i);
     }
}
```

goto DEYİMİ

goto deyimi programın akışını koşulsuz olarak fonksiyon içerisinde istenilen bir noktaya aktarır. Genel biçimi şöyledir:

```
goto <etiket>;
//…
etiket: <deyim>
```

Programın akışı goto anahtar sözcüğünü gördüğünde etiketle belirtilen noktaya atlar. Goto ile akış başka bir fonksiyona atlatılamaz. Aynı fonksiyonda yukarıya ya da aşağıya atlatılabilir.

C# standartlarına göre goto etiketinden sonra en az bir deyimin bulunması gerekmektedir.

Anahtar Notlar:

Consol sınıfının ReadKey isimli fonksiyonu klavyeden bir tuşa basılana kadar bekleme oluşturur.

goto deyimi özellikle iç içe döngülerden çıkmak için, döngü içindeki switch deyiminden tek hamlede çıkmak için ya da ters sırada boşaltım yapmak için kullanılır. goto deyimi ile blok içlerine atlama yapılmaz.

SABİT İFADELERİ

Yalnızca sabit ve operantlardan oluşan ifadelere sabit ifadeler(constant expressions) denir. Örneğin:

```
3 , 3+2 , 5+2-3
```

birer sabit ifadedir.

Sabit ifadelerin net sayısal değerleri derleme aşamasında hesaplanır.

switch DEYİMİ

switch deyimi bir ifadenin çeşitli sayısal değerleri için farklı işlemler yapılması amacıyla kullanılmaktadır. Genel biçimi şöyledir:

```
switch (<ifade>)
{
     case <sabit ifade> :
          [deyim listesi]
          break;

     case <sabit ifade> :
          [deyim listesi]
          break;

     //…

     default :
          [deyim listesi]
          break;

}
```

switch deyimi şöyle çalışır: Derleyici switch parantezi içerisindeki ifadenin sayısal değerini hesaplar. Bu değer ile tamamen aynı olan bir case bölümü araştırır. Bulursa akışı bir goto deyimi gibi ilgili case bölümüne atlatır.(Yani akış : 'nin sağına atlatılır) Akış o noktadan deyimleri çalıştırarak devam eder, break anahtar sözcüğü görüldüğünde akış switch dışındaki ilk deyimle devam eder. Eğer switch parantezi içerisindeki ifadeye tam eşit bir case bölümü bulunamazsa ve switch deyiminin default bölümü varsa akış default bölümüne aktarılır. Eğer default bölümü de yoksa akış switch dışındaki ilk deyimle devam eder.

```
public static void Main()
{
     int val;
     val  =  int.Parse(System.Console.ReadLine()); à  Klavyeden bilgi okuma
switch(val)
{
     case 1:
          System.Console.WriteLine("Bir");
          break;

     case 2:
          System.Console.WriteLine("İki");
          break;

     case 3:
          System.Console.WriteLine("Üç");
```

```
          break;

     default:
          System.Console.WriteLine("Diğer");
          break;
} à switch sonu
System.Console.WriteLine("Son…");
}
```

Aynı değere ait birden fazla case deyimi olamaz. Case bölümlerinin sıralı olması ya da default bölümünün sonda olması zorunlu değildir.

Case ifadeleri tamsayı türlerine ilişkin değer olmalıdır. Gerçek sayı türlerine ilişkin olamaz.(3,2 - 5,4 gibi) Benzer biçimde switch parantezi içindeki ifadenin de tamsayı türlerine ilişkin olmalıdır. Ancak istisna olarak switch içindeki case ifadelerinde string kullanılabilir.

C#'ta aşağıya düşme(fall trough) özelliği yoktur. Akışın bir case bölümünde diğerine geçiyor olması error oluşturur. Akışın bir case bölümünden diğerine geçmemesini sağlamanın en pratik yolu her case bölümünün sonuna break yerleştirmektir.

Akışın aşağı düşmesini engellemenin tek yolu break kullanmak değildir. Örneğin aşağıdaki gibi bir sonsuz döngü de bu gereksinimi sağlar:

```
case 1:
     //…
     for( ; ;) à Geçerli
     ;

case 2:
     //…
```

Aynı amaçla goto da kullanılabilir:

```
case 1:
     //…
     goto EXIT; à Geçerli

case 2:
     //…
```

Ya da return deyimi de aynı görevi yapar:

```
case 1:
     //…
     return; à Geçerli

case 2:
     //…
```

Bazen farklı case ifadeleri için aynı şeylerin yapılması istenebilir. Bunu sağlamanın en pratik yolu aşağıdaki gibi yapı kullanmaktır. Daha pratik bir yolu yoktur.

```
case 1:
case 2:
     //…
     break;
```

C# standartlarına göre eğer bir case bölümünde hiçbir deyim bulundurulmamışsa akışın aşağıdaki case bölümüne düşmesi geçerlidir. Fakat bir case bölümünde en az bir deyim varsa akış aşağıya düşmemelidir. Örneğin:

```
case 1: ; à error
case 2:
     //…
     break;
```

Yine de bazen bir case bölümünde bir takım şeyler yapıldıktan sonra başka bir case bölümündeki işlemlerinde yapılması istenebilir. Bu işlem C/C++'da aşağıya düşme özelliği ile sağlanabilmektedir. İşte bunu C#'da sağlamak için goto case deyimi eklenmiştir. Goto case deyimi switch içerisinde kullanılabilir ve akışı başka bir case bölümüne aktarılır. Örneğin:

```
case 1:
     //…
     goto case 2;
case 2:
     //…
     break;
```

Burada case 1 bölümü çalıştırıldıktan sonra case 2 bölümü de çalıştırılacaktır. goto case deyimi gibi goto default deyimi de vardır.

FARKLI TÜRLERİN BİRBİRLERİNE ATANMASI

C#'ta x=y gibi bir atamada atanan ve atanılan türler farklı olabilir. Bilgi kaybı oluşturmayan atamalar geçerli, bilgi kaybı oluşturabilecek atamalar geçersizdir ve error oluşturur. Başka bir deyişle küçük türlerden büyük türlere doğrudan atama geçerli fakat büyük türlerden küçük türlere doğrudan atama geçersizdir. Büyük tür küçük türlere yapılan atamalara büyük türün içerisindeki sayıya bakılmamaktadır.
Yukarıdaki atama kuralı özet bir biçimde belirtilmiştir. Bazı ayrıntıları vardır:

1. **Küçük işaretli tamsayı türünden büyük işaretsiz tamsayı türüne doğrudan atama yapılmaz. Örneğin: short türünden uint türüne atama yapılmaz ya da int türü ulong türüne atanamaz. Çünkü büyük işaretsiz tür negatif sayıları tutamamaktadır. Tabi şüphesiz küçük işaretsiz tamsayı türünden büyük işaretli tamsayı türüne atama yapılabilir.**

2-Tüm tamsayı türlerinden tüm gerçek tamsayı türlerine doğrudan atama yapılabilir fakat herhangi bir gerçek sayı türünden herhangi bir tamsayı türüne atama yapılmaz.

Anahtar Notlar:

Şüphesiz bir long sayı hatta bir int sayı float türüne atadığımız zaman bir kayıp söz konusudur. Float türü, bu sayı ne kadar büyük olursa bir mantis hatasıyla basamak kaybı yapmadan tutabilmektedir. Bu kayıp C#'ta mazur görülmüştür.

3-Her ne kadar decimal türü 16 byte uzunluğunda geniş bir türse de bu tür 28 digitlik bir sayıyı yuvarlama hatasız tutmak için düşünülmüştür. Float ve double içerisindeki sayılar decimal türü ile ifade edilmeyebilir. Bu nedenle float ve double türünden decimal türüne atama yasaklanmıştır. Decimal türünden de float ve double türlerine atama yapılamaz.

4-C#'ta bool türüne herhangi bir türden atama yapılamaz. Bool bir değerde herhangi bir türe atanamaz.

Yukarıda açıklanan kurallar aslında aşağıdaki gibi bir tablo ile açık bir şekilde belirtilebilir:

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

Anımsanacağı gibi C#'ta byte, sbyte, short, ushort türden sabit yoktur. Peki, bu türlere nasıl değer atanacaktır? İşte bunu mümkün hale getirmek için şu ek kurallar oluşturulmuştur:

1- İnt türünden bir sabit ifadesi hedef türün sınırları içerisinde kalmak koşuluyla byte, sbyte, short, ushort türlerine doğrudan atanabilir. Örneğin:

```
sbyte s;
s = 100 à geçerli
s = 100 + 10; à geçerli
s = 300; à error
```

Benzer biçimde int türden bir sabit ifadesi, belirtilen sayı hedef türün sınırları içinde kalmak koşuluyla uint ve long türlerine doğrudan atanabilir. Örneğin:

```
uint a = 1 à geçerli

uint a = -1 à error
```

2- Long türden bir sabit ifadesi belirtilen sayı hedef türün sınırları içerisinde kalmak koşuluyla ulong türüne doğrudan atanabilir.

İŞLEM ÖNCESİ OTOMATİK TÜR DÖNÜŞTÜRMELERİ

Yalnızca değişken ve sabitlerin değil ifadelerinde toplam bir türü vardır.

C# derleyicisi bir operatörle karşılaştığında önce operantların türlerini araştırır. Eğer operantlar aynı türdense işlemi hemen yapar. İşlem sonrasında elde edilen değer bu ortak türden çıkar. Eğer operantlar farklı türlerden ise önce operantlar aynı türe dönüştürülür, sonra işlem yapılır. İşlem sonucunda elde edilen değer dönüştürülen bu ortak tür türünden olur. Dönüştürme kuralının özeti küçük türün büyük türe dönüştürülmesi şeklindedir.

Küçük türün büyük türe dönüştürülmesi sırasında derleyici büyük türünden geçici bir değişken yaratır. Sonra küçük değişkeni bu değişkene atar. İşlemde bu değişkeni kullanır ve işlem sonunda bu geçici değişkeni yok eder. Örneğin:

```
int a = 10, b;

b = a + 10L; à error
```

Örneğin C#'ta iki int değer bölünürse sonuç int türünden çıkar:

```
double d;

d = 10 / 3;

Sonuç: d = 3.0
```

Küçük türün büyük türe dönüştürülme kuralının ayrıntıları şöyledir:

1- Tamsayı türleri ile gerçek sayı türleri işleme sokulduğunda dönüştürme her zaman gerçek sayı türüne doğru yapılır.

2- Küçük işaretli türden büyük işaretsiz türe dönüştürme mümkün olmadığı için bu iki tür bir arada işleme sokulamaz. Örneğin:

```
short a = 10;
ulong b = 20, c;
c = a + b; à error
```

3- Float ve double türünden decimal türüne dönüştürme olmadığına göre float ve double türü ile decimal türü birlikte işleme sokulamaz.

4- int türünden küçük olan byte, sbyte, short, ushort, char türleri kendi aralarında işleme sokulursa önce her iki operantta int türüne dönüştürülür. Sonuç int türünde çıkar.

```
short s = 10 , b = 20 , c;

c = a + b;  à error (int short türüne dönüştürülmez)
```

5- bool türü hiçbir türle işleme sokulmaz.

TÜR DÖNÜŞTÜRME OPERATÖRÜ

Tür dönüştürme operatörü tek operantlı önek bir operatördür. Genel kullanımı şöyledir:

```
tür (operant)
```

Tür dönüştürme operatörü öncelik tablosunun ikinci sırasında sağdan sola bulunur:

()	**soldan sağa**
+ - ++ -- : (tür)	**sağdan sola**
*** / %**	**soldan sağa**
+ -	**soldan sağa**
……	

Örneğin:

```
int a = 10 , b = 3;

double d;

d = (double) a / b;
```

Yukarıda önce a double türüne dönüştürülür. Sonra bölme işlemi yapılır. Bu durumda b zaten otomatik olarak double türüne dönüştürülecektir. Sonuç double türünden çıkar.

Tür dönüştürme işlemi sırasında önce dönüştürülecek tür türünden geçici bir değişken oluşturulur sonra dönüştürülecek ifade bu geçici değişkene atanır. İşlemde bu geçici değişken kullanılır sonra bu geçici değişken yok edilir.

Anahtar Notlar:

C# standartlarına göre atama işlemi aslında otomatik tür dönüştürmeye yol açmaktadır. Atama işleminde önce sağ taraf sol tarafın türüne otomatik dönüştürülür sonra atama yapılır. Standartlarda atama işleminde yapılan bu otomatik dönüştürmelere "implicit" dönüştürme, tür dönüştürme operatörü ile yapılan dönüştürmelere "explicit" dönüştürme denir.

Görüldüğü gibi büyük türden küçük türe doğrudan atamalar yanlışlıkla yapılmasın diye yasaklanmıştır. Yoksa programcı istedikten sonra tür dönüştürme operatörü ile dönüştürmeyi yapabilir.

Şüphesiz dönüştürme sırasında eğer büyük türün belirttiği sayı küçük türün sınırları içerisinde kalıyorsa zaten bilgi kaybı söz konusu olmaz. Fakat kalmıyorsa bilgi kaybı oluşur. Bilgi kaybının oluşma biçimi şu maddelerde özetlenebilir:

1- **Gerçek sayı türlerinden tamsayı türlerine yapılan dönüştürmelerde sayının noktadan sonraki kısmı atılır tam kısmı elde edilir.**

2- **Büyük tamsayı türünden küçük tamsayı türüne yapılan dönüştürmelerde sayının yüksek anlamlı byte değeri atılır. Düşük anlamlı byte değeri alınır. Eğer büyük tamsayı içerisindeki sayı küçük tamsayı türünün limitleri arasında kalmıyorsa dönüştürme işleminden sonra sayı gerçeği ile ilgisiz bir duruma düşebilir.**

3- **Aynı türün işaretli ve işaretsiz versiyonları arasında dönüştürme yapıldığında sayının bit kalıbında değişiklik olmaz. Yalnızca işaret bitinin yorumu değişir.**

4- **Küçük işaretli türden büyük işaretsiz türe dönüştürmeler iki aşamalı yapılır. 1. aşamada sayı büyük türün işaretli biçimine dönüştürülür. 2. aşamada büyük türün işaretli biçiminden büyük türün işaretsiz biçimine dönüştürme uygulanır.**

İŞLEMLİ ATAMA OPERATÖRLERİ

C#'ta +=, -=, *=, %=biçiminde bir grup işlemli atama operatörü vardır. İşlemli atama operatörleri iki operantlı araek operatörlerdir.

```
a < op > = b

     ile

a = a < op > b ile tamamen eşdeğerdir.
```

İşlemli atama operatörleri öncelik tablosunun solunda atama operatörü ile sağdan sola aynı grupta bulunur:

=, +=, -=, *=, %= sağdan sola

KOŞUL OPERATÖRÜ

Koşul operatörü C#'ın tek üç operantlı operatörüdür. Kullanımı şu şekildedir:

```
< bool türden ifade1 > ? < ifade2 > : < ifade3 >
```

Koşul operatörü adeta if deyimi gibi çalışan bir operatördür. Ancak koşul operatörü bir değer üretmektedir.

Koşul operatörünün birinci operantı bool türden olmak zorundadır. Önce ? 'nin solundaki ifadenin değeri hesaplanır. Eğer değer true ise ? ile : arasındaki ifade yapılır, false ise : 'nin sağındaki ifade yapılır. Koşul operatörü duruma göre ? ile : arasındaki yada : 'nin sağındaki ifadenin değerini üretir.

Şüphesiz koşul operatörü ile yapılan her şey aslında if deyimi ile de yapılabilir. Koşul operatörü if deyimi gibi kullanılmamalıdır. Bir karşılaştırma sonrasında elde edilen değerin işleme sokulacağı durumlarda tercih edilmelidir.

```
     int val , result;
     System.Consol.Write("Sayı Giriniz: ");
     val = int.Parse(System.Console.ReadLine()); à Klavyeden bilgi okuma kalıbı

     result = val > 10 ? 100 : 200;

     System.Console.Write(result)
```

Koşul operatörünün kullanılmasının salık verildiği tipik durumlar şunlardır:

1- **Karşılaştırma sırasında elde edilen değerin bir değişkene atandığı durumlar. Örneğin:**

```
total = 0;

for (int i = 1900 ; i < 2007 ; ++i)
     total += System.DateTime:IsLeapYear(i) ? 366 : 365;
```

Burada 1900 yılından 2007 yılına kadar toplam geçen gün sayısı hesaplanmıştır. Aynı şekilde if ile de şu şekilde yapılır:

```
total = 0;

for (int i = 1900 ; i < 2007 ; ++i)
     if (System.DateTime:IsLeapYear(i))
            total += 366;
     else
            total += 365;
```

2- **Koşul operatörü return işleminde de güzel bir biçimde kullanılabilir. Örneğin:**

```
public static int GetMAx(int a, int b)
{
     return a > b ? a : b;
}
```

Burada a ile b'den hangisi büyükse o değer ile geri dönülmektedir. Aynı işlem şöyle de yapılabilir:

```
public static int GetMAx(int a, int b)
{
     if (a > b)
                return a;
        else
                return b;
}
```

3- Fonksiyon çağırma işleminde koşul operatörü kullanılabilir. Örneğin:

```
Func (a > b ? a : b);
```

Burada fonksiyonun tek bir parametre değişkeni vardır.İşlem aşağıdaki ile eşdeğerdir:

```
if (a > b);
     Func(a);
else
     Func(b);
```

Parantezler kullanılarak bazı operatörler koşul operatörünün operantı olmaktan çıkarılabilir. Örneğin:

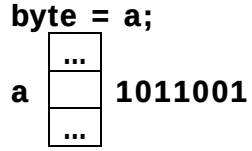
```
int a = 100 , b;
b = (a > 0 ? 100 : 200) + 300;
```

Burada önce parantez içi yapılır. Koşul doğru da olsa yanlış da olsa 300 ile toplama yapılır.

ADRES KAVRAMI

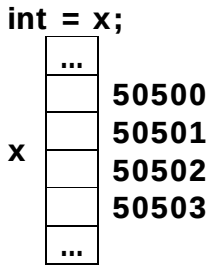
(17.07.2007 – Salı)

Bellek kayıtlarda oluşmuştur. Bellekteki her byte ilk byte 0 olmak üzere ardışık bir sayıya karşılık getirilmiştir. Bu sayıya ilgili byte'in adresi denmektedir. Tanımladığımız tüm değişkenler bir yer kaplamaktadır. Onların bir adresi vardır. Örneğin:



Burada a'nın adresi 1011001'dir.

Bir byte uzunluğunda olan değişkenlerin adresleri onların en düşük adresleri ile ifade edilir. Örneğin:



Burada x değişkenin adresi 50500'dür.

SINIFLARIN VERİ ELEMANLARI

Fonksiyonların dışında fakat sınıf bildirimleri içerisinde bildirilen değişkenlere sınıfın veri elemanları(field) denmektedir.

Sınıfın veri elemanları da static olabilir ya da olmayabilir. Sınıfın veri elemanlarında da erişim belirleyicisi anahtar sözcükler kullanılabilir. Bu anahtar sözcüklerden hiçbiri kullanılmasaydı *private* kullanılmış gibi varsayılmaktadır.(Daha önceden de belirtildiği gibi static anahtar sözcüğü ya da erişim belirleyicisi yerel değişkenlerde kullanılamaz)

Her sınıf aynı zamanda bir tür belirtir. Sınıf türlerinde değişken tanımlayabiliriz.

DEĞER TÜRLERİ VE REFERANS TÜRLERİ

Türler kategori olarak 2'ye ayrılmaktadır: 1- Değer Türleri(Value Types) 2- Referans Türleri(Reference Types) Bugüne kadar gördüğümüz int, long türleri kategori olarak değer türleridir. Değer türlerine ilişkin bir değişken, değerin kendisini tutar. Oysa referans türlerine ilişkin bir değişken, değerin kendisini tutmaz. Değerin kendisi bellekte başka bir yerdedir, değişken onun bellekteki adresinin tutar. Yani referans türlerine ilişkin bir değişken adres tutmaktadır. Değerin kendisi referansın tuttuğu adrestedir.

Sınıf türleri kategori olarak referans türlerine ilişkindir. Yani bir sınıf türünden değişken tanımlandığında bu değişkenin içerisine bir adres yerleştirilebilir.

SINIF TÜRLERİNDEN NESNELERİN YARATILMASI

Sınıf türlerinde değişkenlere referansta denir. Örneğin Sample bir sınıf olsun:

```
Sample s;
```

Burada s nesnenin kendisi değildir. Nesnenin adresini tutacak bir adrestir. Nesnenin kendisi new operatörü ile yaratılır. New operatörünün kullanımı şu şekildedir:

```
new < sınıf ismi > ()
```

Örneğin:

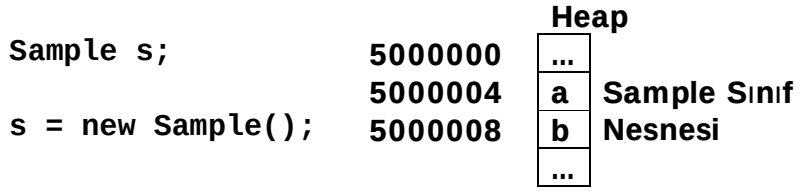
```
new Sample()
```

new operatörü ile tahsisat yapıldığında derleyici sınıfın static olmayan veri elemanları için ardışık bir biçimde belleğin heap denilen bölümünde tahsisat yapmaktadır.

Sınıflar bileşik türlerdir. Yani parçalardan oluşmaktadır. Bir sınıf nesnesinin parçaları static olmayan veri elemanlarıdır. Sınıfın fonksiyonları new ile tahsis edilen alanda yer kaplamaz. Sınıfın static veri elemanları new ile tahsis edilen alanda yer kaplamaz. Yalnızca sınıfın static olmayan veri elemanları yer kaplar.

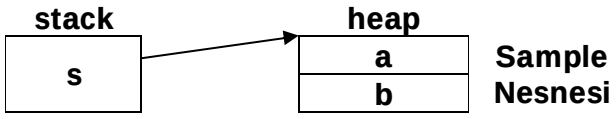
Bir bileşik nesnenin adresi onun ilk byte'nın adresi ile belirtilmektedir.

new operatörü heap'te tahsisat yaptıktan sonra yaratılan birleşik nesnenin adresini verir. new operatörü ile tahsis edilen nesnenin adresi aynı sınıf türünden referansa atanmalıdır. Böylece referans artık tahsis edilmiş sınıf nesnesini gösterir duruma gelmiştir.



Yerel değişkenler ister referans olsun isterse değer türlerine ilişkin olsun stack'te yer kaplar. Fakat new operatörü ile tahsis edilen sınıf nesneleri heap'te yer kaplamaktadır. Örneğin:

```
Sample s;

s = new Sample();
```

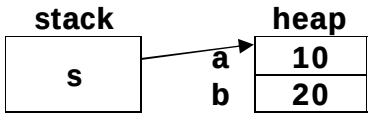


NOKTA OPERATÖRÜ

Nokta operatörü iki operantlı araek bir operatördür. Nokta operatörünün solundaki operant, bir sınıf alanı ismi, bir sınıf ismi olabileceği gibi bir sınıf türünden referansta olabilir. Nokta operatörünün solunda sınıf referansı varsa sağında o sınıfın static olmayan bir elemanı olmak zorundadır.

r bir sınıf türünden referans a bu sınıfın static olmayan bir veri elemanı olmak üzere `r.a` ifadesi r referansının gösterdiği yerdeki nesnenin a parçasını gösterir.

```
Sample s;
s = new Sample();

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



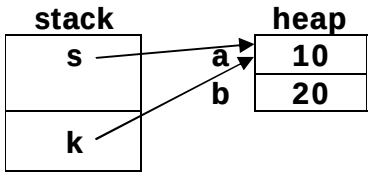
Burada s.a ve s.b int türdendir.

Genel Örnek:

```
class App
{
     public static void Main()
     {
          Sample s;
          s = new Sample();
          s.a = 10;
          s.b = 20;

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

     }
}
class Sample
{
     public int a;
     public int b;
     public int static int c;

     public void Foo()
     {
          //…
     }
     public static void Bar()
     {
          //…
     }
}
```

Aynı türden iki sınıf referansı birbirine atanabilmektedir. İki referans birbirine atandığında referansın içindeki adresler atanmaktadır. Bu durumda referanslar aynı nesneyi gösterir durumdadır.

```
Sample s;
s = new Sample();

Sample k;
k = s;
```



G.Örnek:

```
…
public static void Main()
{
      Sample s;
      s = new Sample();
      s.a = 10;
      s.b = 20;
      Sample k = s;

      System.Console.WriteLine(k.a);
      System.Console.WriteLine(k.b);
}
…
```

Her new işlemi heap'te yeni bir sınıf nesnesi yaratılmasına neden olmaktadır.

ÇÖP TOPLAYICI KAVRAMI

CLR(Common Language Runtime) her nesnenin belli bir anda kaç referans tarafından gösterildiğinin kaydını tutmaktadır. Buna nesnenin referans sayacı denmektedir. Örneğin:

```
                …
                {
                      Sample x = new Sample();
1 ──────────────►     {
                            Sample y;
2 ──────────────►           y = x;
                            {
                                  Sample z;
3 ──────────────►                 z = new Sample();
                            }
4 ──────────────►     }
5 ──────────────►}
6 ──────────────►…
```

Her new ayrı bir nesnenin yaratılmasına neden olmaktadır. Burada akış 1 noktasına geldiğinde heap'te yaratılan 1.nesneyi tek bir referans gösteriyor. Yani bu nesnenin referans sayacı 1'dir. Akış 2 noktasına geldiğinde 1.nesneyi 2 referans gösteriyor durumdadır. Akış 3 numaralı noktaya geldiğinde 1.nesneyi 2 referans, 2.nesneyi 1 referans gösteriyor durumdadır. Her nesnenin referans sayacı aynıdır. Akış 4 noktasına geldiğinde 1.nesneyi 2 referans, 2.nesneyi 0 referans gösteriyor durumdadır.(Bloktan çıkıldığı için). Akış 5 noktasına geldiğinde 1.nesneyi 1 referans, 2.nesneyi 0 referans gösteriyor durumdadır. Nihayet akış 6 noktasına geldiğinde her iki nesneyi de 0 referans gösteriyor durumdadır. (Bloklardan çıkıldıkça stack'taki nesneler yok olmaktadır).

CLR heap'teki yaratılan her nesnenin referans sayacı sürekli izlenmektedir. Nesnenin referans sayacı 0'a düştüğünde artık nesneyi hiçbir referans göstermiyor durumdadır. İşte CLR referans sayacı 0'a düşmüş nesneleri heap'ten otomatik olarak silmektedir. CLR'nin bu işi yapan kısmına kavramsal olarak çöp toplayıcı(Garbage Collector) denmektedir.

Referans sayacı 0'a düşmüş olan nesneler çöp toplayıcı için seçilebilir durumdadır. Fakat .Net standartları nesne seçilebilir duruma geldikten ne kadar süre sonra çöp toplayıcının bu nesneyi sileceği konusunda bir belirlemede bulunmamıştır. Yani nesne seçilebilir duruma gelir gelmez silinmek zorunda değildir. Fakat silinmeye aday bir durumdadır. Belli bir süre sonra çöp toplayıcı tarafında silinecektir. İyi bir çöp toplayıcıdan nesne seçilebilir duruma geldikten sonra nesneyi silmesi beklenir. Bazen çeşitli nedenlerden dolayı nesnenin silinmesi gecikebilir. Nesne silinmeden program bile sonlanabilir.

Görüldüğü gibi stack'taki değişkenler programın akışı değişkenin yaratıldığı bloktan çıkıldığında otomatik olarak yok edilmektedir. Bu yok edilme çok hızlı yapılır. Halbuki new operatörü ile heap'te tahsis edilen nesneler çöp toplayıcı tarafından göreceli olarak yavaş bir biçimde silinmektedir.

SINIFIN STATIC ELEMANLARI

19.07.2007 – Perşembe

Sınıfın static veri elemanları nesne yaratıldığında nesnenin içerisinde yer kaplamaz. Bunlarda toplamda tek bir kopya vardır.

Sınıfın static veri elemanlarından toplamda tek kopya bulunduğu için bunlara referanslarla değil sınıf isimleri ile erişilmektedir. Örneğin Sample bir sınıf ismi c de bunun static bir veri elemanı olsun. Erişim Sapmle.c ifadesi ile yapılmaktadır.

Sınıfın static veri elemanları o sınıfın türünden nesne yaratılsa da kullanılabilir. Örneğin:

G.Ö.

```
…
public static void Main()
```

```
{
     Sample.c = 10;
     System.Console.WriteLine(Sample.c);
}
…
```

SINIFIN STATİC OLMAYAN FONKSİYONLARI

Anımsanacağı gibi sınıfın static fonksiyonları eğer başka bir isim alanı içerisindeki sınıfa ait ise isim alanı ismi ve sınıf ismi belirtilerek, eğer aynı isim alanındaki sınıfa ait ise yalnız sınıf ismi belirtilerek, eğer aynı sınıfa aitse yalnızca fonksiyon ismi yazılarak erişilir.

Sınıfın static olmayan fonksiyonları o sınıf türünden bir referans ile nokta operatörü kullanılarak çağrılır. Örneğin: r bir sınıf türünden referans Func ise bu sınıfın static olmayan bir fonksiyonu olsun. Çağırma `r.Func(..)` ifadesi ile yapılmaktadır.

Sınıfın static olmayan veri elemanları static olmayan fonksiyonlar içerisinde doğrudan kullanılabilir. Fakat static fonksiyonlar içerisinde doğrudan kullanılamazlar. Örneğin:

```
…
class Sample
{
     public int a;
     public static int b;
     public static void Foo()
     {
          a = 10;    à error
     }
     public void Bar()
     {
          a = 10;    à geçerli
     }
}
…
```

Sınıfın static olmayan bir fonksiyonu çağrıldığında bu fonksiyonun kullandığı static olmayan veri elemanları fonksiyon hangi referansla çağrılmışsa o referansın gösterdiği yerdeki nesnenin veri elemanlarıdır. Örneğin:

```
…
class Sample
{
     public int a;
     public int b;
     public void Func(int x, int y)
     {
          a = x
          b = y
```

```
     }
     public void Disp()
     {
          System.Console.WriteLine(a);
          System.Console.WriteLine(b);
     }
}
…
```

```
(Main sınıfı olabilir)
…
Sample s = new Sample();

s.Func(10,20);
s.Disp();    à Ekran Çıktısı 10,20

Sample k = new Sample();
k.Func(30,40);
k.Disp();    à Ekran Çıktısı 30,40

k = s;
k.Disp();    à Ekran Çıktısı 10,20
…
```

Static olmayan fonksiyonun bir referans ile çağrılması bu fonksiyon içerisinde kullanılan static olmayan veri elemanlarının hangi nesnenin veri elemanları olduğunun tespit edilmesi için gerekmektedir. Static fonksiyonlar içerisinde sınıfın static olmayan veri elemanları kullanılmadığına göre static fonksiyonlar sınıf ismi ile çağrılması uygun görülmüştür.

Sınıfın static olmayan bir fonksiyonu başka bir static olmayan fonksiyonu doğrudan çağırabilir. (Aynı sınıf içinde). Bu durumda bu çağırmanın çağrılan fonksiyonda hangi referans kullanılmışsa o referansla yapıldığı varsayılır. Yani başka bir deyişle çağrılan fonksiyonda çağıran fonksiyon ile aynı nesnenin veri elemanlarını kullanmaktadır. Örneğin:

```
namespace CSD
{
     class App
     {
          public static void Main()
          {
               Sample s = new Sample();
               s.Func(10,20);
          }
     }
     class Sample
     {
          public int a;
          public int b;
          public void Func(int x, int y)
          {
               a = x;
```

```
                a = y;
                Disp();    à s.Disp(); gibi
          }
          public void Disp()
          {
                System.Console.WriteLine(a);
                System.Console.WriteLine(b);
          }
     }

}
```

Sınıfın static olmayan bir fonksiyonu sınıfın static veri elemanlarını doğrudan kullanabilir. Bu durumda static veri elemanının sınıf ismi ile kullanıldığı varsayılır. Sınıfın static olmayan fonksiyonu sınıfın static bir bir fonksiyonunu da doğrudan çağırabilir. Bu durumda çağırmanın da sınıf ismi ile yapıldığı varsayılır.

Fakat sınıfın static olan bir fonksiyonu static olmayan veri elemanlarını doğrudan kullanamaz ve static olmayan fonksiyonlarını doğrudan çağıramaz.

Yukarıda anlatılanların özeti şöyledir:

1- **Sınıfın static olmayan fonksiyonları, sınıfın hem static olmayan veri elemanlarını hem de static veri elemanlarını doğrudan kullanabilir.**

2- **Sınıfın static fonksiyonları, sınıfın static olmayan veri elemanlarını doğrudan kullanamazlar.Fakat static veri elemanlarını doğrudan kullanabilirler.**

3- **Sınıfın static olmayan bir fonksiyonu, static olmayan başka bir fonksiyonu ve static olan bir fonksiyonu doğrudan çağırabilir.**

4- **Sınıfın static fonksiyonları, sınıfın static olmayan fonksiyonlarını doğrudan çağıramaz. Fakat static fonksiyonlarını doğrudan çağırabilir.**

Yukarıdaki 4 madde de şöyle özetlenebilir:

1- **Sınıfın static olmayan fonksiyonları, sınıfın static olmayan elemanlarını ve static olan elemanlarını doğrudan çağırabilir.**

2- **Sınıfın static fonksiyonları, yalnızca sınıfın static elemanlarını kullanabilir.**

RASGELE SAYI ÜRETİMİ

.Net'te rasgele sayı üretmek için `System` isim alanı içerisindeki `Random` sınıfı kullanılır. `Random` sınıfının static olmayan `int` parametreli `Next` isimli fonksiyonu her çağrıldığında `0` ile `n-1` arasında düzgün dağılmış rasgele bir sayı belirliyoruz. Örneğin `1` ile `100` arasında rasgele `10` sayının üretimi şöyledir:

```
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));

          }
     }
}
```

Sınıf Çalışması: Bir grupta Ali, Veli, Selami, Ayşe ve Fatma isminde beş kişi vardır. Rasgele bir kişinin ismini ekrana yazdıran program yazınız.
Çözüm:

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

               switch(r.Next(5))
               {
                    Case 0:
                    System.Console.WriteLine("Ali");
                    Break;

                    Case 1:
                    System.Console.WriteLine("Veli");
                    Break;

                    Case 2:
                    System.Console.WriteLine("Selami");
                    Break;

                    Case 3:
                    System.Console.WriteLine("Ayşe");
                    Break;

                    Case 4:
                    System.Console.WriteLine("Fatma");
                    Break;

               }
          }
     }
}
```

Sınıf Çalışması: 0 ile 1 arasında çok sayıda rastsal sayı üretiniz. 0 yazı, 1 tura anlamına gelsin. Deney sayısı çok arttırıldığında yazı ve tura gelme olasılığının 0,5 yakınsadığını gösteriniz.
Çözüm:

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

               int head = 0, tail = 0;

               for(int i = 0; i < 1000000; ++i)
               {
                    if (r.Next(2) == 0)
                         ++tail;
                    else
                         ++head;
               }
               System.Console.WriteLine((double)head / 1000000);
               System.Console.WriteLine((double)tail / 1000000);
          }
     }
}
```

System.String SINIFI

System isim alanındaki String sınıfı yazılarla ilgili faydalı işlemler yapmak için kullanılmaktadır.

`System.String` sınıfı çok fazla kullanıldığı için bu sınıf `string` (küçük harfle) anahtar sözcüğü ile de temsil edilmiştir. Yani `System.String` demekle `string` demek tamamen aynı anlamdadır.

String sınıfının amacı yazı tutmaktır. Yazı heap'te tahsis edilmiş olan nesnenin içerisinde bulunmaktadır. String nesnesi klasik olarak new operatörü ile yaratılmaz.

C#'ta ne zaman çift tırnak içerisinde bir yazı yazılsa derleyici bu yazıyı gördüğünde heap'te kendisi bir string nesnesi tahsis eder ve çift tırnak içerisindeki yazıyı bu nesnenin içerisine yerleştirir. Çift tırnak ifadesi yerine heap'te tahsis ettiği nesnenin başlangıç adresi yani referansı yerleştirilir. Yani C#'ta çift tırnak ifadeleri "*bir string nesnesi tahsis et, içerisine bu yazıyı yerleştir ve nesnenin referansını elde et*" anlamına gelmektedir.

Heap'te yaratılan bu string nesnesi yine çöp toplayıcı sistem tarafından nesne seçilebilir duruma geldiğinde geldiğin de otomatik olarak silinecektir.

System isim alanındaki Console sınıfının Write ve WriteLine fonksiyonlarına bir string referansı verilirse bu fonksiyonlar string nesnesi içerisindeki yazıyı yazarlar. Örneğin:

```
public static void Main()
{
     string s;
     s = "Ankara";
     System.Console.WriteLine(s);
}
```

24.07.2007
Salı

İki string referansı + operatörü ile toplama işlemine sokulabilir. Bu durumda yeni bir string nesnesi yaratılır. Yeni string nesnesinin içerisindeki yazı, iki referansta belirtilen nesnelerin içerisindeki yazının birleşiminden oluşturulur. Örneğin:

```
string a = "Ankara";
string b = "İzmir";
string c = a + b;

System.Console.WriteLine(c)  à Ekran çıktısı "AnkaraİzmirAdana"
```

Örneğin:

```
…
public static void Main()
{
     string a, b, c, d;
     a = "Ankara";
     b = "İzmir";
     c = "Adana";
     d = a + b + c;
     System.Console.WriteLine(d)  à Ekran çıktısı "AnkaraİzmirAdana"
}
…
```

Burada önce a + b işlemi yapılacak, bu işlem sonunda yeni bir string nesnesi yaratılacak, sonra yaratılan nesne c string nesnesi ile toplanacaktır. Bu işlem de yeni bir nesnenin yaratılmasına yol açacaktır. a + b işlemi sonucunda yaratılmış olan string nesnesi işlemden sonra hemen çöp toplayıcı tarafından seçilebilir bir duruma gelecektir.

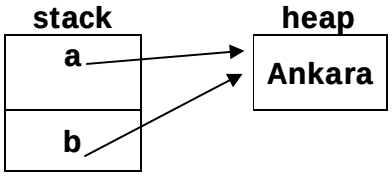
Anahtar Notlar:

.Net dünyasında assembly terimi .exe ve .dll uzantılı dosyalar için kullanılmaktadır. Buradaki assembly teriminin sembolik makine dili anlamına gelen assembly terimi ile bir ilgisi yoktur.

C# standartlarına göre aynı assembly içerisindeki tamamen özdeş karakterlerden oluşmuş stringler için yeniden yerler tahsis edilmez. Bunlar için

toplamda tek bir nesne tahsis edilir. Bu özdeş stringlerin hepsi aynı nesneye ilişkin referans belirtir. Örneğin:

```
a = "Ankara";
b = "Ankara";
```



Boş string kullanmak geçerlidir. Bu durumda derleyici yine bir string nesnesi yaratır. Fakat stringin tuttuğu bir yazı yoktur. Örneğin:

```
string a = "";      à geçerli
```

İki string referansı == ve != operatörleri ile karşılaştırılabilir. Bu durumda referansların aynı nesneyi gösterip göstermediği değil referansın gösterdikleri nesnenin içindeki yazıların aynı olup olmadığı sorgulanmaktadır.

Bir string nesnesinin tuttuğu yazının herhangi bir karakterine [] operatörü ile erişebiliriz. S string türünden bir referans olmak üzere `s[ifade]` ifadesi bu referansın gösterdiği yerdeki string nesnesinin ilgili indeksteki karakterini belirtir. İfade `char` türdendir.

C#'ta bir string nesnesi yaratıldıktan sonra bir daha nesnen tuttuğu yazının herhangi bir karakteri değiştirilemez. Örneğin:

```
string a = "Ankara";
a[2] = 'x';      à error
```

String sınıfının `Length` isimli int türden read only, property elemanı, stringin tuttuğu yazının karakter uzunluğunu vermektedir.

System isim alanındaki Console sınıfının `ReadLine` isimli static fonksiyonu, klavyeden bir yazı girilip enter tuşuna basılana kadar bekler. Girilen yazıyı yarattığı bir string nesnesinin içerisine yerleştirir. O string nesnesinin referansı ile geri döner. Fonksiyonun parametrik yapısı şöyledir(Kaynak MSDN):

```
public static string Readline()
```

Örneğin:

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

     System.Console.Write("Bir yazı giriniz: ");
     s = System.Console.ReadLine();
     System.Console.WriteLine(s);
}
```

String sınıfının static olmayan ToLower fonksiyonu, yazıyı küçük harfe dönüştürmek için kullanılır. Fonksiyon yeni bir string nesnesi oluşturur. Bu string nesnesi içerisine yazının küçük harfe dönüştürülmüş biçimini yerleştirir. Örneğin:

```
string a = "ANKARA";
string b;

b = a.ToLower();
System.Console.WriteLine(b);
```

Sınıfın büyük harfe dönüştürme yapan ToUpper fonksiyonu da vardır.(Dönüştürme işlemi default olarak Windows işletim sistemindeki bölgesel ayarlarda belirtilen ülkenin diline göre yapılmaktadır)

String sınıfının Substring isimli static olmayan fonksiyonu nesnenin belirttiği yazının belli bir indeksinden belli bir uzunluk kadarını alıp bundan yeni bir string nesnesi yaratarak bu yeni nesnenin referansını geri dönmektedir. Parametrik yapısı şöyledir:

```
public string Substring (int StartIndex , int Length)
```

Örneğin:

```
…
public static void Main()
{
     string a = "AĞRI DAĞI ÇOK YÜKSEK";
     string b;

     b = a.Substring(10 , 3)  à (kaçıncı karakterden , kaç tane karakter)
     System.Console.WriteLine(b);    à Çıktı : 'ÇOK'
}
…
```

String sınıfının static olmayan Insert isimli fonksiyonu nesnenin belirttiği yazının belli bir indeksine yeni bir yazı ekler. Tabi string nesnesinin karakterleri üzerinde değişiklik yapılamayacağına göre fonksiyon yeni bir string nesnesi yaratıp onunla geri dönmektedir. Fonksiyonun parametrik yapısı şöyledir:

```
public string Insert (int StartIndex , string value)
```

Örneğin:

```
…
public static void Main()
{
     string a = "Bugün hava sıcak";
     string b;

     b = a.Insert(10 ,"çok");  à (kaçıncı karakterden , kaç tane karakter)
```

```
    System.Console.WriteLine(b);   à Çıktı : 'Bugün hava çok sıcak'
}
…
```

Eğer indeks değeri yazı uzunluğundan büyükse exception oluşur.

String sınıfının Trim isimli fonksiyonu yazının başındaki ve sonundaki boşluk karakterlerinin atılmasında kullanılır.

Daha önce de belirtildiği gibi switch parantezi içerisinde ve case ifadelerinde string kullanılabilir.

String sınıfının static olmayan Remove isimli fonksiyonu, nesnenin belirttiği yazının belirli bir indeksinden itibaren belirli bir miktar karakteri silerek, bu karakterler silinmiş yeni bir string nesnesinin referansı ile geri döner. Parametrik yapısı şöyledir:

```
public string Remove (int StartIndex , int Count)
```

Örneğin:

```
string s= "EskişehirHisar";
string 0 result;

result = s.Remove(4 , 5);
System.Console.WriteLine(result)  à Çıktı : "EskiHisar"
```

Stringler tek satır üzerinde yazılmak zorundadır. Örneğin:

```
string s;

s = "Bugün hava
     çok sıcak";      à geçersiz
```

Eğer yazı çok uzunsa + operatörü kullanılarak böyle bir bölme yapılabilir. Örneğin:

```
string s;

s = "Bugün hava" +
    "çok sıcak";      à geçerli
```

Çift tırnak içerisinde ters bölü karakterleri kullanılabilir. Bunlar tekbir karakter belirtmektedir. Örneğin:

```
string = "Ali\tVeli";   à \t karakteri "" içerisinde tab anlamında
```

Eğer bir stringin başında bitişik bir biçimde @ karakteri varsa böyle stringlere dayanıklı (verbatim) stringler denir. Örneğin:

```
@"Ankara"; à Dayanıklı string
```

Dayanıklı stringler birden fazla satıra bölünebilir. Dayanıklı stringler içerisindeki ters bölü karakterleri özel anlam ifade etmez. Normal ters bölü karakteri olarak ele alınır. Örneğin:

```
string path = "c:\\windows\\temp"; ifadesi ile

string path : @"c:\windows\temp"; ifadesi aynı anlamdadır.
```

String sınıfının static Compare isimli fonksiyonu iki yazıyı karşılaştırmada kullanılmaktadır. Parametrik yapısı şöyledir:

```
public static int Compare (string StrA , string StrB)
```

Fonksiyon 1. yazı 2. yazıdan büyükse pozitif herhangi bir değere, 2. yazı 1. yazıdan büyükse negatif bir değere, yazılar eşitse sıfıra geri döner. Örneğin:

```
…
string a , b;
System.Console.Write("İki yazı giriniz: ");

a = System.Console.ReadLine();
b = System.Console.ReadLine();

int result;
result = string.Compare(a , b);

if(result > 0)
     System.Console.Write("a > b");
else if(result < 0)
     System.Console.Write("a < b");
else
     System.Console.Write("a = b");
…
```

İSİM ALANLARI

26.07.2007
Perşembe

İsim alanı bildirimi şöyle yapılır:

```
namespace < isim alanı ismi >
{
     //…
}
```

İsim alanları iç içe yada ayrık bir biçimde bildirilebilir. Örneğin:

```
namespace A
{
     namespace B
     {
```

```
            //…
      }
}
namespace C
{
      //…
}
```

Hiçbir isim alanı içerisinde olmayan bölgeye global isim alanı denmektedir. Bir sınıf, herhangi bir isim alanı içerisinde yada global isim alanında tanımlanabilir.

İç içe isim alanları araya nokta konularak tek hamlede bildirilebilir. Örneğin:

```
namespace A.B
{
      //…
}
```

ile

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

Yukarıdaki iki ifade tamamen aynı ifadelerdir. İsim alanları parçalı bir biçimde bildirilebilir. Başka bir deyişle bir isim alanının ikinci kez bildirilmesi öncekine ekleme yapıldığı anlamına gelmektedir. Örneğin:

```
namespace A
{
      class x
      {
        //…
      }
}
```

```
namespace A
{
      class y
      {
        //…
      }
}
```

Yukarıda A isim alanı içerisinde hem x hem de y vardır.

Farklı isim alanlarının içerisinde aynı isimli türler bulunabilir.Fakat aynı isim alanı içerisinde aynı isimli birden fazla tür bulunamaz.

.Net'in tüm sınıfları `System` isim alanı içerisinde yada `System` isim alanı içerisindeki isim alanlarının içerisinde bulunmaktadır. Programcının bu isim alanına başka bir eleman yerleştirmesi tavsiye edilmez.

DİNAMİK KÜTÜPHANELERİN KULLANIMI

Uzantısı .dll biçiminde olan dosyalara dinamik kütüphane dosyaları denmektedir. .Net ortamında bağımsız olarak yüklenebilen .exe ve .dll uzantılı dosyalara assembly denmektedir. Teknik anlamda .exe ve .dll dosyalara arasında bir format farklılığı yoktur.

Anahtar Notlar:

.exe ve .dll dosyaları doğal kod yad arakod içerebilir. Başka bir deyişle bu dosyaların yalnızca ismine bakarak bunların .Net ortamı için yazılıp yazılmadığını anlayamayız. Eğer bir .exe yada .dll .Net ortamında kullanılan bir assembly dosyası ise PE formatı bakımından ek bazı sectionlara sahiptir. Örneğin; Bir .Net .exe yada .dll dosyasında "assembly manifest" ve "meta data" sectionları vardır. Bir exe dosyası çalıştırılmak istediğinde işletim sistemi PE formatını inceleyerek bunun bir .Net programı olup olmadığına bakar. Eğer bunun bir .Net ortamında oluşturulmuş exe ise arakod içermektedir ve CLR tarafından çalıştırılır. Değilse doğal yolla çalıştırılır.

dll dosyaları kütüphane amaçlı kullanılmaktadır. Yani bunların içerisinde sınıflar vardır. Programcı bu sınıfları kendi yazmış gibi kullanabilir. Örneğin; br x firması bir grup yaralı sınıf yazmış olsun bizde bu sınıfları kullanmak istiyoruz. Firma bu sınıflar bir .dll içerisine yerleştirir ve bize dll olarak verir. Şüphesiz dll içerisindeki bu sınıflar derlenmiş bir biçimde bulunmaktadır.

.Net'in kendi sınıfları da fiziksel olarak dll dosyaları içerisindedir. Şüphesiz bir dll'nin içerisinde tek bir isim alanına ilişkin sınıflar bulunmak zorunda değildir. Bir dll'nin içerisinde çok farklı isim alanları bulunabilir. Yani isim alanı kavramı ile dll kavramı arasında bir bağlantı yoktur.

DLL DOSYALARININ OLUŞTURULMASI

Csc komut satırı derleyicisinde dll oluşturmak için `/target:library` seçeneği kullanılmalıdır. (`/target` yerine `/t` de kullanılabilir.) Örneğin:

```
csc /target:library test.cs
```

Aslında /target seçeneğinde 4 belirleme yapılabilmektedir:

1- `/target:exe` Bu seçenek bir konsol exe oluşturmaktadır.

Anahtar Notlar:

Bir exe dosya y bir konsol yada bir GUI uygulamasıdır. Konsol uygulaması, işletim sistemi programı çalıştırırken eğer komut satırında değilsek konsol ekranı denilen siyah ekranı açar. Fakat GUI uygulamalarında bu tür siyah ekran bulunmamaktadır.

`/target:exe` seçeneği default durumdadır. Yani /target seçeneği hiç belirtilmezse konsol exe'si yaratılır.

2- `/target:winexe` Bu seçenek GUI exe dosyası oluşturmaktadır. Yani böyle bir exe çalıştırıldığında konsol ekranı gözükmez.

3- `/target:library` dll doyası oluşturmak için kullanılır.

4- /target:module Modül dosyası oluşturmak için kullanılır.

IDE kullanarak dll yapmak için boş bir proje yaratılır. Sonra proje properties menüsüne gelinir. Output Types bölümünden Class Library olarak seçilir. Bu durumda exe yerine dll oluşturulur.

DLL DOSYALARININ KULLANILMASI

Bir dll dosyasını kullanmak için onu referans etmek gerekir. Referans etme işlemi komut satırında /reference:dll dosya ismi seçeneği ile yapılabilir. /reference yerine /r de kullanılabilir. Örneğin:

```
csc /r:test.dll Sample.cs
```

Burada Sample.cs derlenerek Sample.exe yapılmak istenmiştir. Program içerisinde test.dll de bulunan bir takım sınıflar kullanıldığı için test.dll'ye de referans edilmiştir. Şüphesiz birden fazla dll ye referans edilebilir. Örneğin:

```
csc /r:a.dll /r:b.dll /t:winexe Sample.cs
```

Burada programcı Sample.cs içerisinde hem a.dll hem de b.dll içerisindeki sınıfları kullanmıştır. Bu nedenle bu dll dosyalarını referans etmiştir. Buradan ürün olarak bir GUI exe elde edilecektir.

Derleme işlemi sırasında referans edilen dll'lerin ayrıca programın çalışma zamanında da exe dosyası ile aynı klasörde bulundurulması gerekir. Bir exe dosya çalıştırılırken o exe dosyanın referans ettiği dll'ler de çalışma sırasında bütünsel bir biçimde yüklenmektedir. Yani bizim yazdığımız programı başka bir makineye kuracaksak exe ile birlikte o exe'nin referans ettiği dll'leri de o makineya götürmemiz gerekir.

IDE den bir dll'ye referans edebilmek için Solution Explorer'de References üzerine gelinir. Farenin sağ tuşuna basılır ve Add Reference seçilir. Add Reference dialog penceresinde Browse tabına geçilir ve dll'nin yeri bulunarak ekleme yapılır. BU biçimde referans etme işlemi yapıldığında IDE referans edilen dll'yi zaten exe'yi oluşturacağı klasöre kopyalamaktadır.

Sınıf Çalışması: IDE kullanılarak bir dll dosyası oluşturunuz. dll'ye bir tane sınıf ve bir static fonksiyon yerleştiriniz. Sonra bir exe projesi oluşturarak bu dll'yi referans edip söz konusu sınıf ve fonksiyonu kullanınız.
Çözüm:

dll projesi

```
namespace CSD
{
     public class Sample
     {
          public static void Func()
          {
               System.Console.WriteLine("Ben dll dosasıyım…");
          }
```

```
    }
}
```

exe projesi

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample.Func();   à referans yapılıyor.
        }
    }
}
```

Bir sınıf bildiriminin önüne public yada internal belirleyicilerinde bir getirilebilir. Eğer hiçbir belirleyici kullanılmamışsa internal belirleyicisi kullanılmış gibi varsayılır. Public belirleyicisi sınıfın başka bir assembly'den (örneğin bir exe'den) kullanılabileceğini belirtmektedir. Halbuki internal sınıflar yalnızca assembly içinde kullanılabilir. Özetle bir dll içerisindeki sınıfın dışarıda kullanılabilmesi için sınıfın public belirleyicisi ile belirtilmesi gerekir.

.NET'İN SINIFLARINA İLİŞKİN DLL DOSYALARI

.Net'in kendi sınıfları da çeşitli dll dosyaları içerisine yerleştirilmiştir. Yani programcının bu sınıfları kullanabilmesi için o sınıflar hangi dll'ler içerisinde ise o dll'ye referans etmesi gerekir. .Net'in bu dll'leri frameworkun bir parçası kabul edilir ve framework kurulumu yapılırken kurulum sırasında hedef makineye çekilmektedir. Bu dll'ler "Global Assembly Cache" içerisindedir. Bu nedenle exe ile aynı klasörde olmak zorunda değildir.

Csc derleyicisi hiç belirtilmese bile `mscorlip.dll` isimli dll'ye zaten otomatik referans etmektedir. .Net'in çeşitli isim alanlarındaki en çok kullanılan sınıfları `mscorlip.dll` içerisindedir. Bu durumda bir .Net sınıfı eğer `mscorlip.dll` içerisinde ise onu kullanmak için bir referans işlemine gerek yoktur. Fakat `mscorlip.dll` içerisinde olmayan sınıfların hangi dll içerisinde bulunduğu belirlenmeli ve o dll'ye referans edilmelidir.

Bir .Net sınıfı kullanılacaksa 2 şeyin bilinmesi gerekir:

1. **Sınıf hangi dll dosyası içerisindedir?**
2. **Sınıf hangi isim alanı içerisindedir?**

İSİM ARAMA İŞLEMİ

Bir değişkenin kullanıldığını gören derleyici o değişkene ilişkin bildirimi belirlemeye çalışır. Bu belirleme sırasında değişkenin bildirimini çeşitli faaliyet alanlarında arar. Eğer değişken bildirimi ile karşılaşırsa arama işlemini

durdurur, eğer değişkenin bildirimi ile hiç karşılaşmazsa işlem error ile sonlanır. Bu işleme isim araması(name look up) denmektedir. İsim araması her türlü isme uygulanmaktadır. Örneğin yerel değişken isimleri, sınıf isimleri, isim alanı isimleri bu arama işlemine sokulmaktadır. İsim arama işlemi 2 ye ayrılır:

1- **Niteliksiz İsim Arama: "." operatörü kullanılmadan doğrudan yazılmış isimlerin aranmasına denilmektedir.**

2- **Nitelikli İsim Arama: "." operatörünün sağında isimlerin aranmasına denilmektedir. Örneğin; X.Y işleminde X niteliksiz olarak Y'de nitelikli olarak aranmaktadır.**

Niteliksiz İsim Araması: Nokta operatörü kullanılmadan doğrudan yazılan isimler için niteliksiz arama kuralları uygulanır. Niteliksiz arama işlemi şu adımlarla gerçekleştirilir:

1- **Derleyici ismin kullanıldığı yerden yukarıya doğru alan içerisinde fonksiyonun yerel bloklarında ismin bildirimini arar.**

2- **İsim sınıf bildiriminin her yerinde aranır.**

3- **İsim kullanıldığı sınıfın içinde, bulunduğu isim alanının her yerinde aranır.**

Anahtar Notlar

`Sample s;` gibi bir bildirim işleminde s isimi bildirilmektedir. Bu nedenle bu isim aranmaz. Fakat `Sample` ismi niteliksiz arama işlemine sokulacaktır.

4- **İsim kullanıldığı sınıfın içinde, bulunduğu isim alanlarında içten dışa doğru bu isim alanlarının her yerinde aranır. Örneğin; farklı isim alanlarında aynı isimli sınıflar bulunabilir. Eğer kapsayan ve kapsanan isim alanlarında aynı isimli sınıflar varsa arama içten dışa doğru yapıldığından içteki isim alanındaki sınıf bulunur.**

5- **İsim nihayet global isim alanın her yerinde aranır.**

Nitelikli İsim Araması: Nokta operatörünün sağındaki isimlerin aranmasında nitelikli isim arama kuralları uygulanır. Nokta operatörünün sol tarafındaki operand bir isim alanı ismi, tür ismi ya da bir sınıf türünden değişken olabilir.

1- **Nokta operatörünün solundaki isim bir isim alanı ismi ise sağındaki isim yalnızca o isim alanının her yerinde aranır. Kapsayan isim alanlarında aranmaz.**

2- **Nokta operatörünün solunda bir sınıf ya da yapı ismi varsa sağındaki isim o sınıf ya da yapı bildiriminin her yerinde aranır. Fakat Kapsayan isim alanlarında aranmaz.**

3- **Nokta operatörünün solunda bir sınıf türünden referans ya da bir yapı değişkeni varsa sağındaki isim o referansın ya da değişkenin ilişkin olduğu sınıf ya da yapı bildiriminin her yerinde aranır. Kapsayan isim alanlarına bakılmaz.**

A.B.C gibi birden fazla "." operatörü ile bir isim belirtilmiş olsun. Bu durumda A niteliksiz olarak aranır, B ve C nitelikli olarak aranır. (Yani kabaca önce A'nın bulunması gerekir. B A içerisinde C'de A.B içerisinde aranır)

İsim araması karşılaşılan her isim için yapılmaktadır.
Örneğin:

```
Sample s = new Sample();
```

Burada s bildirilmiştir aranmayacaktır. Fakat iki `Sample` ismi niteliksiz olarak aranacaktır.

```
namespace A
{
     namespace B
     {
          class App
          {
               public static void Main()
               {
                    Sample s = new Sample();
                    s.Func();
               }
          }
     }
}
```

```
namespace A
{
     namespace B
     {
          class Sample
          {
               public void Func()
               {
                    //…
               }
          }
     }
}
```

Referans edilen dll'ler içerisindeki bildirilmiş isimler sanki programcı kendisi yazmış gibi arama işlemine sokulmaktadır. Örneğin; `System.Console.WriteLine` gibi bir isim alanı kullanılmış olsun. Burada `System` niteliksiz olarak aranacaktır. Fakat bu arama işlemi yalnızca kaynak kod la değil referans edilmiş olan dll'lerle de yapılacaktır. System isim alanı en azından mscorlib.dll içerisinde bulunacaktır.

Framework 1.1aşağıdaki gibi bir durumda yapılacak bir şey yoktu:

```
namespace CSD
{
     namespace System
     {
          class App
          {
               public static void Main()
               {
                    System.Console.WriteLine("Test");   à error
               }
          }
     }
}
```

Buradaki `System` isim alanı .Net'in global isim alanı altındaki `System` isim alanı değildir.

Anahtar Notlar

Anımsanacağı gibi bir isim alanının iki kez bildirimi yapıldığında bu durum ekleme anlamına gelmektedir. Farklı isim alanlarının altında bulunan aynı isimli isim alanları gerçekte farklı isim alanlarıdır.Örneğin:

```
namespace A
{
     namespace x
     {
          //…
     }
}

namespace B
{
     namespace x
     {
          //…
     }
}
```

Burada x isim alanları birleştirilmez. Çünkü aynı isim alanı değildir.

Örneğimizde `System` isim alanı niteliksiz olarak arandığın da CSD'nin altındaki isim alanı olarak bulunacaktır. Bundan sonra artık `Console` ismi CSD'nin `System` isim alanında bulunacağından bulunamayacaktır.

Framework 2.0'da bu sorun yeni eklenen "`global::`" operatörü ile çözülmüştür. "`global::`" operatörünün sağındaki isimler yalnızca global isim alanında aranmaktadır. O halde sorun şöyle çözülebilir:

```
namespace CSD
{
     namespace System
     {
          class App
          {
               public static void Main()
               {
                    global::System.Console.WriteLine("Test");
               }
          }
     }
}
```

using DİREKTİFİ

using direktifi nokta operatörü ile nitelendirme işlemini kolaylaştırmak için düşünülmüştür. using direktifinin genel biçimi şöyledir:

```
using < isim alanı ismi >
```

Örneğin:

```
using A;

using X.Y;
```

using direktifi isim alanlarının başına yerleştirilmek zorundadır. Başka bir deyişle using direktiflerinden önce başka bildirimler bulunamaz. Örneğin:

```
namespace CSD
{
     using A;      à doğru kullanım
     using B;      à doğru kullanım

     class x;
     {
          //…
     }
     using C;      à yanlış kullanım
}
```

Using direktifi global isim alanına da yerleştirilebilir. Global isim alanının başı kaynak dosyanın tepesidir.

Using direktifinde bir direktifin yerleştirildiği isim alanı, bir de direktifte belirtilen isim alanı vardır. Örneğin

```
namespace CSD
{
```

```
     using System;
     //…
}
```

Burada direktifin yerleştirildiği isim alanı CSD, direktifte belirtilen isim alanı System isim alanıdır. Direktifte belirtilen isim alanı nokta operatörüyle kombine edilmiş olabilir. Bu durumda direktifte belirtilen isim alanının bulunması için yine daha önce görülen isim arama kuralları uygulanır. Örneğin:

```
namespace CSD
{
     using A.B.C;
     //…
}
```

Burada belirtilen isim alanı A.B içerisindeki C isim alanıdır.

Niteliksiz isim arama sırasında isim `using` direktifinin yerleştirildiği isim alanında bulunamasaydı, isim `using` direktifi ile belirtilen isim alanlarında da aranır. Örneğin; `Console` ismi niteliksiz aranırken `CSD` isim alanında bulunamayacaktır. Bu durumda direktifte belirtilen `System` isim alanına da bakılacaktır ve orada bulunacaktır. Örneğin.

```
namespace CSD
{
     using System;

     //…

     Console.WriteLine("Test");
}
```

İsim using direktifinde belirtilen isim alanında arandıktan sonra başka bir isim alanına bakılmaz. Yani kapsayan isim alanlarına bakılmaz. Örneğin:

```
namespace CSD
{
     using B;
     //…

     X a;      à X bulunamaz
}

     namespace A
     {
          class X
          {
               //…
          }
     namespace B;
     {
```

```
     //…
     }
}
```

X ismi A.B'de aranacaktır. Fakat kapsayan isim alanı olan A'da aranmayacaktır.

Using direktifi geçişli değildir. Yani isim using direktifi ile belirtilen isim alanında bulunamadığında o isim alanına yerleştirilmiş using direktifleri dikkate alınmaz. Örneğin:

```
namespace CSD
{
     using B;
     X a;      à X bulunamayacaktır.
}
namespace A
{
     class X
     {
          //…
     }
}
namespace B
{
     using A;
     //…
}
```

Using direktifi, isim direktifinin yerleştirildiği isim alanında bulunamazsa etkili olmaktadır. İsim bulunursa using direktifi de dikkate alınmaz. Örneğin:

```
namespace CSD
{
     using A;
     //…
     X a;
                    ──────►  X CSD isim alanında bulunmuştur.
     class X;
     {
          //…
     }
}
namespace A
{
     class X
     {
          //…
     }
}
```

İsim using direktiflerinin yerleştirildiği isim alanında bulunamamış olsun. Fakat ismin birden fazla using direktifinde belirtilen isim alanında belirtilen isim alanında bulunduğunu varsayalım. Bu durum error oluşturur. Örneğin:

```
namespace CSD
{
  using A;
  using B;

  X a;  à error hem A
          hem de B'de
          bulunuyor
}
```

```
namespace A;
{
     class X
     {
          //…
     }
}
```

```
namespace B;
{
     class X
     {
          //…
     }
}
```

Yukarıda biz hem A firmasının hem de B firmasının yazmış olduğu sınıfları kullanmak istiyoruz. A firması sınıflarını A isim alanında, B firması da B isim alanında bildirmiş olsun. Burada gereksiz niteliklendirmeyi engellemek için hem A hem de B isim alanlarına using direktifi uygulanmıştır. Fakat görüldüğü gibi tesadüfen X ismi her iki isim alanında da bulunmaktadır. İşte bu tür durumlarda artık X isminin A.X ya da B.X biçiminde nitelikli olarak kullanılması gerekir.

Görüldüğü gibi isim alanları içerisindeki isimler çakışmadıktan sonra using direktifi kolaylık sağlamaktadır. Fakat çakışma söz konusu ise nitelendirme yapılmak zorundadır.

Using direktifi nitelikli aramalarda etkili olmaz. Örneğin:

```
namespace CSD
{
     //…
     A.X c;   ──→  X nitelikli arandığı için A içerisindeki
                   using direktifi dikkate alınmaz.
     //…
}

namespace A
{
     using B;
     //…
}

namespace B
{
     class X
     {
          //…
     }
}
```

Bir isim using direktifinin yerleştirildiği isim alanında bulunamamış olsun. Eğer isim using direktifi ile belirtilen isim alanında bir isim alanı ismi olarak bulunursa bu bulma işlemi dikkate alınmamaktadır. Başka bir deyişle isim

using direktifinde belirtilen isim alanında aranırken o isim alanındaki isim alanı isimleri dikkate alınmamaktadır. Örneğin:

```
namespace CSD
{
     using A;
     //…
     B.X k;      → error B ismi A'nın içerisinde bir isim alanı
                   ismi olduğu için bulunamayacak
     //…
}

namespace A
{
     namespace B;
     {
          class X
          {
              //…
          }
     }
}
```

Burada sorunu çözmek için A.B isim alanına using direktifi uygunmalıdır:

```
namespace CSD
{
     using A.B;
     //…
     X k;       à geçerli
     //…
}

namespace A
{
     namespace B;
     {
          class X
          {
               //…
          }
     }
}
```

Burada artık X ismi A.B içerisinde bir sınıf ismi olduğu için bulunacaktır. Bu kuraldan dolayı C# programlarında aşağıdaki gibi bir using merdiveni ile karşılaşılır:

```
using System;
using System.Windows;
using System.Windows.Forms;
```

Bundan sonra mümkün olduğunca using direktifi kullanılarak örneklerde daha az nitelendirme yapılacaktır.

DİZİLER [Arrays]

Birden fazla değişkenin (datanın) oluşturduğu topluluğa veri yapısı denilmektedir. Örneğin; tek başına int türden a veri yapısı değildir. Oysa bir sınıf bir veri yapısıdır. Çünkü sınıf denildiğinde bir grup eleman anlaşılmaktadır. Elemanları aynı türden olan ve bir index yardımıyla erişilebilen veri yapılarına dizi(array) denilmektedir. C# ta T türü için T[] ile temsil edilen bir dizi türü vardır. Örneğin:

```
int a;

int [] b;
```

Burada a int türdendir, b ise int türden bir dizidir. b'nin türü int[] ile belirtilir. C# ta diziler kategori olarak referans türlerine ilişkindir. Örneğin:

```
int [] a;
```

Burada a yalnızca bir referanstır. Dizi nesnesi henüz tahsis edilmemiştir. C# ta diziler bir çeşit sınıftır. Dizi elemanları da bu sınıf nesnesinin içerisinde kabul edilmektedir.

Dizi nesneleri tıpkı sınıf nesnelerinde olduğu gibi new operatörü ile heap ta tahsis edilir. New operatörü ile dizi tahsis etmenin genel biçimi şöyledir:

```
new < tür > < [uzunluk] >
```

Örneğin:

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

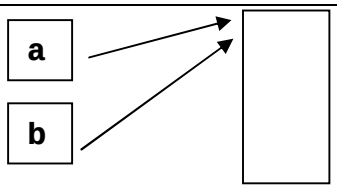
Bu işlemin sonucunda heap ta bir dizi nesnesi yaratılmıştır. Bu dizi nesnesi 10 tane int tutmaktadır.

Dizile new operatörü ile yaratılırken [] içerisine uzunluk ifadesinin tamsayı türlerine ilişkin olması zorunludur. Bu uzunluk ifadesi sabit ifadesi olmak zorunda değildir. Örneğin:

```
//…
int [] a;
int size;

size = int.Parse(Console.ReadLine());

a = new int [size * 2];
//…
```

Aynı türden iki dizi referansı birbirine atanabilir. Bu durumda tıpkı sınıflarda olduğu gibi iki referansta aynı nesneyi gösterir durumdadır. Örneğin:

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

int [] b;

b = a;
```



r, T türünden bir dizi referansı olmak üzere r[ifade] ifadesi referansının gösterdiği yerdeki dizi nesnesinin "ifade" ile belirtilen indexteki elemanını temsil eder. Dizinin ilk elemanı 0. indexteki elemandır. Bu durumda n elemanlı bir dizinin son elemanı n-1. indexteki elemanıdır. Dizinin her elemanı bağımsız bir değişken gibi kullanılabilir.

Dizinin en önemli kullanım nedeni bir döngü içerisinde bir indis yardımıyla dizinin tüm elemanlarının gözden geçirilebilmesidir. Örneğin:

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

//…

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

r T[] türünden bir dizi referansı olsun. Bu durumda r[ifade] T türündendir.

```
using System;

namespace CSD
{
     class App
     {
          public static void Main()
          {
               int [] a = new int[100];
               for (int i = 0; i < 100; ++i)
                    a[i] = i;

               int [] b = a;
               for (int i = 0; i < 100; ++i)
                    Console.WriteLine(b[i]);

          }
     }
}
```

İki dizi referansının birbirine atanabilmesi için türlerinin tamamen aynı olması gerekmektedir. Örneğin; int türünün long türüne doğrudan atabilmesi,

int türden bir dizi referansının long türünden bir dizi referansına atanacağı anlamına gelmez.

Dizi elemanlarına erişirken [] içerisindeki ifade tamsayı türlerine ilişkin olmak zorundadır. Bir dizinin pozitif ya da negatif bakımdan olmayan bir elemanına erişmeye çalışmak exception oluşmasına neden olmaktadır.

Anahtar Notlar

Exception, program hatasız olarak derlendikten sonra çalışma zamanı sırasında oluşan bir hatadır. Yani exception derleme zamanına değil çalışma zamanına ait bir deyimdir.

Her dizinin `Lenght` isimli int türden bir property elemanı vardır. Bu property elemanı dizinin uzunluğunu belirtmektedir. Yani biz `r` bir dizi referansı olmak üzere `r.Lenght` ifadesi ile bu dizinin uzunluğunu bulabiliriz. `Lenght` property elemanı read only bir elemandır. Yani bu elemanı kullanabiliriz. Fakat bir değer atamamayız.

`i < 100` ifadesi ile `i < a.Lenght` aynı türdendir.

DİZİLERİN FONKSİYONLARA PARAMETRE OLARAK AKTARILMASI

Bir diziyi fonksiyona geçirmek oldukça kolaydır. Fonksiyonun parametre değişkeni dizi türünden referans olur. Fonksiyon da aynı türden bir dizi referansı ile çağrılır. Örneğin:

```
using System;

namespace CSD
{
     class App
     {
          public static void Main()
          {
               int [] a = new int[100];
               for (int i = 0; i < a.Lenght; ++i)
                    a[i] = i;

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

          }
     }
}
```

C# ta bir dizi nesnesi `new` operatörü ile yaratıldığında `new` operatörü nesneyi yarattıktan sonra dizinin tüm elemanlarını sıfırlamaktadır. Eğer dizi bir

referans dizisi ise sıfırlama dizinin her elemanına null değerinin atanması anlamına gelmektedir. Böylece dizi yaratıldıktan sonra elemanlarına bir değer atanmamış olsa bile elemanların içerisinde "0" değer olacaktır.

new operatörü ile bir dizi nesnesi yaratılırken dizi elemanlarına aynı zamanda ilk değer verilebilir. []'den sonra hemen küme parantezi açılarak ilk değerler belirtilir. Örneğin:

```
int [] a;

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

Bir diziye new operatörü ile ilk değer verilirken {} içerisindeki dizi uzunluğunun sabit ifadesi olması zorunludur. Aynı zamanda verilen ilk değerlerin tam olarak belirtilen uzunluk kadar olması gerekir. Örneğin:

```
a = new int[size] {1, 2, 3};
             ↓
           error
```

Yukarıda diziye ilk değer verilmiştir. Fakat uzunluk sabit ifadesi şeklinde verilmemiştir. Fakat örneğin:

```
a = new int[1 + 1 + 1] {1, 2, 3};
                ↓
             geçerli
```

Yukarıda ilk değer verme işlemi geçerlidir.

```
a = new int[5] {1, 2, 3};
            ↓
          error
```

Yukarıda verilen ilk değerle dizi uzunluğu uyuşmamaktadır.

Diziye ilk değer verilirken [] parantezlerinin içi boş bırakılabilir. Bu durumda derleyici verilen ilk değerleri sayar ve sanki [] parantez içerisine o değer yazılmış gibi işlem yapar. Örneğin:

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

Bu işlem tamamen aşağıdaki ile eşdeğerdir.

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

C# ta diziye ilk değer verilirken [] içerisine ilk değerler sabit ifadesi ile oluşturulmak zorunda değildir. Örneğin:

```
int x = 1;

int [] a;

a = new int [] {x, x + 1, x + 2, x + 3};  à geçerli
```

Bir dizi referansı [] içerisinde bir değer listesi ile ilk değer verilerek tanımlanabilir. Örneğin:

```
int [] a = {1, 2, 3};    à geçerli
```

Fakat şüphesiz aynı işlem ilk değer verme haricinde geçerli değildir. Örneğin:

```
int [] a;

a = {1, 2, 3};    à error
```

Dizi referansına bu biçimde ilk değer verildiği zaman derleyici verilen ilk değerleri sayar ve new operatörünü kendisi uygulayarak dizi nesnesini bu uzunlukta yaratır. Bu ilk değerleri de dizi elemanlarına yerleştirir. Yani:

```
int [] a = {1, 2, 3}; ile int [] a = new int[3] {1, 2, 3};
```

tamamen eşdeğerdir.

Görüldüğü gibi bu biçimde ilk değer verme tamamen kolaylık sağlama amacı ile düşünülmüştür. Yoksa yine new işlemi uygulanmaktadır.

Fonksiyonun geri dönüş değeri bir dizi türünden olabilir. BU durumda geri dönüş değerinin aynı türden bir dizi referansına atanması gerekir. Örneğin:

```
//…
public static void Main()
{
     int [] r;
     r = Func();
     for (int i = 0; i < r.Lenght; ++i)
     Console.WriteLine(r[i]);
}
public static int[] Func()
{
     int [] a = new int [] {1, 2, 3, 4, 5};
     return a;
}
//…
```

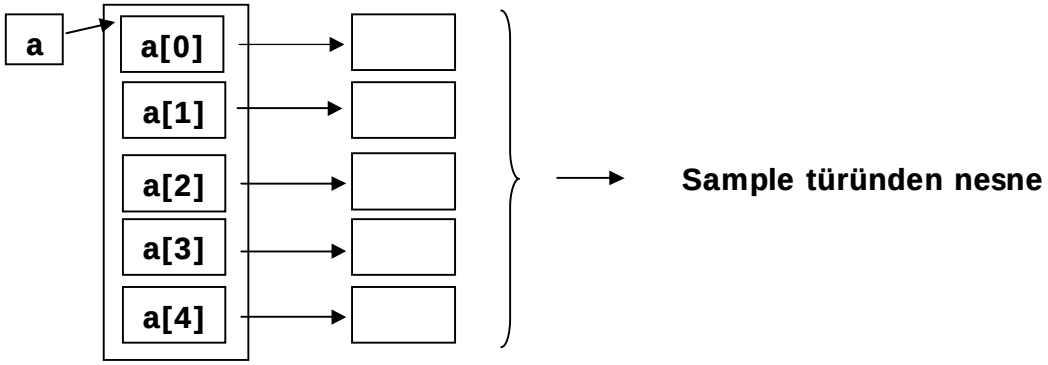
Şimdiye kadar verdiğimiz örneklerde hep temel türlere ilişkin diziler oluşturduk. Halbuki referans türlerine ilişkin diziler de olabilir. Örneğin; x bir sınıf olmak üzere `x [] a = new x [10]` işlemi ile 10 elemanlı x türünden bir referans dizisi oluşturulmuştur. Yani burada dizinin 10 elemanı da bir referanstır ve henüz bunlar için bir tahsisat yapılmamıştır. Sample bir sınıf olmak üzere aşağıda yeni bir dizi yaratmış olalım:

```
Sample [] a;

a = new Sample [10];
```

Burada dizi elemanları olan a[i] ler Sample türündendir. Yani a[i] ler aslında birer referanstır. Bir dizi yaratıldığında dizi temel türlere ilişkinse tüm elemanlarının içerisinde 0, referans türlerine ilişkinse null değeri bulunur. Referans dizisini tahsis ettikten sonra bu referanslar için gerçek nesnelerin de tahsis edilmesi gerekir.

```
Sample [] a;

a = new Sample [10];
for (int i = 0; i < a.Lenght; ++i)
     a[i] = new Sample;
```

Bu durum aşağıdaki gibi gösterilebilir:



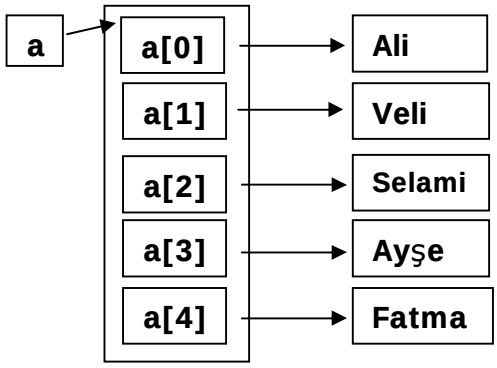
Şüphesiz bir referans dizisi de new operatörü ile yaratılırken ilk değer verilebilir. Bu durumda verilen ilk değerlerin aynı türden sınıf referansları olması gerekir. Örneğin:

```
Sample [] a;

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

string bir sınıf olduğuna göre ve iki tırnak ifadeleri için string nesneleri derleyici tarafından yaratıldığına göre aşağıdaki gibi bir string dizisi oluşturabiliriz:

```
string [] a;
a = new string [] {“Ali”, “Veli”, “Selami”, “Ayşe”, “Fatma”};
```

Durum şekilsel olarak şöyle gösterilebilir:




```
public static void Main()
{
     string [] a;
     a = new string [] {“Ali”, “Veli”, “Selami”, “Ayşe”, “Fatma”};

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

```
public static void Main()
{
     string [] names;
     names = GetNames();

     for (int i = 0; i < names.Lenght; ++i)
          Console.WriteLine(names[i]);

}

public static string [] GetNames()
{
     return new string [] {“Ali”, “Veli”, “Selami”, “Ayşe”, “Fatma”};
}
```

System.IO isim alanındaki Directory isimli sınıfın static GetFiles fonksiyonunun parametrik yapısı şöyledir:

```
public static string [] GetFiles {string path};
```

Directory sınıfı mscorlip.dll içerisinde olduğu için bir dll referans etmeye gerek yoktur. GetFiles fonksiyonu bir klasörün yol ifadesini referans olarak alır. O klasördeki tüm dosya isimlerini bir string dizisine yerleştirir ve o dizinin referansı ile geri döner. Yani bu fonksiyon sayesinde herhangi bir klasördeki dosyalar elde edilebilir. Örneğin:

```
public static void Main()
{
     string [] files;
     files = System.IO.Directory.GetFiles(@”c\windows”);
```

```
     for (int i = 0; i < files.Lenght; ++i)
          Console.WriteLine(files[i]);
}
```

Directory sınıfının başka faydalı fonksiyonları da vardır. Örneğin; sınıfın GetDirectories static fonksiyonu klasör içerisinde bulunan klasörleri verir. Her iki fonksiyonda da parametre olarak var olmayan bir klasör geçirilirse exception oluşur.

FOREACH DÖNGÜSÜ

Foreach döngüsü C'de ve C++'da yoktur. Çok benzeri Java da bulunmaktadır.

Anahtar Notlar

C#'ta INumerible arayüzünü destekleyen sınıf ve yapılara dizilim denilmektedir. Normal diziler de bu arayüzü desteklediklerinden onlar da bir dizilim olarak değerlendirilmektedir. .Net'in collection sınıfları INumerible sınıfını desteklemektedir. Bu nedenle onlarda bir dizilim olarak değerlendirilir.

Foreach döngüleri genel olarak bir dizilimi (dizi ya da collection sınıfı) dolaşmakta kullanılır. Foreach döngüsünün genel biçimi şöyledir:

```
foreach (< tür > < döngü değişkeni > in < dizilim >)
     < deyim >
```

Foreach döngüsünde her yinelemede dizilimin diğer bir elemanı döngü değişkeni içerisine yerleştirilir. Dizilimin tüm elemanları yerleştirildikten sonra döngü sonlanır. Örneğin:

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

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

Burada ilk yinelemede dizinin ilk elemanı olan 1 x'e yerleştirilir. Sonraki yinelemede 2, sonrakiler de 3,4,5 yerleştirilecektir.

Dizi elemanlarının döngü değişkenine atanması işleminin tür dönüştürme operatörü ile yapıldığı varsayılmaktadır. Örneğin:

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

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

Yukarıdaki döngü geçerlidir. Burada dizinin her elemanı int türdendir. int türünden byte türüne doğrudan dönüştürme yoktur. Fakat tür dönüştürme operatörü ile vardır. İşte x'e atamanın bu şekilde yapıldığı varsayılır. Bu işlemin for eşdeğeri aşağıdaki gibidir:

```
//…
int [] a = {1, 2, 3, 4, 5};

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

Örneğin bir string dizisi de benzer biçimde foreach döngüsü ile dolaşılabilir:

```
//…
public static void Main()
{
     string [] names;
     names = new string [] {“Ali”, “Veli”, “Selami”, “Ayşe”,
“Fatma”};

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

Örneğin bir dizideki dosya isimlerini foreach döngüsü ile şöyle yazdırabiliriz:

```
//…
string [] files;
files = System.IO.Directory.GetFiles(@”c:\windows”);

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

Şüphesiz hiç ara değişken kullanmadan aynı işlem pratik bir biçimde şöyle de yapılabilir:

```
//…
foreach (string file in System.IO.Directory.GetFiles(@”c:\windows”))
      System.Console.WriteLine(file);
//…
```

Benzer biçimde aşağıdaki işlem de geçerlidir:

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

Foreach döngüsündeki döngü değişkeni, yalnızca döngü deyimlerinde kullanılabilir. Döngü değişkeninin read only olduğu varsayılmaktadır. Yani bu değişken kullanılabilir fakat bu değişkene yeni bir değer atanamaz.

Java da foreach döngüsü yine for anahtar sözcüğü ile oluşturulmaktadır. Semantik olarak benzerdir fakat syntax olarak in anahtar sözcüğü yerine : atomu bulunmaktadır. Örneğin:

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

for (int x : a)
     System.out.printLn(x);
```

FARKLI PARAMETRİK YAPILI AYNI İSİMLİ FONKSİYONLAR

Anımsanacağı gibi C# ta aynı isim alanı içerisinde, aynı isimli birden fazla sınıf bulunamaz. Fakat farklı isim alanları içerisinde, aynı isimli sınıflar bulunabilir. Yine daha önce farklı sınıflarda aynı isimli ve aynı parametrik yapıya sahip fonksiyonların bulunabileceğini görmüştük.

Bir sınıfta parametrik yapıları farklı olmak koşulu ile aynı isimli birden fazla fonksiyon bulunabilir. Bu özelliğe nesne yönelimli programlama tekniğinde ingilizce "function overloading" denilmektedir. Parametrik yapının farklı olması parametrelerin sayıca ve/veya türce farklı olması demektir. Yani aynı isimli fonksiyonların ya parametre sayıları farklı olmalıdır eğer parametre sayıları aynı ise bunların türleri farklı olmak zorundadır.

```
//…
class Sample
{
     public void Func()
     {
          //…
     }
     public void Func(int a)
     {
          //…
     }
     public void Func(long a)
     {
          //…
     }
}
//…
```

Burada üç Func fonksiyonu da Sample sınıfında bulunabilir. Çünkü bu fonksiyonların parametrik yapıları farklıdır. Parametre değişkenlerinin isimlerinin farklı olmasının hiçbir etkisi yoktur. Önemli olan türlerin farklı olmasıdır.

Fonksiyonların geri dönüş değerinin parametrik yapıyı değiştirmede bir etkisi yoktur. Yani parametrik yapısı aynı fakat geri dönüş değeri farklı aynı isimli fonksiyonlar aynı sınıfta bulunamaz. Örneğin:

```
//…
class Sample
{
     public void Func(int a)
     {
          //…
     }
     public int Func(int a)
     {
          //…
     }
}
//…
```

geçersiz

Benzer biçimde erişim belirleyicilerinin farklı olması ya da static olma durumu parametrik yapıyı farklı hale getirmez. Yani örneğin aynı isimli ve aynı parametrik yapıya sahip biri normal diğeri static olan iki fonksiyon aynı sınıfta bulunamaz.

Parametrelerin sırası farklılaştırmada etkilidir. Örneğin; aşağıdaki iki fonksiyon aynı anda bulunabilir:

```
//…
class Sample
{
     public void Func(int a, long b)
     {
          //…
     }
     public void Func(long a, int b)
     {
          //…
     }
}
//…
```

Aynı isimli bir fonksiyon çağrıldığında acaba derleyici bu fonksiyonların hangisinin çağrılmış olduğunu anlayacaktır? Kabaca ifade edilirse derleyici önce çağrılma ifadesindeki parametrelerin türlerini hesap eder. Bu parametrik türe tam uygun, aynı isimli fonksiyon varsa onun çağrılmış olduğunu kabul eder. Peki ya yoksa?

Aynı isimli bir fonksiyon çağrıldığında, çağrılacak olan fonksiyonun tespit edilmesi işlemine ingilizce "`overload resolution`" denilmektedir. `Overload resolution` işlemi karmaşık bir işlemdir. Programcıların çoğu bu işlemin ayrıntılarını tam olarak bilmez. Programcı, çağırma ifadesindeki türleri tam olarak parametrik yapıdakine uygun hale getirirse, çağırmak istediği fonksiyonu her zaman çağırabilir. Eğer çağırma ifadesindeki parametrelere tam uygun olan bir fonksiyon yoksa işte bu durumda hangi fonksiyonun çağrılacağı `overload resolution` kurallarına göre tespit edilir.

OVERLOAD RESOLUTION İŞLEMİ

Overload resolution işlemi üç aşamada yürütülmektedir. Birinci aşamada aday fonksiyon belirlenir. İkinci aşamada aday fonksiyonlar içerisinden uygun fonksiyonlar seçilir. Üçüncü aşamada ise uygun fonksiyonlar yarışa sokularak en uygun fonksiyon seçilir.

Çağrılma ifadesindeki sınıfta bulunan tüm aynı isimli fonksiyonlar aday fonksiyonlardır. Örneğin:

```
  //…
class Sample
{
     public void Func()                        -1-
     {
          //…
     }
     public void Func(int a)                   -2-
     {
          //…
     }
     public void Func(int a, long b)           -3-
     {
          //…
     }
     public void Func(long a, int b)           -4-
     {
          //…
     }
     public void Func(int a, int b)            -5-
     {
          //…
     }
     public void Func(double a, double b)      -6-
     {
          //…
     }
}
//…
```

Fonksiyonun şöyle çağrıldığını varsayalım:

```
     Sample.Func('a', 'b');
```

Burada aynı isimli tüm fonksiyonlar (1,2,3,4,5,6 nolu fonksiyonlar) aday fonksiyonlardır.

Uygun fonksiyonlar çağrılma ifadesindeki parametrelerle aynı sayıda parametre değişkenine sahip olan ve çağrılma ifadesindeki parametrelerin her birinden, parametre değişkenlerine doğrudan dönüştürmenin söz konusu olduğu fonksiyonlardır. Örneğimizde 1 ve 2 nolu fonksiyonlar uygun fonksiyonlar değildir. Çünkü parametre sayıları uyuşmamaktadır. 3 nolu fonksiyon uygundur. Çünkü char à int, char à long doğrudan dönüştürmesi

başka bir deyişle doğrudan atama işlemi söz konusudur. Benzer biçimde 4,5,6 nolu fonksiyonlar aday fonksiyonlardır. Şimdi fonksiyonun aşağıdaki gibi çağrıldığını varsayalım:

```
Sample.Func(10,10,2);
```

Bu durumda tüm fonksiyonlar adaydır fakat yalnızca 6 nolu fonksiyon uygun fonksiyondur.

En uygun fonksiyon öyle bir fonksiyondur ki çağrılma ifadesindeki parametrelerle yarışa sokulduğunda her parametresi daha iyi dönüşüm sağlayan ya da daha kötü bir dönüşüm sağlamayan fonksiyondur.

İki doğrudan dönüştürme arasında kalite farkı vardır. Kalite farklılığı şu maddelerle açıklanabilir:

1. **Eğer özdeş dönüştürme varsa bu kesinlikle diğer dönüştürmelerden daha kalitelidir. Örneğin int'ten int'e dönüştürme int'ten long'a dönüştürmeden daha kalitelidir.**

2. **T1 à T2 dönüştürmesi ile T1 à T3 dönüştürmesi kıyaslanacak olsun. Eğer T2'den T3'e doğrudan dönüştürme var fakat T3'ten T2'ye doğrudan dönüştürme yoksa T1 à T2 dönüştürmesi, T1 à T3'den daha iyidir. Örneğin; char à int , char à long dönüştürmeleri kıyaslanacak olsun. char à int dönüştürmesi, char à long dönüştürmesinden daha iyidir. Çünkü int'ten long'a doğrudan dönüştürme vardır fakat long'dan int'e doğrudan dönüştürme yoktur. Örneğin int à long dönüştürmesi ile int à double dönüştürmesi kıyaslanacak olsun. int à long dönüştürmesi daha iyidir.**

3. **T1 à T2 dönüştürmesi ile T1 à T3 dönüştürmesi kıyaslanacak olsun. Eğer ne T2'den T3'e ne de T3'den T2'ye doğrudan dönüştürme yoksa bu durumda işaretli türe yapılan dönüştürme daha iyi kabul edilir. Örneğin; ushort à int dönüştürmesi ile ushort à uint dönüştürmesi kıyaslanacak olsun. İşte burada ne int türünden uint türüne ne de uint türünden int türüne dönüştürme vardır. O halde bu durum bu maddeye girmektedir. ushort à int dönüştürmesi daha kalitelidir. Çünkü işaretli türe yapılan dönüştürme tercih edilmektedir.**

4. **Decimal dönüştürmesi en kötü dönüştürme olarak belirlenmiştir. Bu nedenle örneğin; int à decimal dönüştürmesi ile int à double dönüştürmesi kıyaslanacak olsun. Bu kıyaslamayı bu madde de ele alarak, int à double dönüştürmesinin daha iyi olduğu söylenebilir.**

En uygun fonksiyon, çağrılma ifadesindeki tüm parametreler uygun fonksiyonlar yarışa sokulduğunda daha iyi bir dönüştürme sunan ya da daha kötü dönüştürme sunmayan fonksiyondur.

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

Şimdi fonksiyonun şöyle çağrıldığını varsayalım:

```
Sample.Func('x', 'y');
```

Burada 1,2,3,4,5 nolu fonksiyonların hepsi aday fonksiyonlardır. 2,3,4,5 nolu fonksiyonlar uygun fonksiyonlardır. Şimdi en uygun fonksiyonu bulmaya çalışalım. Uygun fonksiyonların her parametresi çağırma ifadesindeki parametreler ile yarışa sokulur. Örneğin birinci parametre char türdendir. char à int dönüştürmesi char à long dönüştürmesinden daha iyi olduğuna göre 2 ve 3 nolu fonksiyonlar en uygun fonksiyon olabilir. Fakat artık 4 ve 5 nolu fonksiyonlar en uygun fonksiyon olamaz. Ancak 4 ve 5 nolu fonksiyonlar yarışa devam etmektedir. İkinci parametreler yarışa sokulduğunda char à int dönüştürmesi char à long ve char à double dönüştürmelerinden daha iyi olduğuna göre 2 nolu fonksiyon en uygun fonksiyon olarak belirlenir. Çünkü 2 nolu fonksiyonun her parametresi çağrılma ifadesindeki parametreler dikkate alındığında diğerlerine göre daha iyi ya da daha kötü olmayan bir dönüştürme sunmaktadır.

Fonksiyonun şöyle çağrıldığını varsayalım:

```
Sample.Func(10, 2.3F);
```

Burada zaten uygun fonksiyon bir tanedir.(4 nolu fonk.) Dolayısıyla yarışı zaten bu fonksiyon kazanacaktır.

Fonksiyonun şöyle çağrıldığını varsayalım:

```
Sample.Func(10L, 20);
```

Burada uygun fonksiyon 4 ve 5 nolu fonksiyonlardır. En uygun fonksiyon 5 nolu fonksiyondur.

Sample sınıfının aşağıdaki gibi olduğunu varsayalım:

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

Fonksiyonun şöyle çağrıldığını varsayalım:

```
Sample.Func(10, 20);
```

Burada 1,2,3 nolu fonksiyonlar aday fonksiyonlardır. 2 ve 3 nolu fonksiyonlar uygun fonksiyonlardır. Burada en uygun fonksiyon yoktur. Dolayısıyla çağrılma error ile sonuçlanır. Birinci parametre yarışa sokulduğunda 2 nolu fonksiyon dönüştürmesi, ikinci parametreler yarışa sokulduğunda 3 nou fonksiyon dönüştürmesi daha iyidir. Dolayısıyla her parametre diğerleri ile yarışa sokulduğunda daha iyi dönüştürme sağlayan ya da daha kötü dönüştürme sağlayan fonksiyon yoktur.

Görüldüğü gibi eğer programcı çağırmayı uygularken parametre türlerini aynı yaparsa buna karşı gelen fonksiyon kesinlikle en uygun fonksiyon olarak seçilecektir.

SINIFLARIN BAŞLANGIÇ FONKSİYONLARI

İsmi sınıf ismi ile aynı olan fonksiyonlara sınıfın başlangıç fonksiyonları denilmektedir. Başlangıç fonksiyonları C# ta static de olabilir. Fakat başlangıç fonksiyonları tipik olarak static olmayan fonksiyonlardır. Static başlangıç fonksiyonu daha ileride ele alınacaktır.

Örneğin:

```
//…
class Sample
{
     public Sample()      à başlangıç fonksiyonu
     {
          //…
     }
}
//…
```

Başlangıç fonksiyonunda geri dönüş değeri kavramı yoktur. Geri dönüş değeri yerine bir şey yazılmaz. Yazılırsa error oluşur. Fakat başlangıç fonksiyonu içerisinde `return` anahtar sözcüğü kullanılabilir. Ancak yanına bir ifade yazılamaz. Sınıfın birden fazla başlangıç fonksiyonu bulunabilir. Şüphesiz bunların parametrik yapıları farklı olmak zorundadır. Örneğin:

```
//…
class Test
{
     public Test()
     {
          //…
     }
     public Test(int a, int b)
     {
          //…
     }
     public Test(int a, int b, int c)
     {
          //…
     }
}
//…
```

Sınıfın parametrik olmayan başlangıç fonksiyonuna özel olarak default başlangıç fonksiyonu denir.

Eğer bir sınıf için hiçbir başlangıç fonksiyonu yazılmamışsa, default başlangıç fonksiyonu derleyici tarafından içi boş bir şekilde yazılmaktadır. Örneğin:

```
//…
class Foo
{
     public void Func()
     {
          //…
     }
}
//…
```

Burada Foo sınıfı için programcı herhangi bir başlangıç fonksiyonu yazmamıştır. O halde default başlangıç fonksiyonu derleyici tarafından içi boş olarak yazılacaktır. Fakat örneğin:

```
//…
class Sample
{
     public Sample(int a)
     {
          //…
     }
     public void Func()
     {
          //…
     }
}
//…
```

Burada derleyici default başlangıç fonksiyonunu kendisi yazmaz. Eğer programcı hiçbir başlangıç fonksiyonu yazmamış olsaydı yazacaktı.

Sınıfın başlangıç fonksiyonu new işlemi sırasında nesne heap ta tahsis edildikten sonra derleyici tarafından otomatik olarak çağrılmaktadır. Aslında new operatöründe sınıf isminden sonra parantezler içerisinde bir parametre listesi girilebilir. Yani new operatörü ile sınıf nesnesi tahsis etmenin genel biçimi şöyledir:

```
new < sınıf ismi > ([parametre listesi])
```

Bu durumda derleyici tahsisatı yaptıktan sonra parametre listesine uygun olan başlangıç fonksiyonunu çağırır. Burada overload resolution kuralları işletilmektedir. Örneğin:

```
Sample a = new Sample();

Sample a = new Sample(10);

Sample a = new Sample(10, 20);
```

Burada üç farklı nesne üç farklı başlangıç fonksiyonu ile yaratılmıştır.

BAŞLANGIÇ FONKSİYONLARINA NEDEN GEREKSİNİM DUYULUR?

Bir nesne yaratıldığında bir takım ilk işlemlerin yapılması gerekebilir. Bu işlemler, tipik olarak sınıfın başlangıç fonksiyonu yapılırsa, nesne yaratılır yaratılmaz otomatik bir ilkleme gerçekleştirilmiş olur. Başlangıç fonksiyonu aynı zamanda sınıfın çeşitli veri elemanlarına uygun ilk değerlerin atanması amacı ile de kullanılmaktadır. Anımsanacağı gibi new operatörü tahsisatı yaptıktan sonra zaten sınıfın veri elemanlarını sıfırlıyordu. Eğer başlangıç fonksiyonunda veri elemanlarına değer atanmazsa veri elemanların da "0"

değeri bulunmaya devam edecektir. Eğer değer atama işlemi yapılırsa, atanan değer bu elemanlarda gözükecektir.

Çağrılan başlangıç fonksiyonu içerisinde kullanılan veri elemanları o anda yeni yaratılmış olan nesnenin veri elemanlarıdır.

Örneğin dosya işlemleri yapmakta kullanılan `FileStream` sınıfının başlangıç fonksiyonu, parametresi ile aldığı dosyayı açmaktadır. Örneğin:

```
FileStream fs = new FileStream ("a.dat");
```

Ya da örneğin seri port işlemlerini yapan `SerialPort` sınıfının başlangıç fonksiyonu seri portu uygun değerlerle set edebilir.

FARKLI PARAMETRİK YAPILARA İLİŞKİN AYNI İSİMLİ FONKSİYONLAR NEDEN KULLANILIR?

Bir sınıfın farlı parametrik yapılara ilişkin aynı isimli fonksiyonlarının bulunabilme durumu (`function overloading`) nesne yönelimli programlama dillerinin hemen hepsinde olan bir özelliktir.

Nesne yönelimli programlama tekniği algısal açıklık sağlamayı hedeflemektedir. Bir sınıfın benzer işlemlerini yapan fonksiyonlarına aynı isimlerin verilmesi algılamayı ve öğrenmeyi kolaylaştırmaktadır. Böylece programcı, sınıfta çok fazla fonksiyon olduğu duygusuna kapılmaz. Nesne yönelimli programlama tekniği, insan-nesne ilişkisi göz önüne alınarak tasarlanmıştır. Günlük yaşamda aslında birbirinin tam aynısı olmasa da nesnelere aynı cins isimler verilmektedir. Çünkü nesnelerin işlevleri birbirine benzerdir.

Anahtar Notlar

Nesne yönelimli programlama tekniği sınıflar kullanılarak program yazma tekniğidir. Birtakım anahtar prensipler topluluğundan oluşmaktadır. C++, Java, C# gibi diller bu tekniğin uygulanmasını mümkün hale getirmek için tasarlanmıştır. Fakat bu dillerin kullanılması, bu tekniğin iyi bir şekilde kullanılmasını garanti etmez. Bu tekniğin prensipleri iyi algılanmalı ve yazılım bu prensiplere uygun olarak tasarlanmalıdır.

Nesne yönelimli programlama tekniği bazı anahtar kavramlardan oluşmaktadır. Bu kavramlar birbiri ile iç içe geçmiş, birbirini dışlamayan kavramlardır ve bu kavramların hepsi algısal bir açıklık sağlamaya yöneliktir.

Benzer işlemleri yapan fonksiyonlara aynı isimlerin verilmesi tavsiye edilen bir durumdur. Çünkü bu durum nesne yönelimli programlama tekniğinin algısal açıklık sağlama fikri ile örtüşmektedir. .Net sınıf kütüphanesindeki sınıflarda da pek çok aynı isimli fonksiyon bulunmaktadır. Örneğin; aslında `Console` sınıfının tek bir `Write` ya da `WriteLine` fonksiyonu yoktur. Her temel tür için ayrı bir `Write` ya da `WriteLine` fonksiyonu vardır.

Biz bu fonksiyona hangi türden parametre geçersek derleyici `overload resolution` işlemi ile uygun fonksiyonu tespit etmektedir.

SINIFLARIN TEMEL ERİŞİM KURALLARI

Şimdiye kadar iki tür sınıf elemanı gördük; sınıfın veri elemanları ve fonksiyonları. Bunlar static olabilir ya da olmayabilir. Sınıf elemanlarının önüne bildirim sırasında erişim belirleyici sözcüklerden biri getirilebilir. Erişim belirleyici anahtar sözcükler yerel değişken bildirimlerinde kullanılamaz. Erişim belirleyici anahtar sözcükler şunlardan biri olabilir: `private`, `public`, `protected`, `internal`, `protected internal`.

Sınıf eleman bildiriminde erişim belirleyici anahtar sözcükler hiç yazılmayabilir. Bu durumda `private` yazılmış gibi işlem görür.

Anahtar Notlar

C++ ta sınıfın default bölümü private bölümüdür. Ancak Java da default internal biçimdedir. C# genel olarak Java ya kıyasla daha fazla C++ a yaklaştırılmıştır. C++ ta da internal ve protected internal bölümü yoktur. Ayrıca protected internal yerine internal protected yazılması da geçerlidir.

Sınıflardaki erişim kuralları birkaç madde ile özetlenebilir:

1- **Sınıfın fonksiyonu olmayan bir fonksiyon içerisinden (yani başka bir sınıfın fonksiyonu içerisinde), ilgili sınıf türünden ya da sınıf ismi yoluyla (eleman static ise) sınıfın yalnız `public` elemanlarına erişilebilir.**

2- **Sınıfın bir fonksiyonu içerisinde, erişim belirleyici ne olursa olsun sınıfın tüm elemanlarına doğrudan erişilebilir. Yani sınıf içerisinde herhangi bir koruma söz konusu değildir.**

`internal` erişim belirleyici aynı zamanda assembly içerisinde `public`, farklı bir assembly içerisinde `private` etkisi yaratır. Yani biz sınıfın `internal` elemanına aynı assembly içerisindeki başka bir sınıftan erişebiliriz. Çünkü bu durumda `internal`, `public` gibi etki yapmaktadır. Fakat `internal` bir elemana başka bir assembly içerisindeki elemandan ulaşılamaz. Sınıfın `protected` ve `protected internal` elemanları ancak türetme durumu söz konusu olduğunda `private` elemanlardan farklı bir erişime sahip olur. Bu nedenle `protected` ve `protected internal` elemanlar türetme konusunda ele alınacaktır.

Sonraki anlatımlarda "sınıfın public bölümü", "sınıfın private bölümü" biçiminde ifadeler kullanılacaktır. Sınıfın public bölümü demek, sınıfın tüm public elemanları, sınıfın private bölümü demekle sınıfın tüm private elemanları anlaşılmaktadır. Yani sanki sınıf bölümlerden oluşuyormuş ta bu elemanlar o bölümdeymiş gibi...

ERİŞİM KURALLARININ ANLAMI

Sınıfın bir elemanını `private` bölüme yerleştirmekle biz o elemanı dışarının kullanımına kapatırız. Yani o eleman ancak sınıf içerisinden kullanılabilir durumda olur. Halbuki biz bir elemanı `public` bölüme yerleştirirsek o elemanı herkesin kullanımına sokarız.

Bir sınıf için iki bakış açısı söz konusudur:

1- **Sınıfı Kullanan Kişinin Bakış Açısı: Sınıfı kullanan kişi, sınıfın `public` bölümüne yoğunlaşmalıdır. Zaten diğer bölümleri kullanmayacaktır. Yani sınıfı kullanan kişi için sınıf adeta `public` bölüm kadardır.**
2- **Sınıfı Tasarlayanın Bakış Açısı: Sınıfı tasarlayan kişi sınıfın her bölümünü bilmek zorundadır.**

Nesne yönelimli programlama tekniği, insan-nesne ilişkisi modellenerek oluşturulmuştur. Yani örneğin; kullandığımız pek çok nesnede de `public` ve `private` bölüm vardır.

Bir sınıf tasarlanırken yalnızca dışarının ilgisini çekecek olan elemanları `public` bölümüne yerleştirmeliyiz. Sınıfın iç işleyişine ait olan, sınıfı kullanan kişilerin bilmesine gerek olmayan elemanlar, sınıfın `private` bölümüne yerleştirilmelidir. Bu çabaya nesne yönelimli programlama tekniğinde `kapsülleme(encapsolution)` denilmektedir. Kapsülleme sınıfın iç işleyişine ilişkin özelliklerini dışarıya gizleyerek kullanımı az sayıda `public` elemana bırakma anlamındadır. Şüphesiz sınıfın iç işleyişine ilişkin elemanlarının `private` bölüme yerleştirilmesi hem algısal açıklık sağlarken hem de dışarıdan yapılacak bozucu etkilere karşı sınıfı korumaktadır. Yani örneğin; arabanın `private` bölümünü oluşturan bölümler bir kaputun altına gizlenmeseydi bunların kurcalanmasına açık bir durum oluşurdu ve araba daha kolay bozulabilirdi.

Örneğin bir sınıfın dışarıdan çağrılacak `DoSometingImportent` isimli önemli bir fonksiyonu olsun. Fakat bu fonksiyon bu önemli işi yaparken `Func1()`, `Func2()`, `Func3()` isimli, işin parçalarını yapan yardımcı fonksiyonları çağırıyor olsun:

```
//…
class Sample
{
     public void DoSometingImportent
     {
          Func1()
          {
               //…
          }
          Func2()
          {
               //…
          }
```

```
        Func3()
        {
            //…
        }
    }
}
//…
```

Buradaki `Func1()`, `Func2()`, `Func3()` fonksiyonları `private` bölüme yerleştirilerek gizlenmelidir.

SINIFIN VERİ ELEMANLARINI GİZLENMESİ

Sınıfın veri elemanları aslında sınıfın iç işleyişine ilişkindir. Bu nedenle `private` bölüme yerleştirilmelidir. Sınıfın veri elemanları `private` bölüme alınarak gizlenmesine nesne yönelimli programlama tekniğinde veri elemanlarının gizlenmesi(data hiding) denilmektedir. Veri elemanlarının gizlenmesi, nesne yönelimli programlama tekniğinin anahtar kavramlarından birisidir. Şüphesiz "data hiding" kavramı, "encopsolution" kavramından bütünüyle farklı değildir. Onun bir yönünü belirtmektedir.

Bir sınıf için, sınıfın kendi kodlarından ve sınıfı kullanan kodlardan bahsedilebilir. Sınıfın veri elemanları daha sonra değiştirilmeye yatkın elemanlardır. Yani sınıfın daha sonraki versiyonlarında bu elemanlar tür ve isim bakımından değiştirilebilir. Eğer biz veri elemanlarını sınıfın `public` bölümüne yerleştirirsek onları dışarıdan kullanabilirler. Daha sonra bu veri elemanları değiştirildiğinde o kodlar geçersiz hale gelebilir.

Sınıfın veri elemanlarını `private` bölüme yerleştirdikten sonra yine de bazı veri elemanlarına dışarıdan erişilmesi anlamlı ve gerekli olabilir. İşte bu durumda, bu veri elemanının değerini veren ve bu elemana değer atayan public fonksiyonlar bulundurulur. Bu tür fonksiyonlara "`Get/Set`" fonksiyonları ya da "erişimci fonksiyonlar" denir. C# ta bu tür `Get/Set` fonksiyonlarını kolay yazmak için property kavramı uydurulmuştur. Property elemanlar, `private` elemanlara ulaşımı sağlayan `Get/Set` fonksiyonlarıdır. Java da ve C++ ta property kavramı yoktur ve bu fonksiyonlar açıkça yazılmak durumundadır. Örneğin; `Date` isimli bir sınıfın `day`, `mount`, `year` isimli üç veri elemanı olsun. Aynı zamanda programcının bu elemanlara dışarıdan erişilmesini de istediğini varsayalım. Burada programcı veri elemanlarını `private` bölüme yerleştirmeli ve onlara `public Get/Set` fonksiyonları ile erişimi sağlamalıdır:

```
//…
class Date
{
    private int day;
    private int mount;
    private int year;

    public int GetDay()
    {
        return day;
```

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

Görüldüğü gibi veri elemanlarına doğrudan değil public fonksiyonlar yoluyla erişilmektedir.

Sınıfı Kullanan Kişi → **public erişimci fonksiyonlar** → **private veri elemanları**

Bir dilin nesne yönelimli olabilmesi için üç özelliğin dilde bulunması gerekir:

1- **Dilde sınıf kavramının bulunması gerekir.**
2- **Dilde sınıfların türetilmesi kavramının bulunması gerekir.**
3- **Dilde çok biçimlilik kavramının bulunması gerekir.**

Eğer dilde ilk iki kavram var fakat üçüncü kavram yoksa dile nesne yönelimli değil nesne tabanlı(object based) denir. C++, Java ve C# ta bu özelliklerin hepsi vardır. Bu nedenle bu diller nesne yönelimli dillerdir. Fakat örneğin Visual Basic'in eski versiyonu olan VB 6.0'da çok biçimlilik kavramı olmadığı için bu dil nesne tabanlıdır. Fakat VB.Net nesne yönelimlidir.

VERİ ELEMANLARININ PRIVATE BÖLÜME YERLEŞTİRİLİP ONLARA PUBLIC ERİŞİMCİLER İLE ERİŞİLMESİNİN ANLAMI

Veri elemanlarının private bölüme yerleştirilerek dışarı ile ilişkisinin kesilmesi ve bunlara gerektiğinde public get/set fonksiyonları ile erişilmesi uzun dönemde oldukça faydalı sonuçlar doğurmaktadır:

1- **Veri elemanları private bölüme yerleştirilirse sınıfı kullanan kişiler bunlara erişemez. Günün birinde bu veri elemanlarının genel yapısında bir değişiklik yapıldığında daha önce yazılmış olan sınıfı kullanan kodlar**

bu değişikliklerden etkilenmez. Çünkü veri elemanlarına get/set arayüz fonksiyonlarla erişilmiştir. Tabi programcının bu get/set fonksiyonlarının içini, parametrik yapıda değişiklik oluşturmadan yeni duruma uygun bir biçimde yazması gerekir. Örneğin; yukarıda yazılmış olan Date sınıfını kullanan aşağıdaki gibi bir kod söz konusu olsun:

```
//…
Date date = new Date();

date.SetDay(10);
date.SetMonth(12);
date.SetYear(2000);
//…
if (date.GetMonth() == 12)
{
     //…
}
//…
```

Şimdi Date sınıfını yazan şirketin tarih bilgisini üç tane ayrı int değişkende değil de "dd/mm/yyy" formatında tek bir string nesnesinde tutmaya karar verdiğini düşünelim.

```
//…
class Date
{
     string date;
}
//…
```

Şirketin tek yapacağı bu get/set fonksiyonlarının içini string ile çalışacak şekilde değiştirmektedir. Sınıfı kullanan kodlarda ki get/set fonksiyonlarının çağırmaları, aynı geçerli anlamını koruyacaktır.

2- Sınıfın birbirleri ile ilişkili olan veri elemanları bulunabilir. Bu ver elemanlarının birinin değiştirilmesi diğer bazı veri elemanlarının değiştirilmesini gerektirebilir. Eğer veri elemanlarını public bölüme yerleştirirsek bütün bu değişiklikler sınıfı kullanan kişi tarafından yapılmak zorundadır. Bu da sınıfın kullanımını zorlaştırır. Halbuki veri elemanlarını private bölümde tutup bunlara public get/set fonksiyonları ile erişirsek sınıfın veri elemanları arasındaki bu uyum set fonksiyonu içerisinden sağlanabilir. Böylece veri elemanları arasındaki ilişkiyi sınıfı kullanan kişi bilmek zorunda kalmaz. Örneğin; dairesel işlemler yapan Circle isimli bir sınıfın veri elemanları, dairenin merkez koordinatları ve yarıçapı olabilir. Fakat sınıfın içinde dairenin alanı pek çok yerde gerekiyorsa programcı sınıfın veri elemanı olarak dairenin alanını tutan bir değişken bulundurmak isteyebilir. Böylece sınıfı tasarlayan kişi dairenin alanını kullanacağı yerlerde, her zaman $pi\ r^2$ işlemini yapmak yerine, yarıçap set edilirken bir kere dairenin alanını tutan değişkene değer atayıp bunu kullanmak isteyebilir. Görüldüğü gibi eğer biz yarıçapı tutan veri elemanını public bölüme yerleştirirsek,

bunu değiştirecek kişinin alan değişkenini de değiştirmesi gerekir. Halbuki biz yarıçap değişkenini `private` bölümde tutup onu bir set fonksiyonu ile değiştirmeye çalışırsak o fonksiyon içerisinde alan değişkenine gizlice yeni değerini atayabiliriz.

```
class Circle
{
     private double centerx, centery;
     private double radius;
     private double area;
     //…
     public Circle(double x, double y, double r)
     {
          centerx = x;
          centery = y
          radius = r;
          area = Math.PI * r * r;
     }
     public double GetRadius()
     {
          return radius;
     }
     public void SetRadius()
     {
          radius = r;
          area = Math.PI * r * r;
     }
}
```

3- Bir veri elemanı set edilirken arka planda başka işlerinde yapılması gerekebilir. İşte bu işlemler gizlice `set` fonksiyonu içerisinde yapılabilir. Ya da benzer biçimde bir veri elemanının değeri alınırken de başka işlemlerin yapılması gerekebilir. Örneğin seri port işlemlerini yapan `SerialPort` isimli bir fonksiyon olsun. Portun hızı sınıfın bir veri elemanında tutuluyor olabilir. Yalnızca bu veri elemanının set edilmesi portun hızının ayarlanacağı anlamına gelmez. İşte portun hızını `set` fonksiyonu, hem bu veri elemanına yeni değerini atarken hem de bir takım sistem fonksiyonları ile portun hızını gerçekten set edebilir.

SINIFIN PROPERTY ELEMANLARI

Sınıfın property elemanları aslında bir veri elemanına dışarıdan erişmeyi sağlayan `get/set` fonksiyonlarından başka bir şey değildir. Bir property elemanının genel biçimi şöyledir:

```
[erişim belirleyicisi] [static] <tür> <isim>
{
     get
     {
          //…
     }
     set
     {
          //…
     }
}
```

Aslında property elemanı, veri elemanı gibi yazılır fakat genel biçiminden de görüldüğü gibi ; yerine blok açılır. Bir property elemanı get ve set olmak üzere iki bölümden oluşur. Property elemanının yalnızca get bölümü olabilir. Ya da hem get bölümü hem de set bölümü aynı yazılabilir. get ve set bölümlerinin hangi sırada yazıldığının bir önemi yoktur. Yalnızca get bölümü olan property "Read Only", yalnızca set bölümü olan property "Write Only", hem get hem set bölümü olan property elemanı "Read/Write" property denir.

Property elemanı onlar sanki bir veri elemanıymış gibi erişilmektedir. Örneğin; r sınıf türünden bir referans, A da bu sınıfın bir property elemanı olmak üzere erişim r.A şeklinde yapılır.

Bir property elemanı bir ifade içerisinde kullanıldığında, ya bir değer alma ya da değer atama amaçlı kullanılır. Örneğin:

```
//…
x = r.A + 1;              à değer alma amaçlı kullanılmıştır.
r.A = 10;                 à değer atama amaçlı kullanılmıştır.
Console.WriteLine(r.A);   à değer alma amaçlı kullanılmıştır.
//…
```

Fakat istisna olarak eğer property elemanı ++ ve -- operatörlerinin bir operandı biçiminde kullanılmışsa onun hem değer alma hem de değer yerleştirme amaçlı kullanıldığı kabul edilir. Örneğin:

```
++r.A à hem değer alma hem de değer atama amaçlı kullanılmıştır.
```

Property elemanının get bölümü sanki bir get fonksiyonu gibi, set bölümü ise sanki bir set fonksiyonu gibidir. Eğer property elemanı değer alma amaçlı kullanılmışsa onun get bölümü, değer atama amaçlı kullanılmışsa onun set bölümü çalıştırılır. Eğer ++ ve -- operatörleri ile kullanılmışsa önce get bölümü sonra set bölümü çalıştırılır.

Propertynin get bölümü, parametresi olmayan geri dönüş değeri property türünden olan bir fonksiyon gibidir. Bu nedenle get bölümünde return kullanılmak zorundadır. Programcı get bölümünde tipik olarak propertynin ilişkili olduğu geri dönüş elemanının türü ile geri döner. Tabi get bölümünde başka tamamlayıcı işlemlerde yapılabilir. Örneğin:

```
//…
class Sample
{
     private int a;
     private int b;

     get
     {
          //…
          return a;
```

```
    }
    set
    {
        //…
    }
}
//…
```

Property elemanı değer alma amaçlı kullanıldığında elemanın get bölümü çalıştırılır. Burada elde edilen geri dönüş değeri işleme sokulur. Örneğin:

```
Sample s = new Sample();
//…
x = s.A + 1;        à C# ta kullanım

x = s.GetA() + 1;   à Java da kullanım
```

Property elemanının set bölümü, parametresi property türünden olan geri dönüş değeri void olan bir fonksiyon gibidir. Set bölümünde "value" anahtar sözcüğü, bu set bölümünün parametresini temsil eder. Value anahtar sözcüğü yalnızca set bölümünde kullanılır, get bölümünde kullanılmaz.

Set bölümündeki value anahtar sözcüğü, property elemanına yerleştirilecek sonuç ifadesini temsil etmektedir. O halde programcı tipik olarak set bölümünde value anahtar sözcüğünü ilgili private elemanına atamalıdır. Tabi önceki konularda da söz edildiği gibi set işlemi sırasında başka işlemlerde yapılabilir. Örneğin:

```
Sample s = new Sample();
//…
s.A = x + y;
```

Burada propertye atanmak istenen değer x + y ifadesidir. Yani x + y değeri hesaplanacak, sonra propertynin set bölümü çalıştırılacak, set bölümünün value anahtar sözcüğü de, x + y değerini temsil edecektir. Aynı işlem C++ ya da Java da aşağıdaki gibi yapılırdı:

```
s.SetA(x + y);
```

Bu durumunda property şöyle yazılacaktır:

```
//…
class Sample
{
    private int a;
    private int b;

    get
    {
        //…
        return a;
```

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

```
using System;
namespace CSD
{
     class App
     {
          public static void Main()
          {
               Sample s = new Sample(10);
               Console.WriteLine(s.A);
               s.A = 100 + 200;
               Console.WriteLine(s.A);
          }
     }
     class Sample
     {
          private int a;
          public Sample()
          {
          }
          public Sample(int x)
          {
               a = x;
          }
          public int A
          {
               get
               {
                    return a;
               }
               set
               {
                    a = value;
               }
          }
     }
}
```

Aslında property kavramı ile daha öncede karşılaşmıştık. Örneğin; string sınıfının Lenght isimli property elemanı yazının uzunluğunu vermektedir. Yani muhtemelen string sınıfı içerisinde sadece yazı değil private bir eleman da yazının uzunluğunu tutmaktadır. Lenght property elemanı get bölümünde yine muhtemelen bu veri elemanının değerini vermektedir. Örneğin:

```
string s = “Ankara”;
Console.WriteLine(s.Lenght);
```

Burada Length property elemanının get bölümü çalıştırılır. Get bölümünde elde edilen değer yazdırılmıştır.

"Read Only" bir property(yalnızca get bölümü olan property) değer atama amaçlı kullanılmaz. Benzer biçimde "Write Only" property değer alma amaçlı bir ifade de kullanılmaz. Örneğin; Lenght property elemanı read only bir propertydir.

Anımsanacağı gibi diziler aslında birer sınıftır ve tüm dizi sınıflarının Lenght isimli read only bir property elemanı vardır. Bu property elemanı dizinin uzunluk değerini vermektedir.

Şüphesiz bir property elemanı yalnızca sınıfın private elemanıyla ilişki kuran bir fonksiyon biçiminde olmak zorunda değildir. Örneğin; property elemanında set işlemi yapılırken aslında arka planda başka işlemlerde yapılıyor olabilir.

Aslında bir property elemanının private veri elamanını hedef alması gibi bir zorunluluk yoktur. .Net GUI programlama modelinde çeşitli GUI sınıflarının pek çok property elemanı vardır. Bunlara set işlemi yapıldığında arka planda bu propertylerin set bölümleri, çeşitli faydalı ve karışık işlemleri yapmaktadır. Böylece bir .Net programında yalnızca çeşitli propertyleri set ederek program bir dereceye kadar oluşturulabilmektedir. Property kavramı görsel programlama fikrini de kuvvetlendirmektedir.

STATIC PROPERTYLER

Anımsanacağı gibi sınıfın static veri elemanının toplamda tek bir kopyası vardır. İşte static veri elemanı da sınıfın private bölümüne yerleştirilmeli ve onlara public static propertyler ile erişilmelidir. Static propertylerin get ve set bölümlerinde, sınıfın static olmayan veri elemanları ve fonksiyonları doğrudan kullanılamaz. Static propertylere, sınıf ismi ile erişilir. Örneğin:

```
class Sample
{
     private static int m_a;
     public static int A
     {
          get
          {
               return m_a;
          }
          set
          {
               m_a = value;
          }
     }
}
```

Erişim şöyle yapılabilir:

```
public static void Main()
{
     Sample.A = 10;
     Console.WriteLine(Sample.A);
}
```

SINIFLARIN TÜRETİLMESİ

Türetme (inheritence) daha önce yazılmış bir sınıfın genişletilmesi için kullanılan bir tekniktir. Örneğin; elimizde zaten yazılmış olan bir A sınıfı bulunuyor olsun. Biz bu A sınıfına çeşitli fonksiyonlar eklemek istiyoruz. İlk akla gelecek yöntem eğer A nın kaynak kodları varsa A nın orijinal biçimini bozmak istemediğimiz için ondan B gibi bir kopya çıkarmak ve eklemeleri bu kopya üzerinde yapmaktır. Fakat bu yöntem zayıf bir yöntemdir. A nın kaynak kodları elimizde olmayabilir. Elimizde olsa bile B sınıfında gereksiz bir biçimde A sınıfındaki aynı elemanlar bulunacaktır.

Türetme yönteminde mevcut bir A sınıfına ekleme yapılmak isteniyorsa bir B sınıfı A sınıfında türetilir. B sınıfına yalnızca eklenecek öğeler yerleştirilir. Böylece B sınıfı hem tamamen A sınıfı gibi davranır hem de eklentilere sahiptir. Burada eklenti yapılmak istenen sınıfa (yani A sınıfına) taban sınıf(base class), eklentilerin yerleştirileceği sınıfa (yani B sınıfına) türemiş sınıf denilmektedir. Türetme işlemi UML gibi sınıf diyagramlarında türemiş sınıftan taban sınıfa çekilen bir okla gösterilmektedir.

Türetme işleminin genel biçimi şöyledir:

```
class <türemiş sınıf>  :  <taban sınıf>
```

Örneğin:

```
class A
{
     //…
}
class B:A
{
     //…
}
```

Türemiş sınıf türünden bir referans, dışarıdan türemiş sınıf ile taban sınıfın tüm public elemanlarına erişilebilir. Örneğin:

```
//…
class App
{
     public static void Main()
     {
          B b = new B();
          b.Foo();
          b.Bar();
     }
}
class A
{
     public void Foo()
     {
          Console.WriteLine(“Foo”);
     }
     //…
}
class B:A
{
     public void Bar()
     {
          Console.WriteLine(“Bar”);
     }
     //…
}
//…
```

Şüphesiz türetme yapabilmek için taban sınıfın kaynak kodlarının elde bulunuyor olması zorunlu değildir. Örneğin taban sınıf bir dll içerisinde olabilir. Biz de o dll ye referans ederek türetmeyi yapabiliriz. Ya da örneğin taban sınıf başka bir isim alanın da bulunabilir. Bu durumda taban sınıf isminin, isim arama kurallarına göre bulunabilecek şekilde belirtilmesi gerekir. Örneğin:

```
namespace X
{
     class A
     {
          //…
     }
}
namespace CSD
{
     class B : X.A
     {
          //…
     }
     //…
}
```

TÜREMİŞ SINIFLARDA ERİŞİM KURALLARI

Daha önce türemiş sınıfın taban sınıf gibi de davranabildiğini belirtmiştik. Türemiş sınıfın taban sınıfa erişimi için şu kurallar geçerlidir:

1- **Türemiş sınıf türünden bir referans yoluyla ya da türemiş sınıf ismi ile (static ise), türemiş sınıf fonksiyonu olmayan bir fonksiyon içerisinden, türemiş sınıfın yalnızca public bölümüne erişilebilir.**

2- **Türemiş sınıf fonksiyonları içerisinde anımsanacağı gibi türemiş sınıfın her bölümüne doğrudan erişilebilir. İşte aynı zamanda türemiş sınıf fonksiyonları içerisinde taban sınıfın public ve protected bölümlerine doğrudan erişilebilir.**

3- **Taban sınıfın private bölümüne ne dışarıdan ne de türemiş sınıftan erişilebilir.**

PROTECTED BÖLÜMÜN ANLAMI

Public bölüm sınıfın herkese açık olan korunmamış bölümüdür. Public bölümdeki elemanlar herkes tarafından her zaman kullanılabilir.

Sınıfın private bölümü tam olarak korunmuş bölümdür. Private bölüme yerleştirilen elemanlar yalnızca sınıf içerisinden kullanılabilir. Dışarıdan ya da türemiş sınıftan kullanılamaz.

Protected bölüm dışarıya kapalı fakat türemiş sınıfa açık olan bölümdür. Biz bir elemanı protected bölüme yerleştirirsek onu dışarıdan kullanamayız. Fakat sınıfın kendi içerisinden ve türemiş sınıflarda kullanabiliriz.

Sınıfın internal bölümü daha önce de belirtildiği gibi aynı assembly içerisinde public, farklı bir assembly içerisinden erişimde private gibi davranır. Internal çok fazla kullanılmaz.

Protected internal bölüm, aynı assembly içerisinden erişimlerde public (tıpkı internal bölümde olduğu gibi), farklı bir assembly den erişimlerde protected etkisi yaratan bölümdür. Protected internal bölüm de çok seyrek kullanılmaktadır.

Sınıfın hangi elemanlarının protected bölüme yerleştirmeliyiz? Sınıfın dışarıyı ilgilendirmeyen içsel, içsel işleyişine ilişkin olabilecek fakat türemiş sınıfı yazanlar tarafından gereksinim duyulabilecek elemanları protected bölüme yerleştirmeliyiz. Programcı daha sınıfı tasarlarken ondan türetme yapılıp yapılmayacağını göz önünde bulundurmalı ve bazı elemanları eğer türetme yapılırsa, türemiş sınıfl yazanlar tarafından kullanılsın diye protected bölüme yerleştirilmeli.

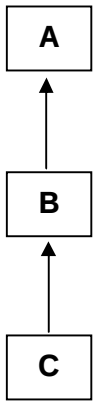
Başkaları tarafından yazılmış bir sınıf olduğunu düşünelim (.Net Kütüphanesi). Bir sınıfın dokümantasyonun da kesinlikle public, protected,

protected internal bölümdeki elemanlar açıklanmalıdır. Public bölüm herkes için, protected ve protected internal bölümler o sınıftan türetme yapmak için dokümante edilmelidir.

BİR DİZİ TÜRETME YAPILMASI DURUMU

Türemiş bir sınıftan tekrar türetme yapılabilir. Örneğin:

```
//…
class A
{
    //…
}

class B:A
{
    //…
}

class C:B
{
    //…
}
//…
```



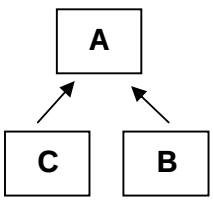
Burada C sınıfı, hem B gibi hem de A gibi davranabilmektedir. Yani dışarıdan C sınıfı türünden bir referans yoluyla ya da C sınıf ismi yoluyla (static olması durumunda) hem C sınıfının, hem B sınıfının hem de A sınıfının public elemanlarına erişilebilir. Benzer biçimde C sınıfının bir fonksiyonu içerisinden doğrudan hem B sınıfının hem de A sınıfının public ve protected elemanlarına erişilebilir.

Anahtar Notlar

Nesne yönelimli programlama tekniğinde bir şeyin sıfırdan yazılması yerine daha önce yazılmış olanlarda faydalanılması önerilen bir davranıştır. Örneğin; sıfırdan bir sınıf yazmak yerine az çok gereksinimimizi karşılayan daha önce yazılmış olan bir sınıfı kullanarak ya da o sınıftan türetme yaparak bu yeni sınıfı oluşturmak iyi bir tekniktir. Daha önce yazılmış olan kodlardan faydalanma teması nesne yönelimli programlama tekniğinin anahtar kavramlarındandır. Buna yeniden kullanılabilirlik denilmektedir.

Bir sınıf birden fazla sınıfın taban sınıfı olabilir. Örneğin:

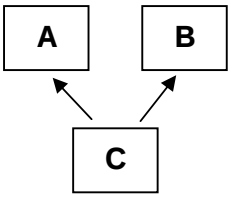
```
//…
class A
{
    //…
}

class B:A
{
```



```
    //…
}

class C:A
{
    //…
}
//…
```

Burada B ile C arasında bir ilişki yoktur.

Yukarıdaki durumun tersi özel bir durumdur. Bir sınıfın birden fazla taban sınıfı olabilir mi?



Burada C sınıfının iki taban sınıfı vardır. Bu türetme işlemine nesne yönelimli programlama tekniğinde çoklu türetme denilmektedir. Ancak maalesef C# ve Java çoklu türetmeyi desteklememektedir. C# ve Java da bir sınıfın tek bir taban sınıfı olabilir.

NE ZAMAN TÜRETME UYGULANMALIDIR?

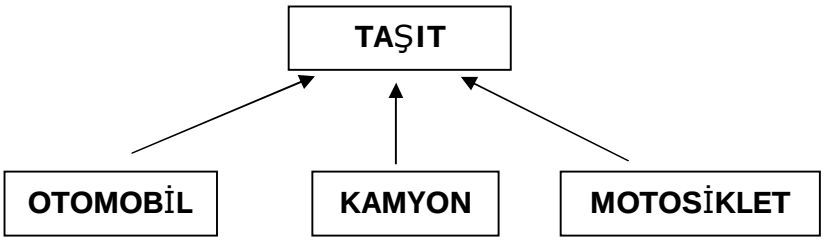
Bir B sınıfı yazma isteyelim. Kabaca üç seçenek söz konusu olabilir:

1. **B sınıfını sıfırdan yazabiliriz.**
2. **Elimizde bir A sınıfı vardır. A sınıfından türetme yaparak yazabiliriz.**
3. **Elimizde bir A sınıfı vardır. Biz B sınıfının içerisinde, A sınıfı türünden bir veri elemanı alırız. B sınıfının başlangıç fonksiyonunda A yı yaratırız. Sonra B nin fonksiyonlarını bu A nesnesini kullanarak yaratırız.**

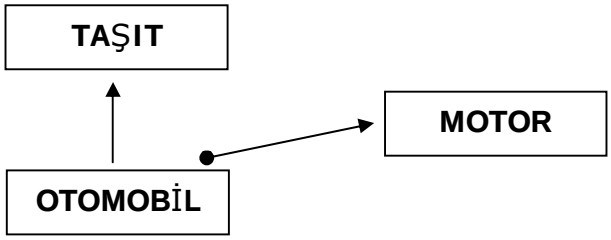
```
class B
{
     private A a;
     public B()
     {
          a = new A();
          //…
     }
     //…
}
```

Eğer B bir çeşit A ise fakat birtakım fazlalıkları varsa bu durumda B A dan türetilerek yazılmalıdır. Örneğin; ister otomobil olsun, ister kamyon olsun, isterse motosiklet olsun her motorlu taşıtın ortak birtakım özellikleri vardır.

Bir taşıtın bütün bu ortak özellikleri, taşıt isimli bir sınıf ile temsil edilebilir. Örneğin; her taşıtın bir ruhsat bilgisi, bir plakası, bir motoru vardır. Tüm bu bilgiler ve özellikler taşıt sınıfının özellikleri olabilir. Şimdi bir otomobil sınıfı yazmak isteyelim. Otomobil de bir çeşit taşıttır. Bu durumda otomobil sınıfının sıfırdan değil taşıt sınıfından türetilmesi gerekir. Bir kamyon sınıfı yazacak olsak ta kamyon da bir çeşit taşıt olduğuna göre kamyon sınıfı da taşıt sınıfında türetilmelidir.



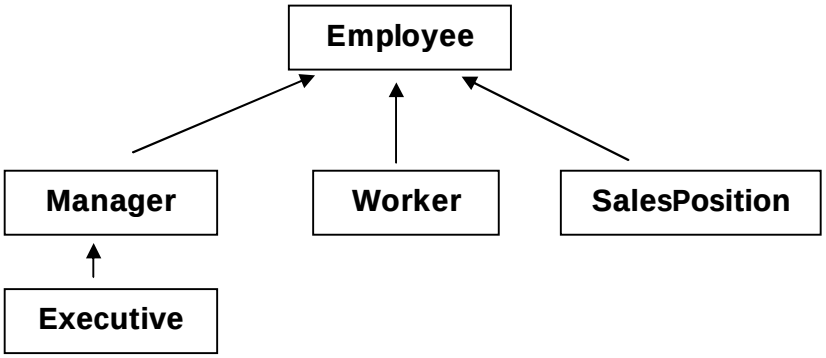
Eğer `B` bir çeşit `A` değilse, `A` `B` nin parçalarından bir ise ya da `B` yi yazarken `A` yı kullanıyor isek bu durumda türetme uygulanmamalıdır. Örneğin; genel olarak bir taşıt motoru üzerinde işlem yapan, motor isimli bir sınıf olsun. Otomobil bir çeşit motor değildir. O halde otomobil sınıfını yazarken motor sınıfından türetme yapmamalıyız. Otomobil sınıfını yazarken motor sınıfından yararlanmalıyız. Bu tür kapsama ya da kullanma işlemine nesne yönelimli programlama tekniğinde "`composition`" ve "`aggregation`" denir. Bu ilişki UML gibi araçların sınıf diyagramlarında, kullanan ya da içeren sınıf tarafında içi dolu bir yuvarlak ya da baklava olacak bir biçimde çekilen ok ile gösterilir.



TÜRETME İŞLEMİNE ÇEŞİTLİ ÖRNEKLER

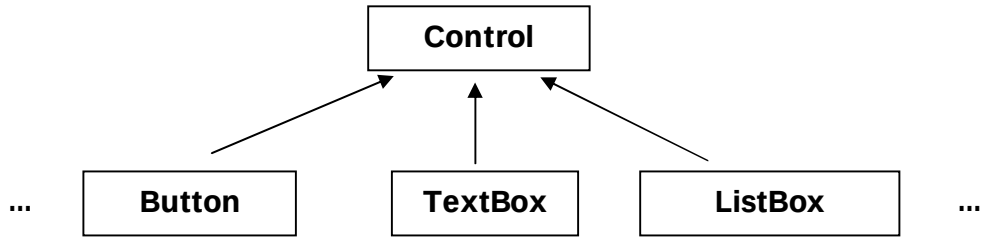
1- **Bir işletme ile ilgili bir program yazılacak olsun. Nesne yönelimli programlama tekniğinde, bir projeye konu olan tüm gerçek nesneler ve kavramlar sınıflarla temsil edilir ve program bu sınıfları kullanarak yazılır. Bu bağlamda işletmede çalışan kişiler görevlerine göre sınıflarla temsil edilecek olsun. Tüm çalışanların bilgileri ve çalışan kim olursa olsun onun üzerinde işlem yapan genel fonksiyonlar "`Employee`" isimli bir sınıf ile temsil edilebilir. İşçiler bir çeşit çalışandır. Fakat bir işçinin işçi olmasından kaynaklanan bir takım farklı özellikleri vardır. İşçiler "`Worker`" sınıfı ile temsil edilebilir. `Worker` sınıfı `Employee` sınıfından türetilebilir. Yöneticiler de bir çeşit çalışandır. O halde "`Manager`" sınıfı da `Employee` sınıfında türetilebilir. Üst düzey yöneticiler de bir çeşit**

yöneticidir. O halde "Executive" sınıfı da Manager sınıfından türetilebilir. Türetme şemasını bir şekille belirtirsek:



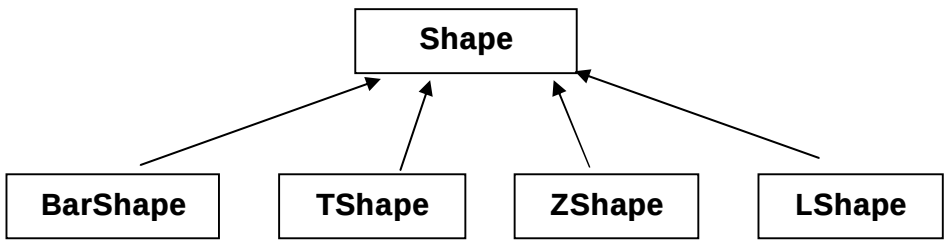
Bir türetme şemasında yukarı çıkıldıkça genelleşme, aşağı inildikçe özelleşme oluşur.

2- Windows GUI programlamada ekranda kontrol edilebilen bağımsız alanlara pencere (window) denir. Örneğin; programın ana penceresi, düğmeler, edit alanları, seçenek butonları, listeleme kutuları hep birer penceredir. Her pencerenin ortak bazı özellikleri vardır. Örneğin; her pencerenin (pencere ne olursa olsun) bir zemin rengi söz konusudur. Her pencerenin bir boyutu ve konumu vardır. İşte .Net sınıf kütüphanesinde tüm bu farklı pencereler farklı sınıflarla temsil edilmiştir. Tüm pencerelerin ortak özellikleri Control isimli bir sınıfta toplanmıştır. Pencere sınıfları bir Control sınıfında türetilmiştir.



Görüldüğü gibi tüm pencere sınıflarında Control elemanları ortaktır. Bu sınıf sisteminin öğrenilmesinde öncelikle ortak elemanlara yoğunlaşılması daha anlamlıdır.

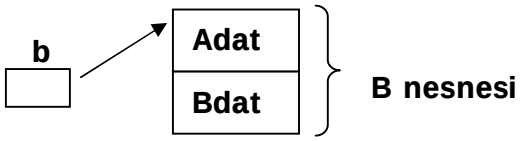
3- Bir tetris programı yazdığımızı düşünelim. Tetris oyunundaki pek çok kavram ve öğe birer sınıfta temsil edilebilir. Örneğin; oyundaki her bir şekil ayrı bir sınıfta temsil edilebilir. Şekiller birbirlerinden farklı olmasına karşın zemin renkleri, konumları gibi ortak özellikleri vardır. O halde bu ortak özellikler Shape isimli bir sınıfta toplanabilir ve bu spesifik şekiller Shape sınıfından türetilmiş sınıflarla temsil edilebilir.



TÜREMİŞ SINIFLARDA VERİELEMANLARININ DURUMU

Türemiş sınıf türünden bir referans yaratıldığında, türemiş sınıf nesnesi yalnızca kendi static olmayan veri elemanlarını değil taban sınıfın static olmayan veri elemanlarını da içerir. Taban sınıfın veri elemanları ile türemiş sınıfın veri elemanları, peşi sıra bir blok oluşturmaktadır. Taban sınıfın veri elemanları yukarıda (yani daha düşük adreste), türemiş sınıfın veri elemanları aşağıda (yani daha yüksek adreste) bir blok oluşturacak şekilde bulunur. Örneğin B sınıf A sınıfında türemiş olsun:

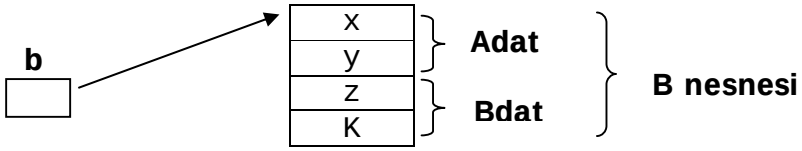
```
B b = new B();
```



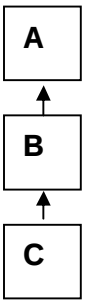
Burada Adat demekle A sınıfının static olmayan veri elemanlarını, Bdat demekle B sınıfının static olmayan veri elemanları kastedilmektedir. Örneği biraz daha somutlaştırabiliriz:

```
class A
{
     private int x;
     private int y;
     //…
}
class B:A
{
     private int z;
     private int k;
     //…
}
```

```
B b = new B();
```

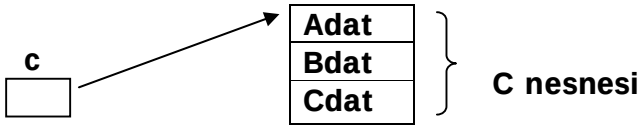


C sınıfı B sınıfından, B sınıfı da A sınıfından türetilmiş olsun:



Şimdi C sınıfı türünden bir nesne yaratalım:

```
C c = new C();
```



this ANAHTAR SÖZCÜĞÜ

Aslında bilgisayarın çalışma prensibi göz önüne alındığında static olmayan fonksiyon kavramı yapay bir kavramdır. Nesne yönelimli programlama dillerinde kolaylık sağlasın diye uydurulmuştur. Aslında biz her zaman static bir fonksiyonu sanki static olmayan bir fonksiyon gibi kullanabiliriz. Bunun için tek yapacağımız şey, static fonksiyona aynı sınıf türünden bir parametre geçmek ve sınıfın static olmayan veri elemanlarına bu parametre yoluyla erişmektir. Örneğin:

```
class Sample
{
     private int a;
     private int b;

     public static void Bar(Sample s)
     {
          s.a = 10;
          s.b = 20;
     }
     //…
}
```

Fonksiyonu şöyle çağırabiliriz:

```
Sample s = new Sample();

Sample.Bar(s);
```

Aynı işlemin eşdeğeri static olmayan bir fonksiyon ile şöyle yapılabilir:

```
class Sample
{
     private int a;
     private int b;

     public void Bar(s)
     {
          a = 10;
          b = 20;
     }
     //…
}
```

```
Sample s = new Sample();

s.Bar();
```

Görüldüğü gibi her static olmayan fonksiyon aslında ek bir parametre ile static fonksiyon biçimine dönüştürülebilir.

Aslında static olmayan fonksiyon yapay ve uydurma bir kavramdır. Static fonksiyonlar gerçek bir kavramdır. Derleyici aslında kod üretimi yaparken static olmayan fonksiyonları, onlara ek bir parametre geçirerek static fonksiyon gibi ifade etmektedir. Aslında static olmayan fonksiyonların içerisinde sınıfın veri elemanlarına doğrudan eriştiğimizde, derleyici aslında bu elemanlara geçirilen bu gizli parametre yoluyla erişmektedir. Örneğin; aslında static olmayan bir fonksiyonun 0 parametresi varsa gerçekte 1 parametresi, 1 parametresi varsa gerçekte 2 parametresi vardır. Derleyici static olmayan fonksiyona, bu fonksiyon hangi referans ile çağrılmışsa onu gizlice geçirmektedir.

İşte static olmayan fonksiyon, geçirilen bu gizli parametreli fonksiyon içerisinde açıkça `this` anahtar sözcüğü ile kullanılabilir. `this` anahtar sözcüğü gizlice geçirilen bu parametreyi temsil eder.

`this` anahtar sözcüğü hangi sınıfta kullanılırsa o sınıf türünden bir referans belirtir. `this` referansı hangi nesneyi göstermektedir? Static olmayan fonksiyon hangi referansla çağrılmışsa, o referansın gösterdiği nesneyi gösterir.

`this` anahtar sözcüğü yalnızca static olmayan fonksiyonlar ve propertylerde kullanılır. Static fonksiyonlarda ve propertylerde kullanılmaz. Çünkü static fonksiyonlar sınıf ismi ile çağrılır ve bunlara gizli bir referans geçirilmemektedir.

Static olmayan bir fonksiyon içerisinden, sınıfın örneğin `a` isimli static olmayan bir veri elemanına doğrudan `a` ifadesi ile erişmekle `this.a` ifadesi ile erişmek arasında hiçbir etkinlik farkı yoktur.

Sınıfın `Foo` isimli static olmayan fonksiyonun, `Bar` isimli static olmayan diğer bir fonksiyonu çağıracağını düşünelim. Bu işlem doğrudan `Bar(…)` biçiminde yapılabilir. Bu durumda aslında derleyici `Bar` fonksiyonuna gizlice `this` referansını geçirmektedir. Yani bu çağırmanın `this.Bar(…)` çağırmasından bir farkı yoktur.

`this` referansı read only bir referans kabul edilmektedir. Yani `this` referansını kullanabiliriz ama ona bir değer atayamayız.

`this` anahtar sözcüğüne neden gereksinim duyulmaktadır? `this` anahtar sözcüğü bazı tipik durumlarda faydalı bir biçimde kullanılabilir. Örneğin; sınıfın veri elemanları ile aynı isimli parametre değişkenlerinin ya da yerel değişkenlerin bulunduğu durumda, sınıfın veri elemanlarına erişebilmek için `this` anahtar sözcüğü kullanılabilir. Çünkü `this` anahtar sözcüğü ile

erişim uygulandığında artık noktanın sağındaki isim nitelikli arama kurallarına göre sınıf bildiriminde aranır. Örneğin:

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

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

Burada doğrudan kullanılan day parametre değişkeni, this.day ile kullandığımız ise sınıfın veri elemanıdır.

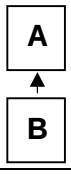
Programcıların çoğu hiçbir isim çakışması olmasa bile sınıfın veri elemanlarına yine de this anahtar sözcüğü ile erişmektedir. Bunun tek nedeni okunabilirliği arttırmaktır. Örneğin this.a gibi bir ifadeyi gören birisi a nın hemen sınıfın veri elemanı olduğunu anlar. Halbuki erişim doğrudan a diyerek yapılsaydı kodu inceleyen kişi bu sonucu hemen elde edemezdi. Bu sonucu çıkarabilmek için kişi, fonksiyonun tamamını incelemesi gerekirdi.

Yine programcıların çoğu static olmayan bir fonksiyonun içerisinde, static olmayan bir fonksiyonu this anahtar sözcüğü ile çağırmaktadır. Burada da amaç okunabilirliği arttırmaktır. Örneğin bir fonksiyonu yalnızca Foo diyerek çağırırsak kodu inceleyen kişi, fonksiyonu static de sanabilir static olmayan bir fonksiyon da sanabilir. Halbuki this.Foo şeklinde çağırırsak kodu inceleyen kişi kesinlikle fonksiyonun static olmadığını anlayacaktır.

TÜREMİŞ SINIFTAN TABAN SINIFA YAPILAN ATAMALAR

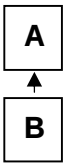
C# ta farklı türden iki sınıf referansı birbirine atanamaz. Örneğin A sınıfı türünden bir referans, B sınıfı türünden bir referansa doğrudan atanamaz. Fakat türemiş sınıf türünden bir referans, taban sınıf türünden bir referansa doğrudan atanabilir. Bu özel bir durumdur. Örneğin:

```
A x;

B y = new B();

x = y //geçerli türemişten tabana atama
```

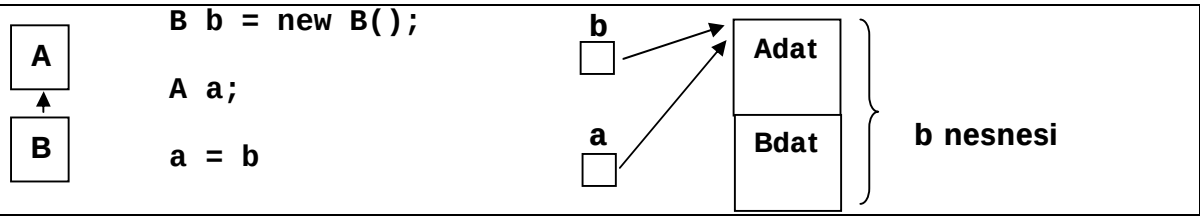


Fakat bunu tersi olan durum yani taban sınıf türünden referansın, türemiş sınıf türünden referansa atanması durumu geçerli değildir.

```
A a = new A();

B b;

b = a //error tabandan türemişe atama
```



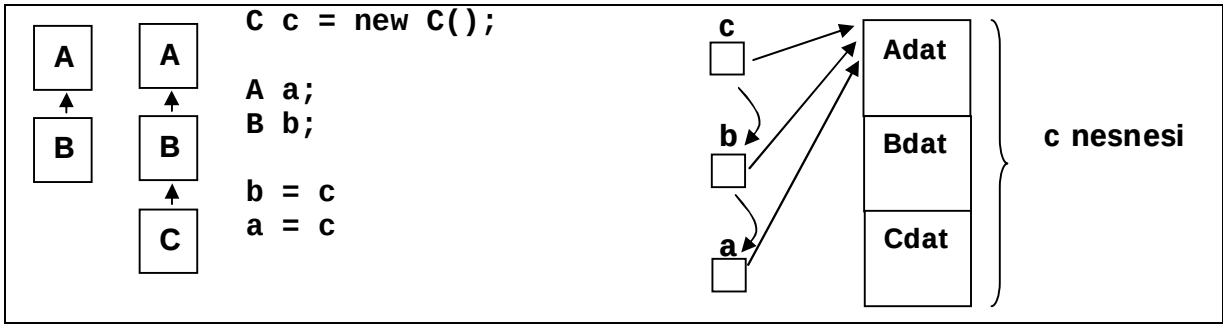
Türemiş sınıf nesne referansını, taban sınıf türünden bir referansa atadığımızda artık taban sınıf referansı, türemiş sınıf nesnesinin taban sınıf kısmını gösteriyor durumdadır. Yani biz bu taban sınıf referansı ile işlem yaptığımızda bu işlemlerden b nesnesinin a kısmı etkilenir.



Görüldüğü gibi burada a referansı, b nesnesinin a kısmını gösteriyor durumdadır.

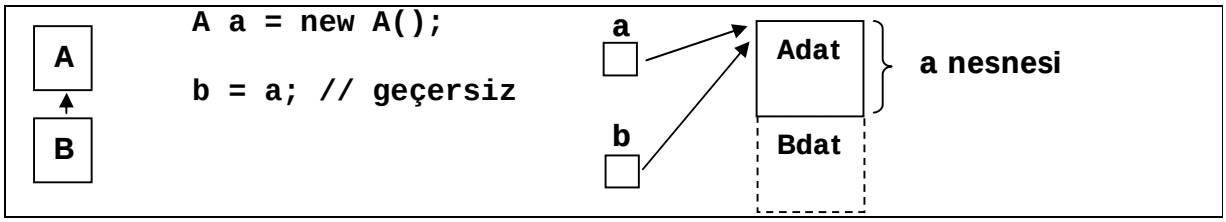
```
public static void Main()
{
     B b = new B();

     A a;

     b.ValB = 10;
     b.ValA = 20;
     Consol.WriteLine(a.ValA);

     a.ValA = 20;
     Consol.WriteLine(b.ValA);
}
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; }
     }
}
```

Bir dizi türetme söz konusu olsun. Bu durumda türemiş sınıfa ilişkin bir referans, onun tüm taban sınıf referanslarına doğrudan atanabilir. Örneğin:

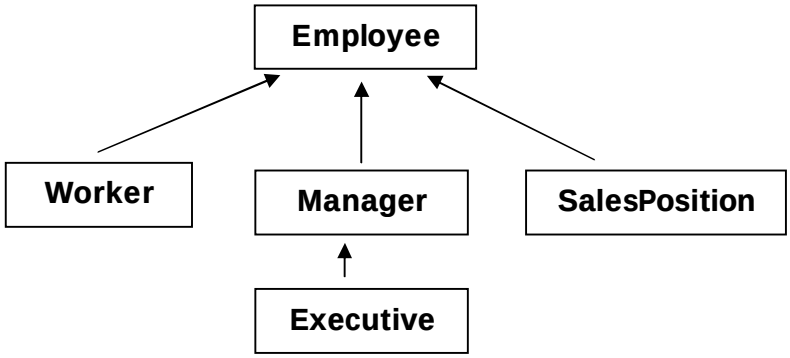


Burada `b = c` ataması ile birlikte `b` referansı, `c` nin `b` kısmını gösterir duruma gelir. Biz `b` referansı ile hem `B` sınıfının veri elemanlarını hem de `A` sınıfının veri elemanlarını değiştirebiliriz.

Türemişten tabana atama yapılabilmesinin gerekçesi, türemiş sınıfın taban sınıfı içermesindendir. Eğer tabandan türemişe atama yapılabilseydi, bu durumda türemiş sınıf referansı ile gerçekte var olmayan elemanlara erişme potansiyeli oluşurdu. Örneğin:

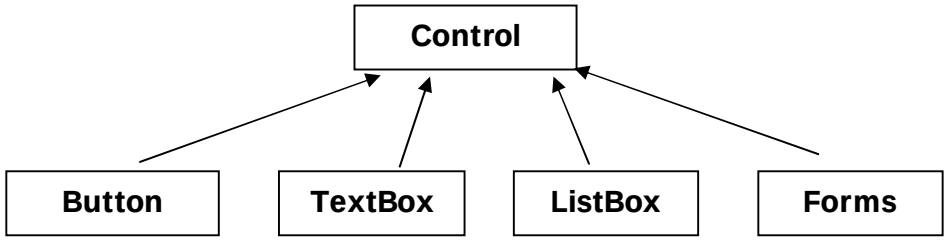


Fonksiyon çağırma işlemi de aslında parametrelerden, parametre değişkenlerine yapılan bir çeşit atama işlemidir. O halde bir fonksiyonun parametre değişkeni, taban sınıf türünden bir referans ise biz o fonksiyonu herhangi bir türemiş sınıf referansı ile çağırabiliriz. Örneğin:



```
Public static void DoSometing (Employee e)
{
     //…
}

     Manager m = new Manager();
     DoSometing(m);    à Manager nesnesinin Employee bölümü

     Worker w = new Worker();
     DoSometing(m);    à Worker nesnesinin Employee bölümü
```

Burada Employee sınıfı muhtemelen tüm çalışanların hepsinin sahip olduğu ortak elemanları içermektedir. DoSometing fonksiyonu bir Employee bilgisi almaktadır. Fakat Manager sınıfının Employee kısmı da olabilir, Worker sınıfının Employee kısmı da olabilir. Biz DoSometing fonksiyonunu Manager nesnesi(m) ile çağırdığımızda DoSometing, Manager nesnesinin Employee kısmı üzerinde işlem yapacaktır. Örneğin:




```
Public static void DoSometing (Control c)
{
     //…
}
```

Biz burada DoSometing fonksiyonun da Form nesnesini de , ListBox nesnesini de, TextBox nesnesini de, Button nesnesini de parametre olarak geçirebiliriz. DoSometing fonksiyonu, parametre olarak geçirilen nesneler farklı olsa bile onların Control kısmına ilişkin işlemler yapar. Windows işletim sisteminde ekranda bağımsız olarak kontrol edilebilen dikdörtgensel alanlara pencere denilmektedir. Pencerenin türü ne olursa olsun, onların çeşitli ortak özellikleri vardır. İşte Control sınıfı, tüm farklı pencerelerin pencerelik özelliği ile ilgili ortak özelliklerini içermektedir. Örneğin; her pencerenin bir zemin rengi söz konusudur. Zemin renginin değiştirilmesine izin veren eleman Control sınıfının elemanı olmalıdır.

System.Object SINIFI

System isim alanı içerisinde Object sınıfı .Net sınıf sistemi için çok önemli bir sınıftır. Bu sınıf çok kullanıldığı için object anahtar sözcüğü ile de temsil edilmektedir. Bu durumda bu sınıfı belirtmek için Using.System direktifinden Object ismini kullanabiliriz, System.Object ismini kullanabiliriz ya da object ismini kullanabiliriz. C# ta tıpkı Java da olduğu gibi her sınıf

doğrudan ya da dolaylı olarak object sınıfından türetilmiştir. Biz bir sınıf tanımlarken, hiç türetme yapmamış olsak bile derleyici yine de o sınıfın object sınıfından türetildiğini varsayar. C# ta object sınıfından türetilmemiş bir sınıf oluşturmak mümkün değildir. Örneğin:

```
class Sample
{
      //…
}
```

Burada hiçbir türetme syntaxı kullanılmamıştır. Fakat derleyici yine de Sample sınıfının object sınıfında türetildiğini varsayar. Tabi biz sınıfı açıkça object sınıfında türetebiliriz. Bu yasak değildir. Fakat gereksizdir. Örneğin:

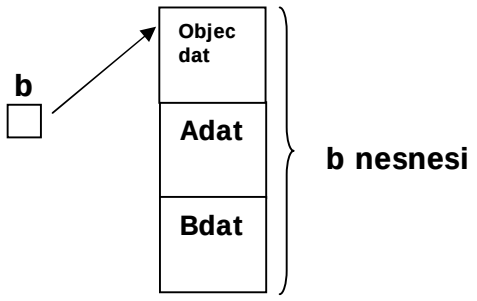
```
class Sample : object
{
      //…
}
```

Burada zaten :object yazılmasa da sanki yazılmış gibi işlem görecektir.

Her sınıfın ister istemez object sınıfından türetilmiş olması, C# ve Java gibi dillerde bazı kolaylıklar sağlamaktadır. C++ gibi dillerde böyle bir kolaylık öngörülmemiştir.

Her sınıf object sınıfından türetildiğine göre aslında her sınıf nesnesinin bir object kısmı da vardır. Fakat biz bugüne kadar çizimlerde bu durumu dikkate almamıştık. Bundan sonrada çizimlerde bu durum dikkate alınmayacaktır. Örneğin; B sınıfı A sınıfından türetilmiş olsun. Aslında A sınıfı da object sınıfından türetilmiştir. Gerçek çizimin şöyle olması gerekirdi:

```
B b = new B();
```



object sınıfının Equals, GetHashCode, GetType, ToString gibi static olmayan, Equals, ReferenceEquals isimli static olan fonksiyonları vardır. Bu fonksiyonlar daha sonra ele alınacaktır.

Madem ki her sınıf object sınıfından türetilmiştir, o halde her sınıf türünden referans object sınıfına atanabilir. Örneğin:

```
Sample s = new Sample();

object o;
```

```
o = s; //geçerli
```

TÜREMİŞ SINIFLARDA BAŞLANGIÇ FONKSİYONLARININ ÇAĞRILMASI

Başlangıç fonksiyonlarının temel amacı, sınıfın veri elemanlarına bir takım güvenli ilk değerler vermektir. Türemiş sınıf türünden bir nesne new operatörü ile yaratıldığında, türemiş sınıfın başlangıç fonksiyonu çağrılır. Türemiş sınıfın başlangıç fonksiyonu, türemiş sınıf veri elemanlarına ilk değerlerini verebilir. Fakat taban sınıfın private bölümüne erişemediğine göre taban sınıfın elemanlarına ilk değerleri veremeyecektir. Halbuki taban sınıf veri elemanlarına da ilk değerlerin verilebiliyor olması gerekir. İşte türemiş sınıfın başlangıç fonksiyonu, taban sınıfın başlangıç fonksiyonunu otomatik olarak çağırmaktadır. Böylece new operatörü ile türemiş sınıf türünden bir nesne yaratıldığında, türemiş sınıfın başlangıç fonksiyonu çağrılır. Türemiş sınıfın başlangıç fonksiyonu da, taban sınıfın başlangıç fonksiyonunu çağıracaktır.

Türemiş sınıf başlangıç fonksiyonunun hangi taban sınıf başlangıç fonksiyonunu çağıracağı :base syntaxı ile belirtilir. :base syntaxı türemiş sınıf başlangıç fonksiyonlarının kapanış parantezinden sonra yerleştirilir. Genel biçimi aşağıdaki gibidir:

```
:base ([ parametre listesi ])
```

Base syntaxı hiç belirtilmeyebilir. Bu durumda sanki :base() belirtmesi yapılmış gibi kabul edilir. Yani base syntaxı belirtilmemişse taban sınıfın default başlangıç fonksiyonu çağrılır.

:base syntaxının parametre listesinde, başlangıç fonksiyonu parametreleri kullanılabilir. Örneğin:

```
class A
{
     //…
}
class B:A
{
     public B()
     {
          //…
     }
     public B(int a) : base (a)
     {
          //…
     }
     public B(int A, int B) : base (a)
     {
          //…
     }
}
```

:base syntaxı yalnızca başlangıç fonksiyonlarında kullanılabilir. Herhangi bir fonksiyonda kullanılmaz. Başlangıç fonksiyonu çalıştırılma sırası önce, taban sınıf sonra türemiş sınıf biçimindedir. Taban sınıfın başlangıç fonksiyonun çağrılması, derleyicinin türemiş sınıf başlangıç fonksiyonunun ana bloğunun başına yerleştirdiği gizli bir kod yoluyla yapılmaktadır. Çağrılma sırasının önce taban sonra türemiş olmasının gerekçesi akış türemiş sınıf başlangıç fonksiyonuna geldiğinde burada taban sınıf veri elemanları ya da fonksiyonları kullanıldığında, taban sınıf veri elemanlarının ilk değerlerini almış olması gerekliliğidir.

Anahtar Notlar

Bir sınıfın doğrudan taban sınıfı, onun bir yukarısındaki taban sınıftır. Sınıfın diğer taban sınıflarına dolaylı taban sınıflar denilmektedir. Örneğin;

C in doğrudan taban sınıfı B, B nin A, A nın object sınıfıdır.

:base syntaxında doğrudan taban sınıf belirtilmektedir. Bir dizi türetme yapıldığında, başlangıç fonksiyonlarının çağrılma sırası en tepeden aşağıya doğrudur. Örneğin:

```
C c = new C();
```

Burada C sınıfının başlangıç fonksiyonu çağrılacaktır. Fakat başlangıç fonksiyonunun ana bloğunun başında akış B sınıfının başlangıç fonksiyonuna oradan da A sınıfının başlangıç fonksiyonuna gidecektir. O halde başlangıç fonksiyonu çağrılma sırası A,B,C şeklindedir.

Bir sınıf için bir başlangıç fonksiyonu yazdığımızda derleyici artık default başlangıç fonksiyonunu yazmaz. Bu durumda aşağıdaki örnekteki gibi error oluşur:

```
class A
{
     public A(int a)
     {
```

```
            //…
      }
      //…
}
class B:A
{
      public B() à error
      {
            //…
      }
      //…
}
```

Türemiş sınıf için hiçbir başlangıç fonksiyonu yazmamış olalım. Derleyicinin içi boş olarak yazacağı default başlangıç fonksiyonunu, taban sınıfın default başlangıç fonksiyonunu çağıracak şekilde yazılacaktır. Örneğin:

```
class A
{
      public A()
      {
            //…
      }
      //…
}
class B:A
{
      //…
}

B b = new B();
```

Burada A nın başlangıç fonksiyonu çalıştırılacaktır.

Yukarıdaki anlatımlarda object sınıfının başlangıç fonksiyonundan bahsedilmemiştir. Şüphesiz en önce object sınıfının başlangıç fonksiyonu çağrılmaktadır.

SINIF BİLDİRİMİ İÇERİSİNDE VERİ ELEMANLARINA İLK DEĞER VERİLMESİ

Sınıfın veri elemanlarına sınıf bildirimi içerisinde ilk değer verilebilir. Örneğin:

```
class Sample
{
      private int a = 10;
      private int b = 20;
      //...
}
```

Derleyici verilen bu ilk değerleri sırasıyla atama deyimlerine dönüştürerek sınıfın her başlangıç fonksiyonunun başına yerleştirir. Bu işlem, sınıfın pek çok başlangıç fonksiyonu olduğunda ve her başlangıç fonksiyonunda, veri elemanlarına aynı değerlerin atanması istendiğinde pratik olabilmektedir. Şüphesiz programcı ayrıca başlangıç fonksiyonunda bu değişkenlere değer atarsa, onun atadığı değerler kalacaktır. Atama deyimlerinin yerleştirilme sırası bildirimdeki sıraya göre yapılmaktadır.

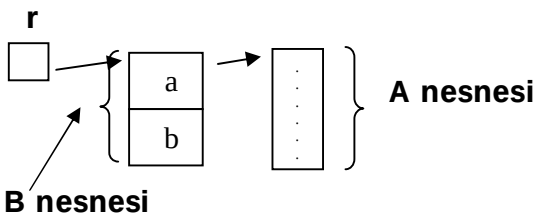
```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s = new Sample(100);

            Console.WriteLine(s.a);
            Console.WriteLine(s.b);
            Console.WriteLine(s.name);
        }
    }

    class Sample
    {
        public int a = 10;
        public int b = 20;
        public string name = "Savas";
        //...
        public Sample(int x)
        {
            a = x;
        }
    }
}
```

SINIFIN BAŞKA SINIF TÜRÜNDEN REFERANS VERİ ELEMANLARINA SAHİP OLMASI

Bir sınıf başka sınıf türünden veri elemanlarına sahip olabilir. Bu durumda elemana sahip sınıfın başlangıç fonksiyonu içerisinde, eleman için new operatörü ile tahsisat yapılmalıdır. Tabi bu tahsisat veri elemanına ilk değer verme şeklinde de yapılabilir. Örneğin:

```
class A
{
     //...
}

class B
{
```

```
     private A a;
     int b;
     public B()
     {
          a = new A();
          b = 10;
     }
     //...
}
B r = new B();
```



Şüphesiz aynı işlem şöyle de yapılabilir:

```
class A
{
     //...
}

class B
{
     private A a = new  A();      à burası farklı
     int b = 10;
     public B()
     {

     }
}
B r = new B();
```

NULL REFERANS KAVRAMI

Tıpkı diğer türlerde olduğu gibi referanslar da henüz kendilerine değer atanmadan herhangi bir biçimde kullanılamazlar. Örneğin:

```
Sample a;

a.Func();   à error
```

null anahtar sözcüğü boş bir adres belirtmektedir. Her türlü referansa doğrudan atanabilir. Bir referansa null atanmışsa, o referansa değer atanmıştır fakat referans hiçbir nesneyi göstermiyor durumdadır. İçerisinde null değer taşıyan bir referans kullanılırsa, derleme işlemi başarıyla

sonuçlanır fakat programın çalışma zamanı sırasında exception oluşur. Oluşan bu exception programın çökmesine neden olur. Örneğin:

```
Sample a;

a = null;
a.Func();   à Derleme aşamasını geçer fakat çalışma zamanı
sırasında exception oluşur.
```

Bir referansın içerisinde null olup olmadığı == ve != operatörleri ile test edilebilir:

```
if (a == null)
{
     //...
}
```

```
if (a != null)
{
     //...
}
```

null referans bir nesneyi çöp toplayıcı için seçilebilir duruma getirmek için kullanılabilir:

```
Sample a = new Sample();

//...

a = null;  // Nesne çöp toplayıcı tarafında seçilebilir durumda!
```

new operatörü tahsisatı yaptıktan sonra, sınıfın temel türden veri elemanlarına 0, referans türden veri elemanlarına ise null değerini yerleştirir. Temel türlerin içerisine null değeri atayamayız.

YAPILAR

Yapılar sınıflara çok benzer türlerdir. Bir yapı struct anahtar sözcüğü ile belirtilir. Örneğin:

```
struct Foo
{
     //...
}
```

Bugüne kadar sınıflar hakkında söylenilen her şey burada aksi belirtilmedi ise yapılar için de geçerlidir. Yapılar kategori olarak değer türlerine ilişkindir. Yani bir yapı türünden değişken tanımlandığında bu bir referans değildir. Değerlerin kendisini tutan parçalı bir nesnedir.

```
struct Test
{
     public int a;
     public int b;
```

```
    //...
}
Test t;
```

Biz şimdi doğrudan t.a ve t.b elemanlarını kullanabiliriz. t.a ve t.b bağımsız değişkenler gibi kullanılabilir fakat tabi kullanmadan önce değer atamış olmak yine bunlar içinde geçerlidir.

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Test t;
            t.a = 10;
            t.b = 20;
            Console.WriteLine(t.a);
            Console.WriteLine(t.b);
        }
    }

    struct Test
    {
        public int a;
        public int b;
        //...
    }
}
```

Yapı değişkenleri stack ta yaratılmaktadır. Dolayısıyla bunların yok edilmelerinin çöp toplayıcı ile bir ilgisi yoktur. Yapı nesneleri programın akışı nesnenin tanımladığı bloktan çıkıldığında otomatik olarak yok edilmektedir.

Yapılarında fonksiyonları, başlangıç fonksiyonları olabilir. Aynı türden iki yapı değişkeni birbirine atanabilir. Bu durumda yapının karşılıklı veri elemanları birbirine atanacaktır. Örneğin:

```
Test x;

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

Test y;
y = x;
```

Burada artık y nin a ve b parçaları içerisinde 10 ve 20 vardır.

Yapılar için de new operatörü ile tahsisat yapılabilir. Fakat bu durum sınıflar için tahsisat yapılmasından farklı bir anlam ifade eder. new

operatörünün operandı bir yapı ise new operatörü önce stack ta ilgili yapı türünden geçici bir değişken oluşturur. Sonra belirtilen başlangıç fonksiyonu çağrılır. new operatöründen ürün olarak stack ta yaratılmış geçici nesne elde edilir. Elde edilen bu ürün aynı türden bir yapı değişkenine atanabilir. Bu geçici yapı nesnesi, ilgili ifade bittiğinde otomatik derleyici tarafından yok edilir. Tabi atanan değişkende bu değerler kalacaktır. Örneğin:

```
struct Test
{
     private int a;
     private int b;

     public Test(int x, int y)
     {
          a = x;
          b = y;
     }
     public void Disp()
     {
          Console.WriteLine(a);
          Console.WriteLine(b);
     }
     //...
}
Test t;
T = new Test(10,20);
t.Disp();
```

Yapılar için hiçbir zaman programcı default başlangıç fonksiyonu yazamaz. Her zaman default başlangıç fonksiyonunu derleyici yazar. Derleyici yapılar için default başlangıç fonksiyonunu, programcı herhangi bir başlangıç fonksiyonu yazsa da yazmasa da her zaman yazmaktadır. (halbuki sınıflar için eğer biz hiç başlangıç fonksiyonu yazmamışsak, derleyici default başlangıç fonksiyonu yazmaktadır). Derleyicinin yazdığı default başlangıç fonksiyonu yapının tüm elemanlarını sıfırlamaktadır.

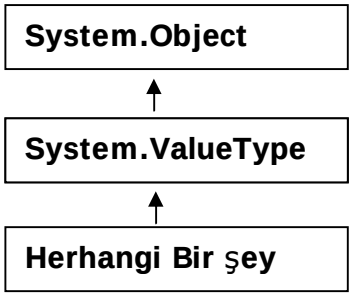
Anımsanacağı gibi sınıflar için new operatörü kullanıldığında, new operatörü heap ta tahsisatı yaptıktan sonra, sınıfın tüm temel türden elemanlarını sıfırlayıp, referans veri elemanlarına null yerleştiriyordu. Başlangıç fonksiyonu bu işlemden sonra çağrılıyordu. Halbuki yapılar için new uygulandığında, new operatörünün kendisi sıfırlama yapmaz. new operatörü doğrudan yapının başlangıç fonksiyonunu çağırır. Söz konusu başlangıç fonksiyonu default başlangıç fonksiyonu ise sıfırlama onun içerisinde yapılmaktadır. Örneğin:

```
Test t;
t = new Test();
```

Burada sıfırlama işini new operatörü değil default başlangıç fonksiyonu yapmaktadır. Halbuki sınıf söz konusu olsaydı new operatörü yapacaktı.

Eğer programcı yapı için parametreli bir başlangıç fonksiyonu yazacaksa başlangıç fonksiyonu bitene kadar kesinlikle yapının her elemanına değer atamış olmak zorundadır. Aksi halde derleme zamanında error oluşur. Böylece bir yapı için new operatörü ile tahsisat yapıldığında, yaratılan geçici nesnenin her elemanı kesinlikle değer almış olmak durumdadır. Aslında bu durum sınıflar için de böyledir. Fakat sınıflarda bu işlem, başlangıç fonksiyonu yoluyla değil new operatörü ile sağlanır.

Bir yapı hiçbir yapı yada sınıftan türetilemez. Yani yapılar türetmeye kapalıdır. Bu nedenle yapılarda ":" türetme syntaxı kullanılamaz. Fakat derleyici tüm yapıların `System` isim alanındaki `ValueType` sınıfında türetildiğini varsayar. `ValueType` sınıfı da `Object` sınıfında türetilmiştir.



Benzer biçimde bir yapı, bir yapıya yada sınıfa tabanlık ta yapamaz.

Bir yapı `protected` ve `protected internal` elemanlara sahip olamaz. Zaten yapılarda türetme yapılamadığına göre bunlar için `protected` ve `protected internal` elemanların da bir anlamı olmayacak.

TEMEL TÜRLERE İLİŞKİN YAPILAR

C# ta daha önce temel türler olarak gördüğümüz `int`, `long`, `double` gibi türler aslında birer yapı belirtmektedir. Örneğin; `int` türü aslında `System.Int32` denen yapıyı, `long` türü `System.Int64` denen yapıyı göstermektedir. Örneğin:

`int a` bildirimi ile `System.Int32` bildirimi tamamen aynıdır.

Madem ki temel türler aslında birer yapıdır o halde bu türlerde `System.ValueType` sınıfından türetilmiştir.

SINIF ELEMANI OLARAK YAPILARIN KULLANILMASI

Bir yapı eğer yerel bir değişkense stack ta yaratılır. Yoksa bir sınıfın elemanı olduğu zaman kendisi de sınıf nesnesi ile birlikte, sınıf nesnesinin bir parçası olarak heapta bulunacaktır. Örneğin:

```
struct Foo
{
     public int a;
     public int b;
```

```
     //...
}
class Bar
{
     private int x;
     private Foo y;
     //...
}

//...

Foo foo = new Foo() à stacta yaratım yapılıyor.
Bar bar;            à  “        “        “

bar = new Bar();    à heapta yaratım
```

Yukarıdakinin tersi bir durum yani bir yapının elemanının, bir sınıf referansı olması durumu da geçerli ve normaldir. Bu durumda eğer yapı nesnesi yerel bir biçimde yaratılmışsa, yaratım stack ta yapılır. Fakat eleman olan sınıf referansı heapta bir nesneyi gösterecektir. Örneğin:

```
class Foo
{
     private int a;
     private int b;
     //...
}
struct Bar
{
     private int x;
     private Foo y;
     public Bar(int x)
     {
          this.x = x;
          y = new Foo();
     //..
     }
     //...
}
{
     Bar bar = new Bar(10);
     //...
}
```

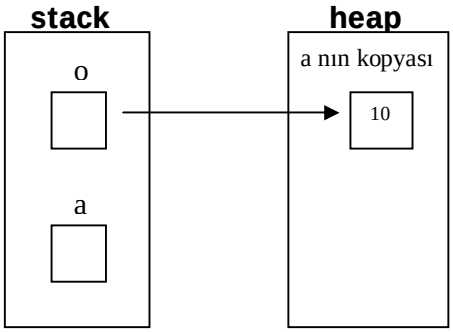
Burada programın akışı, yapı nesnesinin tanımlandığı bloğu bitirdiğinde heap ta tahsis edilmiş olan sınıf nesnesi de çöp toplayıcı tarafından seçilir duruma gelir.

KUTULAMA DÖNÜŞTÜRMESİ(Boxing Corversion)

Bir referans hiçbir zaman stack taki bir nesneyi göstermez. Her zaman heaptaki bir nesneyi göstermek zorundadır. Bir yapı, onun taban sınıfı olan

ValueType yada Object türünden sınıflara doğrudan atanabilir. Fakat bu durumda referans stack taki yapıyı gösteremeyeceğine göre ne olacaktır? İşte ne zaman bir yapı değişkeni ValueType yada object referanslarına atansa, derleyici otomatik olarak o yapı nesnesinin heapta bir kopyasını oluşturur ve bu referans heaptaki bu nesneyi gösterir durumda olur. Otomatik yapılan bu işleme kutulama dönüştürmesi denir.

```
{
     int a = 10;
     object o;

     o = a;
     //...
}
```

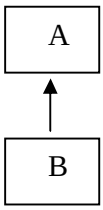


Burada o = a işlemi ile derleyici a nın heapta bir kopyasını oluşturur. "o" artık heaptaki kopyayı gösterir durumdadır. Bu işlemden sonra artık stack taki a ile heaptaki kopyası bağımsız iki ayrı nesnedir. Bu işlemden sonra artık biz örneğin stack taki a yı değiştirsek bu işlemden heaptaki a etkilenmez, heaptaki a yı değiştirsek stack taki a etkilenmez. Stack taki a, akış bloktan çıktıktan sonra otomatik olarak, heaptaki a ise çöp toplayıcı tarafından yok edilecektir. Şüphesiz o referansı heaptaki a nın object kısmını göstermektedir.

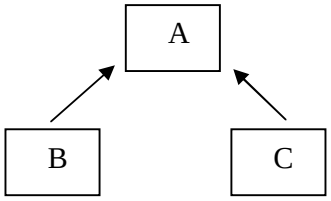
AŞAĞIYA DOĞRU YAPILAN DÖNÜŞTÜRMELER

Anımsanacağı gibi türemiş sınıf referansından taban sınıf referansına doğrudan atama yani dönüştürme vardır. Fakat taban sınıf referansını doğrudan türemiş sınıf referansına atayamayız. Bu işlemi ancak tür dönüştürme operatörü ile yapabiliriz. Örneğin:

```
B b = new B();
A a;

a = b; //geçerli

B x;
x = a; //error

x = (B)a; //geçerli
```



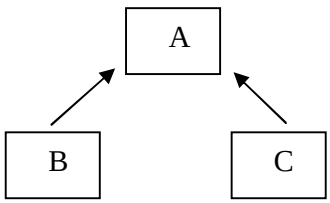
Şüphesiz doğrudan yapılan her atama aynı zamanda tür dönüştürme operatörü ile de yapılabilir. Aralarında türetme ilişkisi olmayan iki sınıf arasında tür dönüştürme operatörü ile de dönüştürme yapılamaz. Örneğin:



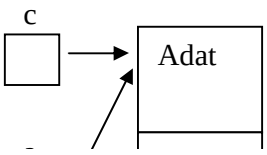

```
B b = new B();

C c;

C = (B)c;  à error
```

Burada `B` ile `C` arasında bir türetme ilişkisi yoktur. O halde `B` ile `C` arasında biz tür dönüştürme operatörü ile de dönüştürme yapamayız.

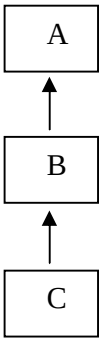
Tabanda türemişe dönüştürme(down_cast) işlemi derleyici tarafından her zaman kabul edilir. Ancak bu işlem ayrıca programın çalışma zamanı sırasında da CLR tarafından kontrol edilmektedir. CLR akış dönüştürmenin yapıldığı bölüme geldiğinde dönüştürülecek referansın gösterdiği yerdeki nesnenin içerisinde, dönüştürülecek türe ilişkin bir bölümün olup olmadığına bakar. Eğer böyle bir bölüm varsa dönüştürme haklıdır ve sorun çıkmaz. Eğer böyle bir bölüm yoksa, dönüştürme haksızdır ve çalışma zamanı sırasında `InvalidCastException` ortaya çıkmaktadır. Bu da programın çökmesine yol açar. Örneğin:



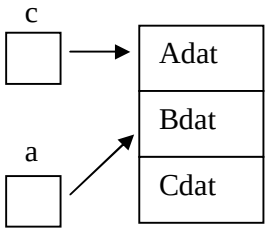

```
C c = new C();
A a;
a = c; //geçerli
B b;
b = (B)a; //derleme aşamasını geçer fakat çalışma zamanında
exception oluşur.
```



Yukarıdaki kod başarılı bir biçimde derlenecektir. Fakat akış `b = (B)a` bölümüne geldiğinde `a` referansının gösterdiği yerdeki nesnede `B` dataları olmadığı için dönüştürme haksızdır. CLR exception oluşturacaktır. Aslında burada programcı araya taban bir sınıf sokarak `C` den `B` ye dönüştürme yapmak istemiştir. Derleyici kandıra bilmiştir ama CLR durumu tespit etmiştir. Örneğin:




```
C c = new C();
A a = c; //geçerli
B b;
b = (B)a; //haklı dönüştürme
```



Burada `b = (B)a` dönüştürmesi haklı bir dönüştürmedir. Çünkü `a` referansının gösterdiği yerde `B` türünden datalar vardır.

Görüldüğü gibi bir referansı taban sınıf türünde bir referansa atayıp onu yeniden türemiş sınıf türüne dönüştürürsek bu dönüştürme her zaman haklı bir dönüştürme olur.

Görüldüğü gibi yukarıya doğru dönüştürme her zaman geçerlidir. Fakat aşağıya doğru dönüştürme ancak tür dönüştürme operatörü ile yapılabilir ve ayrıca programın çalışma zamanı sırasında CLR tarafından da denetim uygulanmaktadır.

Anahtar Notlar

Bazı programcılar sınıfın veri elemanlarının `m_` ***ya da*** `d_` ***gibi öneklerle başlatarak isimlendirmektedir..... Böylece çakışma durumunda*** `this` ***anahtar sözcüğünü kullanmaya gerek kalmaz. Aynı zamanda bu öneklerle başlayan isimlerin veri elemanı olduğu anlaşılabilir. Kursumuzda bazı örneklerde*** `m_` ***öneki de kullanılacaktır.***

KUTUYU AÇMA DÖNÜŞTÜRMESİ(UNBOXING CORVERSATION)

Kutulama dönüştürmesi ile heap'e aktarılan bir yapı, sonra yeniden aşağıya doğru dönüştürme yapılarak stack'a çekilebilir. Tabi aşağıya doğru dönüştürme için tür dönüştürme operatörünün kullanılması gerekir. `System.ValueType` **ya da** `System.Object` **türünden bir referansı, o referansın ilişkin olduğu yapıya dönüştürmeye çalıştığımızda, stackta dönüştürülecek yapı türünden bir geçici yapı nesnesi oluşturulur. Heapte bulunan bu yapı nesnesi, stacktaki bu geçici nesneye kopyalanır. İlgili ifade bittiğinde stacktaki bu geçici nesne yok edilir. Bu işleme kutuyu açma dönüştürmesi denilmektedir. Örneğin:**

```
int a = 100;
object o = a;
//...
int x;
x = (int)o;    à kutuyu açma dönüştürmesi
```

Burada `x = (int)o` **işlemi ile önce stackta geçici bir değişken yaratılır. Sonra o referansının gördüğü yerdeki nesne, bu geçici nesneye kopyalanır.** `x = (int)o` **işlemi ile içerisinde** `100` **değeri olan bir geçici değişken elde edilir.**

```
public static void Main()
{
     int a = 100;
     object o = a;

     int x = (int) o;

     Console.WriteLine(x);
}
```

Kutuyu açma dönüştürmesinde bilinçli dönüştürme, eğer farklı türden bir yapıya uygulanırsa (temel yapı türleri de dahil olmak üzere) programın çalışma zamanı sırasında exception oluşur. Örneğin:

```
public static void Main()
{
     int a = 100;
     object o = a;

     long  x = (long) o; à derleme aşamasını geçer ,çalışma sırasında exception oluşur.
```

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

Şüphesiz kutulama dönüştürmesi sabitler ile de gerçekleştirilir. Örneğin:

```
object o = 123;
```

Burada yine heapte int türden bir değişken yaratılır, 123 değeri bu değişkenin içerisine atanır, o referansı da artık heapteki bu nesneyi gösteriyor durumda olur. Bu değeri kutuyu açma dönüştürmesi ile geri alabiliriz. Örneğin:

```
object o = 123;

int x;

x = (int) o;
```

Örneğin object türünden bir diziye kutulama dönüştürmesi yoluyla çeşitli değerler atayabiliriz. Sonra foreach döngüsü ile bunları geri alabiliriz. Anımsanacağı gibi foreach döngüsü, her yinelemede dizilimin bir elemanını tür dönüştürmesi yoluyla döngü değişkenine atamaktadır:

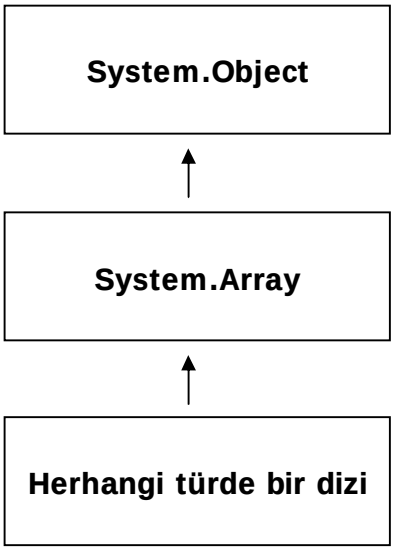
```
object [] obj = new object [] {10,20,30,40,50};

foreach (int x in obj)
{
    Console.WriteLine(x);
}
```

collection SINIF KAVRAMI

Amacı birden başka sınıf nesnelerini tutmak olan sınıflara collection sınıf denilmektedir. Örneğin; dizilerde bir bakıma collection sınıflardır. .Net içerisinde pek çok collection sınıf vardır. Bu sınıflar System.Collection isim alanında bulunmaktadır.

DİZİLERİN TÜRETME DURUMLARI

Aslında diziler de birer sınıf olarak değerlendirilebilir. .Nette bütün dizilerin türü ne olursa olsun System.Array isimli sınıftan türetildiği varsayılmaktadır. System.Array sınıfı da System.Object sınıfından türetilmiştir.



Yani biz herhangi bir dizi referansı ile System.Array sınıfının yada Object sınıfının elemanlarına erişebiliriz. Aslında dizinin uzunluğunu elde etmekte kullandığımız Lenght isimli property ve dizi elemanlarına ulaşmakta kullandığımız indeksleyici System.Array sınıfının elemanlarıdır.

Madem ki tüm diziler dolaylı olarak object sınıfından türetilmiştir o halde biz bir dizi referansını object türünden bir referansa atayıp geri dönüştürebiliriz. Örneğin:

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

object o = a;

int b[] = (int []) o;

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

ArrayList Collection SINIFI

Aslında dizilerde birer collection sınıfıdır. Fakat bazı gereksinimleri karşılamakta diziler yetersiz kalmaktadır. Örneğin; bir dizi açmış olalım ve bir kaynaktan gelen bilgileri diziye ekleyelim. Dizi dolunca ne olacaktır? C# ta diziler büyütülemez. İşte bu durumda tek seçenek daha büyük yeni bir dizi tahsis etmek, eski dizideki elemanları yeni diziye kopyalamak ve yeni diziden işleme devam etmektir. Bu yeni dizi de taşarsa bu işlemleri yinelemek gerekir. İşin başında çok büyük dizi tahsis etmek belleğin verimsiz kullanılmasına yol açar. İşte C# ta bu tür durumlarla çok sık karşılaşılmaktadır. Bu tipik duruma dinamik olarak büyüyen dizi denilmektedir.

Anahtar Notlar

Bir diziyi bir diziye kopyalama basit bir for döngüsü ile yapılabilir. Ya da bunun için System.Array sınıfının, static Copy fonksiyonları ya da static olmayan CopyTo kullanılabilir.

```
public static void Copy (Array SourcaArray, Array destinationArray, int length)
```

Fonksiyonun birinci parametresi kaynak diziyi, ikinci parametresi hedef diziyi ve üçüncü parametresi dizinin kopyalanacak eleman sayısını almaktadır. Örneğin:

```
{
int [] a = new int [5] {1,2,3,4,5};
int [] b = new int [10];

Array.Copy(a,b,5);

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

Aynı işlem static olmayan Copyto fonksiyonu ile de yapılabilir.

```
public void CopyTo(Array array, int index)
```

Fonksiyonun birinci parametresi hedef diziyi, ikinci parametresi kopyalamanın hedef dizinin hangi indeksinden itibaren yapılacağını belirtir. Yani kaynak dizinin tüm elemanları hedef dizinin belli bir indeksinden itibaren kopyalanabilir. Örneğin:

```
{
int [] a = new int [5] {1,2,3,4,5};
int [] b = new int [10];

a.CopyTo(b,0);

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

ArrayList sınıfı dinamik büyütülen bir diziyi temsil etmektedir. Sınıfın içerisinde muhtemelen private bölümünde object türünden bir dizi vardır. Sınıf kendi içerisinde ayrıca count ve capacity biçiminde iki bilgi de tutmaktadır. Bu bilgiler Count ve Capacity isimli property elemanlarla ulaşılabilir. Capacity sınıfın içindeki object dizisi için kaç elemanlık yer tahsis edildiğini gösterir. Count ise o anda dizinin kaç elemanının dolu olduğunu belirtmektedir. Sınıfın object parametreli bir Add fonksiyonu vardır. Bu fonksiyon alınan, değeri Count ile belirtilen indekse yazar ve Count değerini bir arttırır. Count değeri Capacity e geldiğinde Add fonksiyonu kendi

içerisinde eski Capacity değerinin iki katı kadar yeni bir dizi tahsis eder ve işlemlere bu yeni diziden devam eder. ArrayList sınıfının indeksleyicisi olduğu için istenilen eleman köşeli parantez operatörü ile geri alınabilir.

ArrayList sınıfı elemanları object gibi tutar ve yine bize objectmiş gibi verir. Örneğin:

```
{
     ArrayList al = new ArrayList();

     for (int i = 0; i < 100; ++i)
          al.Add(i);

     for (int i = 0; i < al.Count; ++i)
     {
          int x = (int) al[i];

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

Add fonksiyonunun parametrik yapısı şöyledir:

```
public virtual int Add(object value)
```

Fonksiyon parametre olarak yerleştirilecek elemanı alır, geri dönüş değeri olarak elemanın yerleştirildiği indeksi verir. Köşeli parantez ile elde edilen değer object türündendir.

Görüldüğü gibi biz ArrayList içerisinde int türünden değerleri saklamak istediğimizde aslında ArrayList kutulama dönüştürmesi ile onu object türüne dönüştürmekte ve onu object olarak saklamaktadır.

Foreach deyimi aslında Ienumerable arayüzünü destekleyen her türlü sınıfta kullanılabilir. Diziler de ArrayList sınıfı da Ienumerable arayüzünü desteklemektedir. O halde foreach deyimi ArrayList sınıfı ile kullanılabilir. Örneğin:

```
{
     ArrayList al = new ArrayList();

     for (int i = 0; i < 10; ++i)
          Al.Add(i);

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

ArrayList sınıfının Insert isimli fonksiyonu, belirli bir değeri o değer ArrayList içerisindeki dizide belirli bir indekste olacak şekilde ekleme yapar.

```
public virtual void Insert (int index, Object value)
```

Fonksiyonun birinci parametresi Insert işleminin yapılacağı indexi ikinci parametresi Insert yapılacak değeri belirtmektedir. Şüphesiz bu fonksiyonda ArrayList içerisindeki dizide bir kaydırmaya yol açmaktadır.

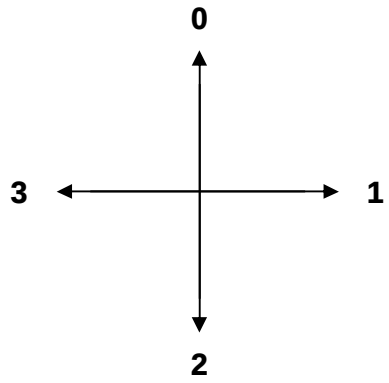
```
using System;
using System.Collections;

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, 100);

            foreach(int x in al)
                Console.WriteLine(x);

        }
    }
}
```

ArrayList.RemoveAt fonksiyonu belirli bir indeksteki değeri silmek için kullanılır. Fonksiyon parametresi ile belirtilen indeksteki değeri silmektedir. Şüphesiz silme işlemi sırasında bir sıkıştırma işlemi yapılmaktadır. (ArrayList sınıfında eleman sildiğimiz zaman Capacity değeri hiçbir zaman küçültülmemektedir.)
ArrayList sınıfının Reverse fonksiyonu elemanları ters yüz eder. Sort fonksiyonları sıraya dizme işlemi yapmaktadır.

enum TÜRÜ VE enum SABİTLERİ

Bazen bir fonksiyonun parametresi kısıtlı sayıda seçenekten hangisinin söz konusu olduğunu belirlemek için kullanılır. Örneğin; bir şekli hareket ettiren MoveShape isimli fonksiyon şeklin hangi yönde hareket ettirileceğini belirlemek için bir yön parametresi almaktadır.

```
void MoveShape (int direction)
{
     //...
}
```



Burada parametre olan yön bilgisi `int` türü ile temsil edilmiş olmasına karşın kısıtlı sayıda değer almaktadır.

Şüphesiz fonksiyonun parametresini `string` yapmak daha okunabilir bir durum oluşturur. Fakat tamsayılarla çalışmak her zaman daha hızlıdır. Örneğin:

```
MoveShape(3);       à Daha hızlı fakat daha az okunabilir.
MoveShape("Left");  à Daha okunabilir fakat daha yavaş.
```

Haftanın günleri, aylar, meyveler, klavyedeki tuşlar gibi pek çok bilgi aslında `int` türden bir sayı ile temsil edilebilir. Fakat hangi sayının hangi kavramı belirttiği, kodu inceleyen kişi tarafından doğal olarak bilinmediğinden dolayı `int` gibi bir tamsayı ile temsil işlemi okunabilirliği azalmaktadır.

Bir `enum` bildiriminin genel biçimi şöyledir:

```
enum <isim>
{
     S1, S2, S3...
}
```

Örneğin:

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

Bir `enum` sabiti, `enum` ismi ve "." operatörü ile kullanılır.

Örneğin:

```
Günler.Pazar;
```

Örneğin:

```
enum Directions
{
     Up, Down, Left, Right
}

Direction.Up
```

Her enum sabiti aslında bir sayı belirtmektedir. İlk enum sabiti 0 olmak üzere her enum sabiti öncekinden bir fazla değer belirtmektedir. Yani Direction.Up à 0 Direction.Down à1 Direction.Left à 2 Direction.Right à 3

Her enum ayrı bir tür belirtir ve enum sabitleri ilişkin olduğu enum türündendir. Örneğin:

Direction.Up ifadesi Direction isimli bir enum türündendir. Günler.Sali ifadesi Günler isimli enum türündendir.

Bir enum türünden değişkin tanımlanabilir. Örneğin:

```
Direction d;
Günler g;
d = Direction.Up
g = Günler.Sali
```

enum türleri kategori olarak değer türlerine ilişkindir. Yani tıpkı int türünde olduğu gibi enum türünden bir değişkenin içerisine bir adres bilgisi değil değerin kendisi yerleştirilir. Her ne kadar enum türü aslında bir tamsayı türü ise de temel türlere ilişkin bir değer doğrudan bir enum türüne atanamaz. Örneğin:

```
Direction d;
d = 2; à error
```

Aynı türden iki enum türü birbirine atanabilir. Örneğin:

```
Direction d;
d = Direction.Up; à geçerli
Direction x;
x = d; à geçerli
```

İşte bir bilginin hem sayısal olmasını hem de okunabilir bir biçimde ifade edilmesini istiyorsak enum kullanmalıyız. Örneğin:

```
void MoveShape(Direction d)
{
     //...
}
```

```
//...
MoveShape(Direction.Left);
```

Bir enum sabitine eşittir ile bir değer verilebilir. Bu durumda onu izleyen enum sabiti bir öncekine verilen değerin bir fazlası olur. Ayrıca iki farklı enum sabiti aynı sayısal değerde de olabilir. Örneğin:

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

Burada XX = 0 , YY = 10, ZZ = 11, KK = -1, MM = 0 değerdedir. Enum sabitine verilen ilk değer sabit ifadesi olmalıdır. Önceki enum sabitleri kullanılabilir. Örneğin:

```
enum Test
{
     XX,
     YY = XX + 2
}
```

switch parantezinin içerisinde enum türünden bir ifade olabilir. Bu durumda case ifadeleri enum türünden olmalıdır.

```
Directions d = GetSomeDirection();

Switch (d)
{
     case Directions.Up
     //...
     break;
     case Directions.Right
     //...
     break;
     //...
}

static Directions GetSomeDirection()
{
     return Directions.Right;
}
enum Direcitons
{
     Up,Down,Left,Right
}
```

Aslında çoğu kez bir enum sabitinin hangi değerde olduğu programcıyı ilgilendirmemektedir. Programcı enum sabitinin isminden hareketle ne olacağını tahmin edebilmektedir. Her enum türünün ilişkin olduğu kardeş bir

tamsayı türü vardır. Kardeş tamsayı türü ":" syntaxı ile belirtilir. Eğer bu syntax kullanılmazsa kardeş tamsayı türü int anlaşılır. Örneğin:

```
enum Foo
{
     XX,
     YY
}

enum Bar : long
{
     MM,
     ZZ
}
```

enum türlerinden tamsayı türlerine, tamsayı türlerinden enum türlerine doğrudan atama yapılamaz. Fakat tür dönüştürme operatörü ile atama yapılabilir. Tür dönüştürme operatörü kullanıldığında oluşacak durum, enum türünün ilişkin olduğu kardeş tamsayı türünün, ilgili tamsayı türüne dönüştürülmesi sırasında oluşan durumla aynıdır. Dönüştürme sonucunda enum türünün içerisindeki sayısal değer elde edilir. Örneğin:

```
static Directions GetSomeDirection()
{
     Directions d = Directions.Right;
     long i;

     i = d; // error

     i = (long)d;  à geçerli i de 3 var
}
```

(long)d dönüştürmesinde Directions isimli enum türünün ilişkin olduğu tamsayı türü int ise, int türünden long türüne dönüştürme kuralları uygulanır. Görüldüğü gibi d ifadesi Directions türündendir. Fakat (long)d ifadesi long türdendir.

Görüldüğü gibi bir enum türünden değerin belirttiği tamsayı değeri istenildiği zaman tür dönüştürme operatörü ile elde edilebilir.

Enum türünden bir değer Console sınıfının Write ve WriteLine fonksiyonları ile yazdırılabilir. Fakat bu durunda enum değerine ilişkin sayısal değil yazısal bilgi ekrana basılır.Örneğin:

```
Directions d = Directions.right;

Console.WriteLine(d);          àRight
Console.WriteLine((int)d);     à3

Enum Directions (class dışına yaz)
{
```

```
    Up,
    Down,
    Right,
    Left
}
```

Bir tamsayı türünden değer de, tür dönüştürme operatörü ile bir enum türüne dönüştürülebilir. Bu durumda önce dönüştürülecek değer, enum türünün ilişkin olduğu kardeş tamsayı türüne dönüştürülür. Sonra dönüştürülmüş olan bu değer, ilgili enum türüne dönüştürülür. Ayrıca enum türünden bir değişkenin kendi türünden bir enum sabitinin değerini tutması zorunu değildir. Örneğin:

```
long x = 1234567;
Directions d;

d = x;              à error
d = (Directions)x;  à geçerli
```

Burada (Directions)x dönüştürmesi sırasında önce long türden int türe dönüştürme kuralı uygulanır. Yani herhangi bir kayıp oluşmayacaktır. Daha sonra int türüne dönüştürülen bu değer Directions türüne dönüştürülecektir. Şimdi artık d nin içerisinde sayısal olarak 1234567 değeri vardır. Şüphesiz burada biz d değişkeninin içerisindeki değeri Console sınıfının Write ve WriteLine fonksiyonları ile yazdırmaya çalışsak ekrana bir yazı basılamayacağına göre sayı basılacaktır.

.Net sınıf kütüphanesinde enum türü çok sık kullanılır. Örneğin DateTime isimli yapının DayOfWeek isimli property elemanı DayOfWeek isimli bir enum türündendir. DayOfWeek isimli enum türünün elemanları da haftanın günlerini barındırmaktadır. Bugünün hangi gün olduğu şu şekilde yazdırılabilir:

```
DayOfWeek dow = DateTime.Today.DatOfWeek;
Console.WriteLine(dow);
```

MSDN dokümanlarında DayOfWeek isimli enum türünün Sunday isimli elemanından Saturday elemanına kadar olan değerlerin 0 ile 6 arasında olduğu belirtilmiştir. Eğer günleri Türkçe yazdırmak istersek switch kullanabiliriz.

```
switch (DateTime.Today.DayOfWeek)
{
    case DayOfWeek.Sunday:
        C.W.("Pazar");
        break;
    //...
}
```

Aynı işlem şöyle de yapılabilirdi:

```
string [] days = {“Pazar”, “Pazartesi”, “Salı”, “Çarşamba”, “Perşembe”, “Cuma”, “Cumartesi”};

int index = (int) dateTime.Today.DayOfWeek;

Console.WriteLine(days[index]);
```

Anahtar Notlar

Bir yapı ya da sınıfın bir elemanının ismi, o elemana ilişkin tür ismi ile aynı olabilir. Bu durum karışıklığa yol açmaz. Şüphesiz sınıf ya da yapı içerisinde bu isim kullanılırda isim arama kuralına göre eleman olan isim anlaşılır. Dışarıda kullanılırsa zaten isim niteliklendirileceği için sorun kalmaz. Örneğin:

```
enum Test
{
     XX,YY,ZZ
}

class Sample
{
     public Test Test;
     //...
}
//...
Sample s = new Sample();
Test t = s.Test; à geçerli
```

System.Windows.Forms isim alanı içerisindeki MessageBox isimli sınıfın Show isimli fonksiyonları mesaj penceresi çıkartmak için kullanılır. Bir mesaj penceresinin dört özelliği vardır: başlık yazısı, pencere içerisindeki mesaj yazısı, tuş takımı ve ikon görüntüsü. Show fonksiyonlarından biri şöyledir:

```
public static DialogResult Show (string text, string caption, MessageBoxButtons buttons)
```

Fonksiyonun birinci parametresi pencere içerisine yazılacak yazıyı, ikinci parametresi pencere başlık yazısını, üçüncü parametresi tuş takımı belirtir. MessgeBoxButtons isimli enum türünün elemanları tuş takımı seçeneklerini belirtir. Show fonksiyonunun geri dönüş değeri DialogResult türünden bir enum'dır. DialogResult isimli enum türünün elemanları dialog penceresinin hangi tuşla kapatıldığını belirtir. 4 parametreli Show fonksiyonunun dördüncü ve son parametresi görüntülenecek ikonu belirlemede kullanılır.

```
public static DialogResult Show (string text, string caption, MessageBoxButtons buttons, MessageBoxIcon icon)
```

```
using System;
using System.Collections;
```

```
using System.Windows.Forms; //Windows.Forms dll referans

namespace CSD
{
    class App
    {
        public static void Main()
        {
            DialogResult dr;
            dr = MessageBox.Show("Emin misin?", "Uyarı",
MessageBoxButtons.YesNoCancel);
            if (dr == DialogResult.Yes)
                MessageBox.Show("Yes");
            else
                MessageBox.Show("No");
        }
    }
}
```

enum TÜRLERİ ÜZERİNE İŞLEMLER

Aynı türden iki enum ==, !=, <,>, >=,<= operatörleri ile karşılaştırılabilir. Bu durumda aslında o enum değerlerinin belirttiği tamsayı karşılaştırılmaktadır.

Bir enum değeri ile o enum türünün ilişkin olduğu kardeş tamsayı türüne ilişkin yada kardeş tamsayı türüne doğrudan dönüştürülebilen bir değer toplanabilir. İşlem sonucunda ilgili enum türünden bir değer elde edilir. e E isimli bir enum türünden, u da bu enum türünün ilişkin olduğu U türünden ya da U türüne doğrudan dönüştürülebilen bir türden olsun. Bu durumsa e + u ya da u + e işlemi geçerlidir. işlem sonucunda (E)((U)e + (U)u) sonucu elde edilir. Örneğin:

```
enum Test
{
     XX, YY, ZZ
}
//...
Test x = Test.XX;
Test y;
y = x + 1;
```

Burada y nin içerisinde tamsayı olarak 2 değeri bulunmaktadır.

Aynı türden iki enum – operatörü ile çıkartılabilir. Elde edilen değer enum türünün ilişkin olduğu kardeş tamsayı türündendir. Yani e1 ve e2 aynı türden iki enum değeri olsun. U da bu enum türünün ilişkin olduğu tamsayı türü olsun. (U)e1 – (U)e2 ile eşdeğerdir. Örneğin:

```
enum Test
{
```

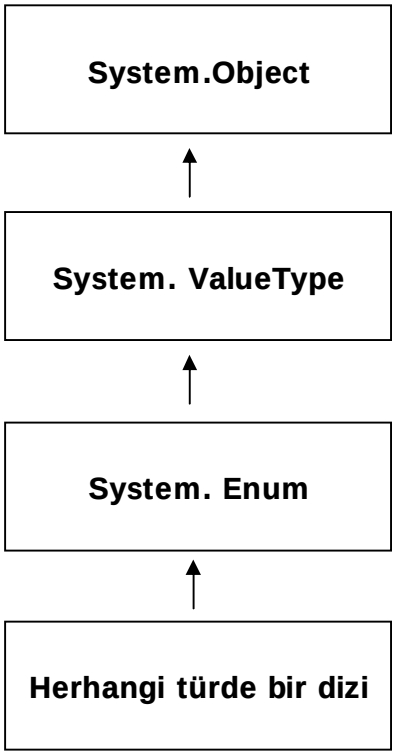
```
    XX, YY, ZZ
}
//...
Test x = Test.YY;
Test y = Test.XX;
int result;
result = x-y;
```

result değişkeninde 1 değeri bulunacaktır.

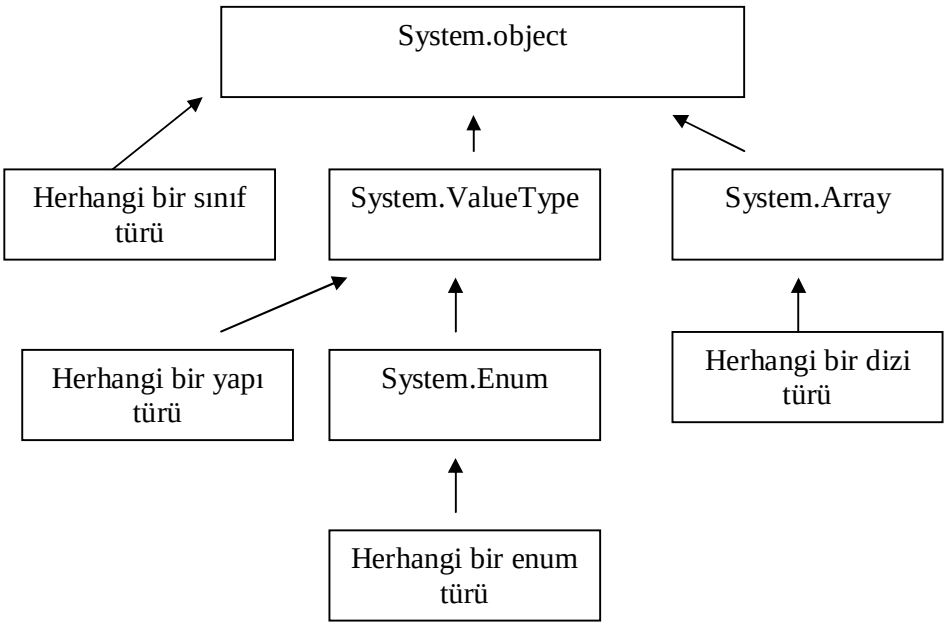
Bir tamsayı değerinden bir enum değeri çıkarılamaz. Fakat bir enum değerinden o enum türünün ilişkin olduğu tamsayı türünden yada o tamsayı türüne doğrudan dönüştürülebilen bir değer çıkartılabilir. Elde edilen sonuç ilgili enum türünden olur. Yani e – u işlemi (E)((U)e – (U)u) işlemi ile eşdeğerdir.

Bir enum türünden değişken ++ yada -- operatörü ile önek yada sonek biçiminde kullanılabilir. Bu durumda enum türünden değişkenin içerisinde önceki değerinin bir fazlası ya da bir eksiği bulunacaktır. Aynı türden iki enum birbiri ile toplanamaz.

Enum türleri herhangi bir türden türetilemez ve enum türünden de türetme yapılamaz. Fakat herhangi bir enum türünün System.Enum isimli sınıftan türetilmiş olduğu varsayılmaktadır. System.Enum türü de System.ValueType sınıfında türetilmiştir.



Bu durumda tüm türlere ilişkin türetme şeması şöyledir:



Delege İŞLEMLERİ

Delege (delegate) fonksiyon tutan özel bir sınıftır. Biz bir delegeye istediğimiz bir fonksiyonu verdiğimizde delege bu fonksiyonu tutar. Sonra biz delegenin tuttuğu fonksiyonun çağrılmasını delegeden isteyebiliriz.

Anahtar Notlar

Aşağı seviyeli ele alınırsa aslında her fonksiyonun bir adresi vardır. Bir fonksiyonun tutulması fonksiyonun kodlarının değil adresinin tutulması anlamındadır. Adresini bildiğimiz bir fonksiyonun çağrılmasını sağlayabiliriz.

Delege sınıf bildiriminin genel biçimi şöyledir:

```
delegate <geri dönüş değeri türü> <delege sınıf isim> ([
parametre bildirimi ]);
```

Örneğin:

```
delegate void Foo();
delegate int Bar(int a, int b);
```

Görüldüğü gibi delege sınıf bildirimi sanki bir fonksiyon bildirimi gibi yapılmaktadır. Fakat aslında yukarıdaki örnekte Foo ve Bar birer sınıftır. Delege sınıfı terimi ile delege aynı anlamda kullanılacaktır.

Delegeler kategori olarak referans türlerine ilişkindir. Yani bir delege türünden değişken tanımlandığında o bir referanstır ve o referans heap ta delege nesnesini göstermektedir.

Delege nesnesinin kendisi tıpkı diğer sınıf nesnelerinde olduğu gibi new operatörü ile yaratılır.

Bir delege sınıfı her türden fonksiyonu tutmaz. Ancak geri dönüş değerinin türü ve parametrelerin türü uygun olan fonksiyonları tutar. Örneğin:

```
delegate void Foo();
```

Burada Foo isimli delege sınıfı static olsun yada olmasın geri dönüş değeri void olan ve parametresi olmayan fonksiyonları tutar. Örneğin:

```
delegate int Bar(int a, int b);
```

Burada Bar isimli delege sınıfı, geri dönüş değeri int, parametre türleri int,int olan fonksiyonları tutar. Delege bildiriminde parametre değişken isimlerinin hiçbir önemi yoktur. Yani delegenin tutacağı fonksiyonun parametre değişkenlerinin isimleri bildirimdeki ile uyuşmak zorunda değildir.

Her delege sınıfının, tutulacak fonksiyonu parametre olarak alan bir başlangıç fonksiyonu vardır. Delege sınıflarının default başlangıç fonksiyonları yoktur. Örneğin:

```
delegate void Proc();
//...

Proc p = new Proc();    à error delege sınıflarında default başlangıç fonksiyonları yoktur.

Proc p = new Proc(Sample.Func);  à geçerli
```

d bir delege referansı olmak üzere, bu delege referansı d(...) biçiminde fonksiyon çağırma operatörü ile kullanılabilir. Bu durumda delegenin tuttuğu fonksiyonlar çağrılır. Çağırma ifadesinden çağrılan gerçek fonksiyonun geri dönüş değeri elde edilir.

```
using System;
using System.Collections;
using System.Windows.Forms; //Windows.Forms dll referans

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Proc p = new Proc(Sample.Func);
            p();
        }
    }
    delegate void Proc();
```

```
    class Sample
    {
        public static void Func()
        {
            Console.WriteLine("Func Called");
        }
    }
}
```

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Proc p = new Proc(Sample.Add);
            int result = p(10, 20);

            Console.WriteLine(result);

        }
    }
    delegate int Proc(int a, int b);

    class Sample
    {
        public static int Add(int x, int y)
        {
            return x + y;
        }
    }
}
```

Bir delege yaratılırken, delegenin tutacağı fonksiyonun ismi niteliksiz olarak belirtilirse söz konusu fonksiyon, isim arama kuralına göre bulunan sınıfta aranacaktır. Örneğin:

```
class App
{
     public static void Main()
     {
          Proc d = new Proc(Add);
          //...
     }
     public static int Add(int x, int y)
     {
          return x + y;
     }
}
```

Burada fonksiyon ismini `App.Add` belirmeye gerek yoktur.

Bir delege static olmayan fonksiyonları da tutabilir. Fakat static olmayan fonksiyonlar, `referans.fonksiyon` ismi biçiminde verilmelidir. Böylece delege hem referans hem de fonksiyonu tutar. Bu fonksiyon çağrılacağı zaman o referansla çağırır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s = new Sample(100);

            Proc d = new Proc(s.Disp);
            d();
        }
    }
    delegate void Proc();

    class Sample
    {
        private int m_a;

        public Sample(int a)
        {
            m_a = a;
        }
        public void Disp()
        {
            Console.WriteLine(m_a);
        }
        //...
    }
}
```

Görüldüğü gibi `d` isimli delege nesnesi `s` referansı ile birlikte `Disp()` fonksiyonunu tutmaktadır. `d()` ifadesi ile aslında `s.Disp()` işlemi yapılmaktadır.

Aynı türden iki delege referansı `+` operatörü ile toplama işlemine sokulabilir. Bu işlem sonucunda yeni bir delege nesnesi yaratılır. Bu yeni delege nesnesinin fonksiyon listesi iki delegenin tuttuğu fonksiyonların birleşiminden oluşur. Örneğin:

```
Proc d1 = new Proc(Sample.Foo);

Proc d1 = new Proc(Sample.Bar);

Proc d3;
```

```
d3 = d1 + d2;
```

Burada d1 delegesi Sample.Foo(), d2 delegesi Sample.Bar() fonksiyonlarını tutmaktadır. d1 + d2 işlemi ile yeni bir delege nesnesi yaratılır. Bu delege nesnesinin fonksiyon listesi Sample.Foo ve Sample.Bar fonksiyonlarından oluşur. Görüldüğü gibi başlangıçta delege nesneleri tek bir fonksiyonu tutacak şekilde yaratılmaktadır. Fakat daha sonra + operatörü ile birden fazla fonksiyon tutabilen delege nesneleri oluşturulabilir.

Bir delege referansına fonksiyon çağırma operatörü uygulandığında o delegenin tuttuğu fonksiyonların hepsi sırası ile çağrılır. Eğer delegenin geri dönüş değeri varsa, geri dönüş değeri olarak, son çağrılan fonksiyonun geri dönüş değeri elde edilir.

İki delege toplanırken delege referanslarından birinde null referans varsa toplama işleminde yeni bir delege nesnesi yaratılmaz. null olmayan delegenin referansı elde edilir. Örneğin:

```
Proc d1 = new Proc(Sample.Foo);

Proc d2 = null;

d3 = d1 + d2;
```

Bu işlemden sonra d3 ile d1 aynı delege nesnesini gösterir. Eğer toplama işleminde her iki delege referansı da null içeriyorsa toplama işleminin sonucunda null referans elde edilir.

a += b işlemi a = a+b anlamına geldiğine göre aşağıda işlem geçerlidir.

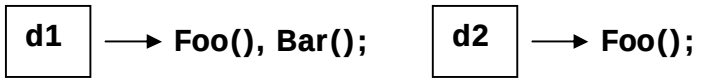
```
Proc d = null;

d += new Proc(Sample.Foo);

d();
```

Burada d referansının gösterdiği delege nesnesi Sample.Foo() fonksiyonun tutuyor durumdadır. Örneğin:

```
Proc d = null;

d += new Proc(Sample.Foo);
d += new Proc(Sample.Bar);

d();
```

Burada d referansının gösterdiği delege nesnesinde hem Foo() hem de Bar() fonksiyonları tutulmaktadır.

Aynı türden iki delege referansı – operatörü ile çıkartılabilir. Çıkartma sonucunda yeni bir delege nesnesi yaratılır. Yeni yaratılan delege nesnesinin fonksiyon listesinde sol taraftaki delegenin fonksiyon listesinden, sağ taraftaki delege nesnesinin fonksiyon listesinin çıkartılması ile elde edilen fonksiyon listesine sahip, yeni bir delege nesnesi elde edilir. Örneğin:

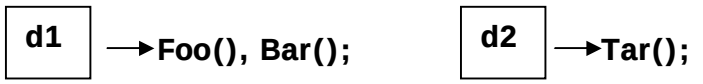



```
d3 = d1 – d2;
```

Çıkartma işleminden elde edilen delegenin fonksiyon listesinde `Bar()` fonksiyonu bulunur. Örneğin:

```
Proc d1 = new Proc(Sample.Foo);
d1 += new Proc(Sample.Bar);

Proc d2 = new Proc(Sample.Foo);
Proc d3 = d1 – d2;

d3();
```

Eğer çıkartma işleminde sol taraftaki delegenin fonksiyon listesinde, sağ taraftaki delegenin fonksiyon listesi yoksa bu durumda yeni bir nesne yaratılmaz. İşlem sonucunda sol taraftaki delege referansının aynısı elde edilir. Örneğin:




```
d3 = d1 – d2;
```

Burada çıkartma işleminin sonucu olarak `d1` referansının aynısı elde edilir. Yani `d1` ve `d3` referanslarında aynı adres bulunmaktadır. <u>Çıkartma işleminde sıra önemlidir</u>. <u>Yani çıkartmanın gerçekleşmesi için fonksiyonların aynı sırada bulunması gerekir</u>. Örneğin:


```
d3 = d1 – d2;
```

Burada `d2`'nin fonksiyon listesi, `d1`'in fonksiyon listesinde aynı sırada yoktur. Dolayısıyla çıkartma işleminde `d1` referansı elde edilir.

Sağdaki delegenin fonksiyon listesi soldaki delegenin fonksiyon listesinde birden fazla kez varsa, listede en son bulunan çıkartılır. Yani son eklenen çıkartılır. Örneğin:

```
d3 = d1 – d2;
```

Burada yeni bir delege nesnesi yaratılacaktır. Onun fonksiyon listesinde Func1 + Func2 sırasında fonksiyon bulunacaktır.

Çıkartma işleminde sağ taraftaki delege referansında null varsa yani d1 – d2 işleminde d2 de null referans varsa çıkartma işleminde yeni bir nesne yaratılmaz. D1 referansının aynısı elde edilir. Eğer soldaki operand da null varsa yada her iki operand da null varsa çıkartma işleminde null referans elde edilir. Çıkartma işleminin uygulandığı her iki operandın da fonksiyon listesinde aynı fonksiyonlar bulunuyorsa (d1 – d1 işleminde olduğu gibi) yine çıkartma işleminden null referans elde edilir.

Toplama ve çıkartma işleminde operandlardan biri, bir delege referansı ise diğeri o delegenin tutacağı bir fonksiyon olabilir. Bu durumda fonksiyon, otomatik olarak bir delege nesnesi yaratılarak onun içerisine yerleştirilmektedir. Örneğin:

```
Proc d1 = new Proc(Sample.Foo);

d2 = d1(delege) + Sample.Bar(fonksiyon)
```

Bu işlemle aşağıdaki eşdeğerdir.

```
d2 = d1 + new Proc(Sample.Bar);
```

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Proc d1 = new Proc(Sample.Foo);
            Proc d2;

            d2 = d1 + Sample.Bar;

            d2();
        }
    }
    delegate void Proc();

    class Sample
    {
        public static void Foo()
        {
```

```
            Console.WriteLine("Sample.Foo");
        }
        public static void Bar()
        {
            Console.WriteLine("Sample.Bar");
        }
    }

}
```

Aynı dönüştürme atama işlemi içinde söz konusudur. Örneğin:

```
Proc d = Sample.Foo;  à geçerli
```

Şüphesiz += yada -= operatörü ile de aynı kullanım geçerlidir.

```
Proc d = Sample.Foo;  à geçerli
d += Sample.Bar;
```

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int[] a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

            DoForEach(a, new DoProc(Display));
            DoForEach(a, new DoProc(Add));
            Console.WriteLine(m_total);

        }

        public static void DoForEach(int[] a, DoProc dp)
        {
            foreach (int x in a)
                dp(x);
        }
        public static void Display(int a)
        {
            Console.WriteLine(a);
        }
        public static void Add(int a)
        {
            m_total += a;
        }
        private static int m_total;
    }
    delegate void DoProc(int a);
}
```

SINIFLARIN VE YAPILARIN EVENT ELEMANLARI

Bir sınıfın ya da yapının delege türünden veri elemanları event eleman yapılabilir. Bunun için event anahtar sözcüğü, erişim belirleyicisi ve varsa static anahtar sözcüğünden sonra getirilir. Örneğin:

```
delegate void Proc();

class Sample
{
     public Proc A;
     public event Proc B;
     public event int C; à error delege değil
     public event string D; à error delege değil
     //..
}
```

Sınıfın ya da yapının event olmayan delege türünden veri elemanları kısıtlamasız olarak(eğer public bölümde ise ve erişim geçerliliği varsa) tam bir delege olarak kullanılabilir. Örneğin:

```
Sample s = new Sample();

s.A = new Proc(Test.Foo);
s.A();
```

Bir event delege eleman sınıfın dışarısında ancak += ve -= operatörleri ile kullanılabilir. Sınıfın ya da yapının dışında event delege elemana = operatörü ile atama yapılamaz. Delege fonksiyonları fonksiyon çağırma operatörleri ile çağrılamaz. Event delege eleman += ve -= operatörü dışında hiçbir şekilde dışarıdan kullanılamaz(halbuki delege eleman event yapılmasaydı böyle bir kısıt söz konusu olmayacaktı ve dışarıdan istenildiği gibi kullanılacaktı). r bir sınıf ya da yapı türünden referans ya da değişken, E de Proc isimli bir delege türünden event veri elemanı olsun.

```
r.E = new Proc(Tes.Foo);  à error atama yapılamaz
r.E();  à error çağrılamaz.

r.E += new Proc(Test.Foo);  à geçerli
r.E -= new Proc(Test.Foo);  à geçerli
```

Bir event delege eleman kendi sınıfı içerisinde kısıtlamasız olarak tam bir delege yeteneği ile kullanılır.

Anımsanacağı gibi türemiş sınıf, taban sınıfın public ve protected bölümlerine erişebilir. Fakat event erişiminde türemiş sınıfta taban sınıfın event elemanlarının += ve -= operatörleri ile kullanabilmektedir. Yani event elemanları tam bir delege olarak, yalnızca kendi sınıf ya da yapısı kullanabilmektedir.

Madem ki event elemanlar dışarıdan yalnızca fonksiyon eklemek ve çıkartmak amaçlı kullanılabiliyor peki bu durumda event elemanın fonksiyonları nasıl çağrılacaktır? Şüphesiz çağırma ancak dolaylı olabilmektedir. Yani yapı ya da sınıfın public bir fonksiyonu event fonksiyonlarını çağırmaktadır. Örneğin:

```
r.E += new Proc(Test.Foo);

r.Fire();
```

Burada Fire() fonksiyonu aşağıdaki gibi yazılmış olabilir:

```
public void Fire()
{
     //...
     E();
     //...
}
```

event anahtar sözcüğü bir tür belirtmez. event eleman delege türündendir. event anahtar sözcüğü yalnızca delegeye dışarıdan kullanım kısıtlığı getirmektedir.

.Net'in sınıf kütüphanesindeki pek çok sınıf ve yapının delege türünden veri elemanları vardır. Bu veri elemanları event yapılarak dışarının kullanımı için kısıtlanmıştır.

EVENT ELEMANLARIN .NET PROGRAMLAMA MODELİNDE KULLANILMASI

.Net'in sınıf kütüphanesinde pek çok sınıfın ve yapının event elemanı vardır. Bu event elemanlara fonksiyonlar eklenip çıkartılabilir. Bazı olaylar gerçekleştiğinde framework, bu event elemana ait event fonksiyonları çağırır.

Örneğin; düğmeler Button isimli sınıfla temsil edilmiştir. Button sınıfının aşağıdaki gibi Click elemanı vardır:

```
public event EventHandler Click;
```

Görüldüğü gibi Click event elemanı EventHandler isimli bir delege türündendir. EventHandler isimli delege şöyledir:

```
delegate void EventHandler(object sender, EventArgs ea);
```

Düğmeye klik yapıldığında framework, Click eventinin tuttuğu fonksiyonları kendisi çağırmaktadır. Bu durumda düğmeye klik yapıldığında bir fonksiyonun çağrılmasını istiyorsak Click event elemanına bu fonksiyonu girmeliyiz.

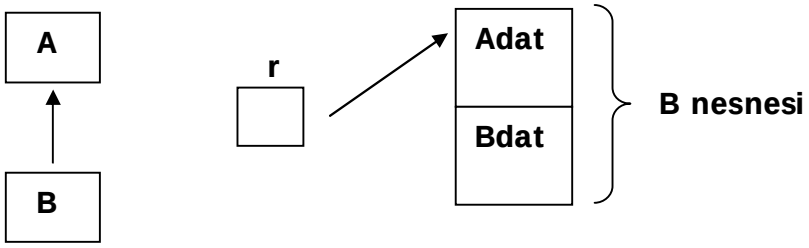
```
Button button = new Button();
//...
button.Click += new EventHandler(Test.Func);
```

Burada `Test` sınıfının `Func()` fonksiyonu, geri dönüş değeri `void`, birinci parametresi `Object` türünden, ikinci parametresi `EventArgs` sınıf türünden olmalıdır.

Görüldüğü gibi .Net'te ne zaman bir olay gerçekleştiğinde bir fonksiyonun çağrılması isteniyorsa, çağrılması istenen fonksiyon bir delegeye yerleştirilmektedir. O delege de bir sınıf ya da yapının veri elemanı durumdadır ve event yapılarak dışarıya kısıtlanmıştır.

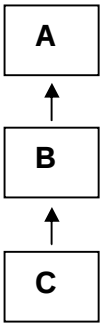
REFERANSLARIN STATIC VE DİNAMİK TÜRLERİ

Bir referansın `static` türü bildirimde belirtilen türdür. Referansın `dinamik` türü ise referansın gösterdiği bütünsel türdür. Örneğin referans daha büyük bir nesnenin bir parçasını gösteriyorsa, referansın dinamik türü o nesnenin bütünsel en geniş halinin türüdür.




```
A r = new B();
```

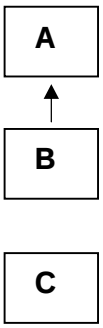
Burada `r` referansını <u>static türü `A` dır</u>. `r` referansının gösterdiği yerde bir `A` nesnesi yoktur. `B` nesnesinin `A` kısmı vardır. O halde `r` referansının <u>dinamik türü `B` dir</u>. Örneğin:



```
A x;
B y;
C z;

z = new C();
y = z;
x = y;
```

Burada z referansının static ve dinamik türü C dir. y referansının static türü B dinamik türü C dir. x referansının static türü A dinamik türü C dir.

Static tür bildirim sırasında bildirilir ve bir daha değişmez. Halbuki dinamik tür program içerisinde sürekli değişebilir. Örneğin:




```
A r;

r = new A();
// r nin dinamik türü A

r = new B();
// r nin dinamik türü B

r = new C();
// r nin dinamik türü C
```

Örneğin:

```
int a;
object o = a;
```

Burada o nun static türü `object`, dinamik türü `Int32` yani `int` türdendir.

ÇOKBİÇİMLİLİK(POLYMORPHISM)

Bir dilin nesne yönelimli olması için, o dilde sınıf kavramının olması, türetme kavramının olması ve çokbiçimliliğin bulunması gerekir. Çokbiçimlilik biyolojiden aktarılmış bir terimdir. Evrim sürecinde çeşitli canlıların alt türlere ayrılması sırasında bazı organların o alt türe özgü bir biçimde değişikliğe uğramasına denilmektedir. Örneğin kulak bir çok canlıda vardır ve duyma işlevine yerine getirmektedir. Fakat her canlıda kulak onun yaşam koşullarına özgü bir biçimde değişlik göstermiştir.

Nesne yönelimli programlama tekniğinde çokbiçimlilik üç şekilde tanımlanabilir:
1-Biyolojik Tanım: Taban sınıfın bir takım fonksiyonlarını türemiş sınıfların kendilerine özgü bir biçimde yeniden tanımlamalarına çokbiçimlilik denir.
2-Aşağı Seviyeli Tanım: Çokbiçimlilik önceden yazılmış kodların sonradan yazılan kodları çağırabilme özelliğidir.
3-Yazılım Mühendisliği Tanımı: Çokbiçimlilik türden bağımsız kod parçalarını oluşturabilmek için kullanılan bir tekniktir.

C# ta çokbiçimlilik sanal fonksiyonlarla gerçekleştirilmektedir.

Anahtar Notlar:

Javada default olarak her fonksiyon zaten sanaldır. Fakat bu durum etkinliği azaltabilmektedir. C#, javayı temel almakla birlikte sanal fonksiyon mekanizmasını daha çok C++ a benzetmiştir.

SANAL FONKSİYONLAR VE OVERRIDE İŞLEMLERİ

Bir sınıfın (fakat yapının değil) static olmayan bir fonksiyonu `virtual` anahtar sözcüğü kullanılarak sanal fonksiyon yapılabilir. Örneğin:

```
class Sample
{
     public virtual void Func()
     {
          //...
     }
     //...
}
```

`virtual` anahtar sözcüğü erişim belirleyici anahtar sözcüğü ile aynı syntax grubu içerisindedir. Yer değiştirmeli olarak kullanılabilir. (yani örneğin public virtual yada virtual public yazılabilir)

Taban sınıftaki bir sanal fonksiyon, türemiş sınıfta aynı erişim belirleyicisi, aynı isim, aynı geri dönüş değeri ve aynı parametrik yapı ile fakat `virtual` anahtar sözcüğü yerine `override` anahtar sözcüğü ile tanımlanırsa bu işlem "taban sınıftaki sanal fonksiyonun türemiş sınıfta override edilmesi" denilmektedir. Örneğin:

```
class A
{
     public virtual void Foo()
     {
          //...
     }
     //...
}
```

```
class B : A
{
     public override void Foo()
     {
          //...
     }
     //...
}
```

override etme işleminde şu kurallara uyulmalıdır:

- **Fonksiyon türemiş bölümde başka bir yerde override edilemez. Örneğin:**

```
class A
{
     public virtual void Foo()
     {
          //...
     }
     //...
}

class B : A
{
     protected override void Foo() à error
     {
          //...
     }
     //...
}
```

- Fonksiyonun override edilebilmesi için geri dönüş değerlerinin uyuşması gerekir. Örneğin:

```
class A
{
     public virtual void Foo()
     {
          //...
     }
     //...
}

class B : A
{
     public virtual int Foo() à error
     {
          //...
     }
     //...
}
```

- **Fonksiyon farklı bir parametrik yapı ile override edilmez. Örneğin:**

```
class A
{
     public virtual void Foo()
     {
          //...
     }
     //...
}

class B : A
{
     public virtual void Foo(int a) à error
     {
          //...
     }
     //...
}
```

Türemiş sınıfta override edilmiş bir fonksiyon, türemiş sınıftan türetilen başka bir sınıfta yeniden override edilebilir. Örneğin:

```
class A
{
     public virtual void Func()
     {
          //...
     }
     //...
}

class B : A
{
     public override void Func()
     {
          //...
     }
     //...
}
class C : B
{
     public override void Func()
     {
          //...
     }
     //...
}
```

Görüldüğü gibi fonksiyonun, en yukarıda virtual anahtar sözcüğü ile sanallığı başlatılır sonra hep override anahtar sözcüğü ile devam ettirilir.

Sanal fonksiyon terimi hem `virtual`, hem `override` hem de `abstract` fonksiyonları kapsayacak şekilde kullanılacaktır.

Şüphesiz bir fonksiyonun sanallığı herhangi bir türemiş sınıftan başlatılabilir.

Bir fonksiyonun `override` edilebilmesi için, o fonksiyonun hemen o sınıfın doğrudan taban sınıfında, `virtual` ya da `override` şeklinde tanımlanması gerekmez. Fakat şüphesiz yukarıya doğru herhangi bir taban sınıfta `override` ya da en kötü olasılıkla `virtual` biçiminde tanımlanmış olma zorunluluğu vardır. Örneğin:

```
Class A
{
     public virtual int Foo(int a)
     {
          //...
     }
     //...
}

class B :A
{
     //...
}

class C :B
{
     public override int Foo(int a)
     {
          //...
     }
     //...
}
```

Görüldüğü gibi `Foo()` fonksiyonu `B` sınıfında `override` edilmemiştir. Fakat `C` sınıfında `override` edilmiştir.

Bir sınıf, aynı isimli birçok fonksiyona sahip olabilir (function overloading). Fakat bunlarda belirli parametrik yapıya sahip olanlar sanal yapılmak istenebilir. Örneğin:

```
class A
{
     public void Func()
     {
          //...
     }
     public virtual void Func(int a)
     {
          //...
```

```
    }
    public void Func(long a)
    {
        //...
    }
}
```

Burada A sınıfında üç tane Func() fonksiyonu tanımlanmış olmasına karşın yalnızca int parametreli Func() fonksiyonu sanaldır. Dolayısıyla türemiş sınıflarda yalnız bu fonksiyon override edilebilir.

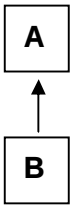
Taban sınıftaki bir sanal fonksiyon aynı isim, geri dönüş değeri ve parametrik yapı ile fakat override anahtar sözcüğü kullanılmadan türemiş sınıfta yeniden tanımlanabilir. Bu durum error ile sonuçlanmaz. Fakat bir uyarı oluşur. Bu durum override etme anlamı taşımamaktadır. Örneğin:

```
class A
{
    public virtual void Func()
    {
        //...
    }
}

class B :A
{
    public void Func() à error değil
    {
        //...
    }
}
```

Benzer biçimde taban sınıftaki virtual fonksiyon türemiş sınıfta yeniden virtual olarak tanımlanabilir. Bu durum da override etme anlamına gelmez. Bu özel durumlar new belirleyicisinin anlatıldığı bölümde ele alınacaktır.

SANALLIK MEKANİZMASI

Bir referansla bir fonksiyon çağrıldığında fonksiyon ismi, referansın static türüne ilişkin sınıfın faaliyet alanında aranır. Sonra fonksiyonun sanal olup olmadığına bakılır. Eğer fonksiyon sanal değilse bulunan fonksiyon çağrılır. Eğer fonksiyon sanalsa referansın dinamik türüne ilişkin sanal fonksiyon çağrılır. Örneğin:




```
class A
```

```
{
     public void Foo()
     {
          //...
     }
     public virtual void Bar()
     {
          //...
     }
     //...
}

class B : A
{
     public override void Bar()
     {
          //...
     }
     //...
}

B b = new B();
A a = b;
a.Bar();
```

Örneğin:


```
class A
{
     public void Foo()
     {
          //...
     }
     public virtual void Bar()
     {
          //...
     }
     //...
}

class B : A
{
     public override void Bar()
```

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

class C :B
{
     public override void Bar()
     {
          //...
     }
     //...
}

A a;
B b = new B();
C c = new C();
a = b;
a.Bar(); à B sınıfının Bar fonksiyonu çağrılır.
a = c;
a.Bar(); à C sınıfının Bar fonksiyonu çağrılır.
b = c;
b.Bar(); à C sınıfının Bar fonksiyonu çağrılır.
```

Anımsanacağı gibi sanal fonksiyonlar override edilmek zorunda değildir. Eğer sanal fonksiyon, referansın dinamik türüne ilişkin sınıfta override edilmedi ise yukarıya doğru bu sanal fonksiyonun override edildiği, ilk taban sınıfın sanal fonksiyonu çağrılır. Yukarıdaki örnekte eğer C sınıfında Bar() fonksiyonu override edilmiş olmasaydı B sınıfındaki Bar() fonksiyonu çağrılacaktı.

Anahtar Notlar

Java da static olmayan tüm fonksiyonlar default olarak sanal fonksiyon durumundadır. Dolayısıyla bu dilde virtual ya da override gibi anahtar sözcükler yoktur. Taban sınıftaki bir fonksiyon türemiş sınıfta aynı bölüm, aynı geri dönüş değeri türü ve aynı imza ile tanımlanırsa bu durum override etmek anlamına gelmektedir. Örneğin:

```
class A
{
     public void Func()
     {
          //...
     }
     //...
}

class B extends A
{
     public void Func()
```

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

A a = new B();
a.Func();
```

Burada adeta Func fonksiyonu A sınıfında virtual tanımlanmış ve B sınıfında override edilmiş gibidir.

Anahtar Notlar:

C++ ta sanallık yine virtual anahtar sözcüğü ile başlatılır. Fakat override anahtar sözcüğü bu dilde yoktur. Türemiş sınıfta eğer bir sanal fonksiyon aynı geri dönüş değeri ve imza ile bildirilirse fonksiyon override edilmiş olur.

```
class A {
public:
    virtual void Func()
    {
        //...
    }
    //...
};

class B : public A {
public:
    void Func();
    {
        //...
    }
    //...
};
```

object SINIFININ ToString SANAL FONKSİYONU

En tepedeki `object` sınıfının `ToString` isminde, bir string nesnesi (referansı) ile geri dönen sanal bir fonksiyonu vardır. `ToString` sanal fonksiyonu türemiş fonksiyonlarda `override` edilebilir. Örneğin:

```
class Sample
{
    public override string ToString
    {
        return "This is a test";
    }
    //...
}
object o;
```

```
o = new Sample();
Console.WriteLine(o.ToString());
```

Burada Sample sınıfının ToString fonksiyonu çağrılacaktır.

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            object o;
            Sample s = new Sample();
            o = s;
            Console.WriteLine(o.ToString());
        }
    }
    class Sample
    {
        public override string ToString()
        {
            return "this is a test";
        }
    }
}
```

Object sınıfının ToString fonksiyonu, referansın dinamik türüne ilişkin sınıf ismini geri döndürmektedir.

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
          Sample s = new Sample();
            Console.WriteLine(s.ToString());
        }
    }
    class Sample
    {

    }
}
```

Bir ArrayList collection sınıfı içerisinde, çeşitli sınıflara ilişkin nesnelerin referanslarını saklayabiliriz. Sonra bu collectionu dolaşarak ToString sanal fonksiyonun çağırırsak, her nesneye ilişkin sınıfın override edilmiş kendi ToString fonksiyonları çağrılır. Örneğin:

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();

            al.Add(new A());
            al.Add(new B());
            al.Add(new C());
            al.Add(new D());

            foreach (object o in al)
                Console.WriteLine(o.ToString());
        }
    }
    class A
    {
        public override string ToString()
        {
            return "A.ToString";
        }
    }
    class B
    {
        public override string ToString()
        {
            return "B.ToString";
        }
    }
    class C
    {
        public override string ToString()
        {
            return "C.ToString";
        }
    }
    class D
    {
        public override string ToString()
        {
            return "D.ToString";
        }
    }
}
```

.Net'teki sınıfların hemen hepsinde ToString fonksiyonları faydalı birtakım işler yapacak biçimde override edilmiştir. Örneğin: int, long, double

gibi yapılarda ToString fonksiyonları onların tuttukları değerleri yazı olarak gelecek biçimde yazılmıştır. Örneğin:

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int i = 123;

            Console.WriteLine("Sayı : " + i.ToString());
        }
    }
}
```

Örneğin:

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int i = 123;
          object o = i;

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

Toplama işleminde operandlardan biri string fakat diğer operand herhangi bir türden ise diğer operand üzerinde derleyici tarafından ToString fonksiyonu uygulanır. Elde edilen yazı birleştirilir. Örneğin:

```
int i = 123;

string s = “Sayi : ” + i;
```

Console sınıfının object türünden parametreye sahip Write ve WriteLine fonksiyonları vardır. Biz bu fonksiyonlara herhangi bir sınıf türünden referans geçirdiğimizde overload resolution işlemine göre object parametreli Write yada WriteLine fonksiyonu çağrılır. Bu fonksiyon da

parametresine ToString sanal fonksiyonunu uygulayarak elde ettiği yazıyı yazdırır. Örneğin:

```
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 "A.ToString";
        }
    }
}
```

Örneğin; DateTime yapısının ToString sanal fonksiyonu yapının tuttuğu tarih ve zamanı yazı olarak verir. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Console.WriteLine(DateTime.Now);
        }
    }
}
```

ÇOKBİÇİMLİLİĞE İLİŞKİN ÖRNEKLER

Çokbiçimliliğe ilişkin mekanizma yukarıda açıklandı. Bu mekanizma bir referansla sanal fonksiyon çağrıldığında, referansın dinamik türüne ilişkin sınıfın sanal fonksiyonunun çağrılması anlamındadır. Peki bu mekanizmanın programcıya ne faydası vardır?

Çokbiçimlilik türden bağımsız kod parçalarının oluşturulması için kullanılmaktadır. Bir projede değişebilecek öğeler tespit edilir ve bunlara çokbiçimli mekanizma ile erişilir. Yani çokbiçimlilik gerçekte farklı olan nesnelerin aynı türdenmiş gibi ele alınmasını sağlamaktadır.

Nesne yönelimli bir proje modellemesinde, projedeki tüm gerçek nesneler ve kavramlar sınıflarla temsil edilir. Sonra sınıflar arasında ilişkiler belirlenir.

1-Topla oynanan bir oyun programı yazacağımızı düşünelim. Oyundaki top çeşitli seçeneklere göre değişebilecek şekilde yazılmak istenmektedir. Örneğin top yuvarlak olabilir, dikdörtgen olabilir, zıplayan top olabilir, daha pek çok farklı top olabilir. Fakat ne olursa olsun topun bazı temel işlevleri vardır. Bu temel işlevler, tema olarak aynı kalmakla beraber toptan topa farklılık göstermektedir. İşte top bir sınıfla temsil edilebilir ve topun temel işlevleri sanal fonksiyonlarla belirlenebilir.

```
class Ball
{
     public virtual void Move()
     {
          //...
     }
     public virtual void Jump()
     {
          //...
     }
     //...
}
```

Şimdi çeşitli toplar `Ball` sınıfından türetilen sınıflar biçiminde oluşturulur.

```
class CircleBall : Ball
{
     public override void Move()
     {
          //...
     }
     public override void Jump()
     {
          //...
     }
     //...
}

class RectagleBall : Ball
{
     public override void Move()
     {
          //...
     }
     public override void Jump()
     {
          //...
     }
     //...
}
```

Şimdi bu topu kullanan bir program parçası düşünelim.

```
Ball ball = new CircleBall();

ball.Move();
ball.Jump();
```

Görüldüğü gibi burada top kavramı, genel bir kavram olarak kodda kullanılmıştır. İşlemler `Ball` sınıfı türünden referansla gerçekleştirilmiştir. Burada çağrılan fonksiyonlar aslında `CircleBall` sınıfının fonksiyonlarıdır. Oyundaki topu değiştirmek için tek yapılacak şey `new CircleBall` ifadesi yerine diğer bir top sınıfının yaratım ifadesini yerleştirmektir. Çünkü topu kullanan tüm kodlar, taban `Ball` sınıfının sanal fonksiyonları ile bu işi yapmaktadır. Programa yeni bir top eklemekte oldukça kolaydır. Tek yapılacak şey, `Ball` sınıfında yeni bir sınıf türetip, ilgili sanal fonksiyonları `override` etmektir. Bu tür durumlarda eskiden yazılmış kodlar geçerliliğini koruyacağı için bunların yeniden test edilmesine gerek kalmayacaktır.

2-Bir tetris programı yazmak isteyelim. Oyunun kendisi bir sınıf ile temsil edilebilir. Bu sınıfın `Run` isimli bir fonksiyonu oyunu başlatabilir.

```
class Tetris
{
     //...
}
//...

Tetris t = new Tetris();
t.Run();
```

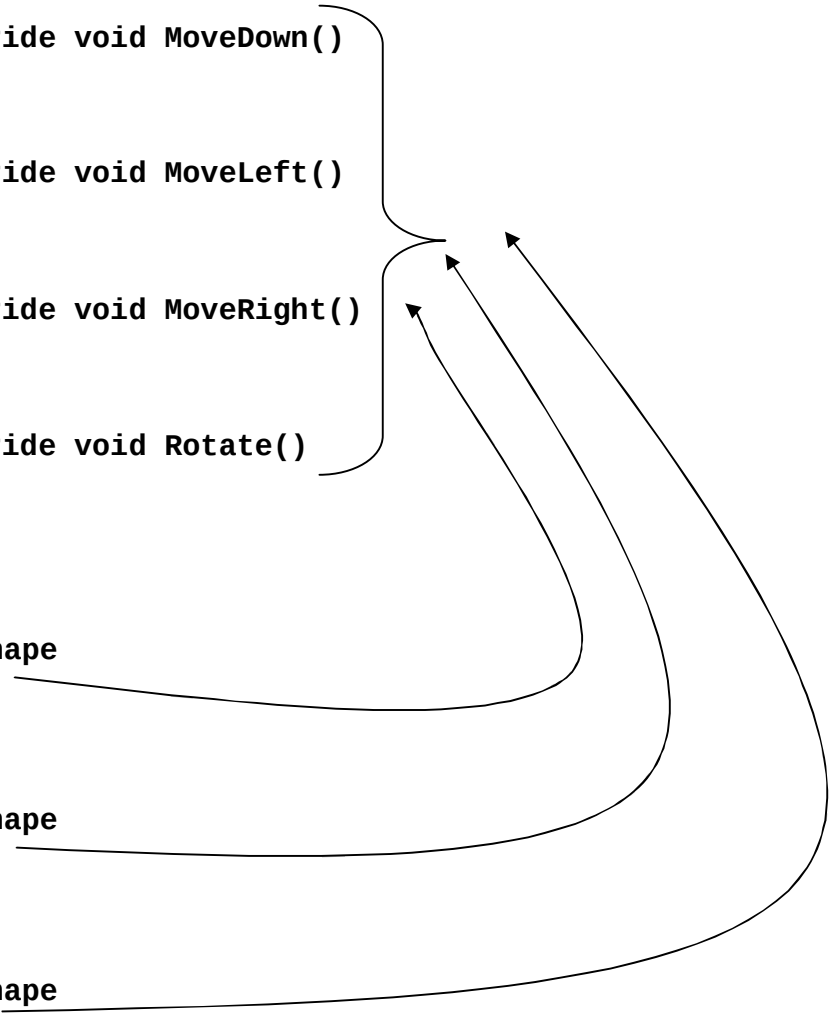
Biz burada oyundaki şekiller üzerinde duracağız. Oyundaki şekiller benzer işlevlere sahiptir fakat kendine özgü hareketlere sahiptir. Kodlama yapılırken tüm şekiller sanki aynı türdenmiş gibi işleme sokulur. Tüm şekiller `Shape` sınıfı ile temsil edilebilir.

```
class Shape
{
     public virtual void MoveDown()
     {
          //...
     }
     public virtual void MoveLeft()
     {
          //...
     }
     public virtual void MoveRight()
     {
          //...
     }
     public virtual void Rotate()
```

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

Tüm şekil sınıfları Shape sınıfından türetilen sınıflarla temsil edilir. Her şekil sınıfında Shape sınıfındaki sanal fonksiyonlar o sınıfa özgü olarak override edilecektir.

```
class BarShape : Shape
{
    public override void MoveDown()
    {
        //...
    }
    public override void MoveLeft()
    {
        //...
    }
    public override void MoveRight()
    {
        //...
    }
    public override void Rotate()
    {
        //...
    }
}

class LShape : Shape
{
    //...
}

class ZShape : Shape
{
    //...
}

class TShape : Shape
{
    //...
}
```



Şimdi artık oyunda düşen şeklin hareket ettirilmesi türden bağımsız olarak ifade edilebilecektir.

```
using System;

namespace CSD
{
```

```
    class App
    {
        public void Run()
        {
            for (; ; )
            {
                Shape shape = GetRandomShape();

                for (int i = 0; i < 10; ++i)
                {
                    ConsoleKeyInfo cki = Console.ReadKey(true);
                    switch (cki.Key)
                    {
                        case ConsoleKey.LeftArrow:
                            shape.MoveLeft();
                            break;
                        case ConsoleKey.RightArrow:
                            shape.MoveRight();
                            break;
                        case ConsoleKey.UpArrow:
                            shape.Rotate();
                            break;
                        case ConsoleKey.Escape:
                            goto EXIT;
                    }
                    shape.MoveDown();
                    System.Threading.Thread.Sleep(600);
                }
            }
            EXIT:
                ;
        }
    }
}
```

Yukarıdaki kod tetris oyununu simüle etmektedir. GetRandomShape rasgele bir şekil elde eder. Bu şekil uygun biçimde hareket ettirilir. Görüldüğü gibi kod, düşen şeklin ne olduğunu bilmeden yazılmış olan genel bir koddur. Oyuna yeni bir şekil ekleyecek olsak tek yapacağımız şey Shape sınıfında yeni bir sınıf türetip, Shape sınıfındaki sanal fonksiyonları override etmektir. Fakat Run fonksiyonunda bir değişiklik yapılmayacaktır.

Tetris.cs

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Tetris tetris = new Tetris();
```

```
            tetris.Run();
        }
    }

    enum ShapeType
    {
        BarShape, LShape, ZShape, TShape, SquareShape,
    }

    class Tetris
    {
        public void Run()
        {
            for (; ; )
            {
                Shape shape = GetRandomShape();

                for (int i = 0; i < 10; ++i)
                {
                    if (Console.KeyAvailable)
                    {
                        ConsoleKeyInfo cki =
Console.ReadKey(true);
                        switch (cki.Key)
                        {
                            case ConsoleKey.LeftArrow:
                                shape.MoveLeft();
                                break;
                            case ConsoleKey.RightArrow:
                                shape.MoveRight();
                                break;
                            case ConsoleKey.UpArrow:
                                shape.Rotate();
                                break;
                            case ConsoleKey.Escape:
                                goto EXIT;
                        }
                    }

                    shape.MoveDown();
                    System.Threading.Thread.Sleep(600);
                }

            }

        EXIT:
            ;
        }

        private Shape GetRandomShape()
        {
            Shape shape = null;
            Random r = new Random();
            int val = r.Next(4);
```

```
            switch ((ShapeType)val)
            {
                case ShapeType.BarShape:
                    shape = new BarShape();
                    break;
                case ShapeType.LShape:
                    shape = new LShape();
                    break;
                case ShapeType.SquareShape:
                    shape = new SquareShape();
                    break;
                case ShapeType.TShape:
                    shape = new TShape();
                    break;
                case ShapeType.ZShape:
                    shape = new ZShape();
                    break;
            }

            return shape;
        }
        //...
    }
}
```

shape.cs

```
using System;

namespace CSD
{
    class Shape
    {
        public virtual void MoveLeft()
        { }
        public virtual void MoveRight()
        { }
        public virtual void MoveDown()
        { }
        public virtual void Rotate()
        { }
        //...
    }

    class BarShape : Shape
    {
        public override void MoveLeft()
        {
            Console.WriteLine("BarShape MoveLeft");
        }
        public override void MoveRight()
        {
```

```
            Console.WriteLine("BarShape MoveRight");
        }
        public override void MoveDown()
        {
            Console.WriteLine("BarShape MoveDown");
        }
        public override void Rotate()
        {
            Console.WriteLine("BarShape Rotate");
        }
        //...
    }

    class LShape : Shape
    {
        public override void MoveLeft()
        {
            Console.WriteLine("LShape MoveLeft");
        }
        public override void MoveRight()
        {
            Console.WriteLine("LShape MoveRight");
        }
        public override void MoveDown()
        {
            Console.WriteLine("LShape MoveDown");
        }
        public override void Rotate()
        {
            Console.WriteLine("LShape Rotate");
        }
        //...
    }

    class ZShape : Shape
    {
        public override void MoveLeft()
        {
            Console.WriteLine("ZShape MoveLeft");
        }
        public override void MoveRight()
        {
            Console.WriteLine("ZShape MoveRight");
        }
        public override void MoveDown()
        {
            Console.WriteLine("ZShape MoveDown");
        }
        public override void Rotate()
        {
            Console.WriteLine("ZShape Rotate");
        }
        //...
    }
```

```
class TShape : Shape
{
    public override void MoveLeft()
    {
        Console.WriteLine("TShape MoveLeft");
    }
    public override void MoveRight()
    {
        Console.WriteLine("TShape MoveRight");
    }
    public override void MoveDown()
    {
        Console.WriteLine("TShape MoveDown");
    }
    public override void Rotate()
    {
        Console.WriteLine("TShape Rotate");
    }
    //...
}

class SquareShape : Shape
{
    public override void MoveLeft()
    {
        Console.WriteLine("SquareShape MoveLeft");
    }
    public override void MoveRight()
    {
        Console.WriteLine("SquareShape MoveRight");
    }
    public override void MoveDown()
    {
        Console.WriteLine("SquareShape MoveDown");
    }
    public override void Rotate()
    {
        Console.WriteLine("SquareShape Rotate");
    }
    //...
}

class Shape
{
    public bool IsInside(int x, int y)
    { }
    public void Paint(Graphics g)
    { }
    public void Select(Graphics g)
    { }
    public void Deselect(Graphics g)
    { }
    //...
}
```

```
}
```

3-PowerPoint benzeri bir program yazacak olalım. Bu programda kabaca dikdörtgen gibi, daire gibi, çizgi gibi çeşitli geometrik şekiller çizilebilmektedir. Tüm bu geometrik şekiller Shape isimli taban bir sınıftan türetilen sınıflarla temsil edilebilir. Her şekil sınıfı o şeklin koordinatlarını tutabilir. Programda çizilen tüm şekiller temel nitelik olarak birer şekildir. Fakat çokbiçimli bir davranış göstermektedir. Örneğin; bir şeklin üstüne tıklandığında her şeklin seçilmesi yani şekilde küçük kutucuklar çıkarılması farklı olmaktadır. O halde bu işi yapacak Shape sınıfının Select isimli sanal bir fonksiyonu olabilir. Bu fonksiyon türemiş sınıflarda override edilebilir. Benzer biçimde tıklanan noktanın şeklin içinde olup olmadığı, her şekil için o şekle özgü biçimde belirlenmektedir. Bu işlemde çokbiçimlidir. Şekillerin çizilmesi yine kendilerine özgü bir biçimde yapılmaktadır. O halde Shape taban sınıfı aşağıdaki biçimde olabilir:

```
class Shape
{
     public bool IsInside(int x, int y)
     {}
     public void Paint(Graphics g)
     {}
     public void Select(Graphics g)
     {}
     public void Deselect(Graphics g)
     {}
}
```

Çokbiçimli uygulamaların çoğunda bir türetme şeması içinde bulunan sınıflara ilişkin nesneler, bir collection içerisinde tutulur. Sonra bu collection dolaşılarak, çokbiçimli bir biçimde sanal fonksiyonlar çağrılır. PowerPoint uygulamasında da, çizilen her şekil için bir sınıf nesnesi yaratılmalı ve bu nesne ArrayList gibi bir collection içerisine yerleştirilmeli. Böylece programcı çizilen her şekli onlar sanki bir Shape nesnesiymiş gibi saklar. Böylece fare ile bir yere tıklandığında programcı tıklanan noktanın bir şekil içerisinde olup olmadığını anlamaya çalışır.

Tıklanan noktanın bir şeklin içerisinde olup olmadığının anlaşılması için tüm şekillerin çokbiçimli bir şekilde dolaşılması gerekir. Bunun için çokbiçimli IsInside fonksiyonu kullanılır. Basılan noktanın şeklin içinde olup olmadığı aşağıdaki gibi sorgulanabilir:

```
foreach (Shape shape in m_shapes)
     if (shape.IsInside(e.X, e.Y))
     {
          //...
     }
```

abstract FONKSİYONLAR VE SINIFLAR

Çokbiçimli uygulamaların çoğunda, en tepede bir taban sınıf bulunur ve bu en tepedeki sınıf, türden bağımsız işlemleri yapmak için kullanılır. Yani aslında bu en tepedeki taban sınıf türünden hiç nesne tanımlanmaz. Örneğin tetris uygulamasında ya da PowerPoint uygulamasında en tepedeki Shape sınıfı aslında çokbiçimli etki yaratmak için kullanılmıştır.

Bir fonksiyonu `abstract` yapmak için, `abstract` anahtar sözcüğü kullanılır. `Abstrack` fonksiyonların gövdesi yoktur. Fonksiyon bildirimi ";" ile kapatılır. Örneğin:

```
public abstract void Func(int a, int b);
```

`Abstract` anahtar sözcüğü, erişim belirten anahtar sözcüklerle aynı syntax grubundandır. Bunlarla yer değiştirmeli olarak yazılabilir. Fonksiyonlar hem `abstract` hem de `virtual` olamaz. `Abstract` fonksiyon, gövdesi olmayan `virtual` fonksiyondur.

En az bir `abstract` elemana sahip sınıfa `abstract` sınıf denir. Bu durumda, vurgulama amacı ile sınıf bildiriminin başına ayrıca `abstract` anahtar sözcüğü de yazılmak zorundadır. Örneğin:

```
abstract class Sample
{
     private int a;
     //...

     public Sampe()
     {
          //...
     }

     public void Test()
     {
          //...
     }

     public abstract void Func();
}
```

Bir `abstract` sınıf, ayrıca veri elemanlarına, `abstract` olmayan fonksiyonlara sahip olabilir. Sınıfın `abstract` olabilmesi için, en az bir elemanının `abstract` olması yeterlidir.

`Abstract` sınıf türünden referanslar tanımlanabilir. Fakat `new` operatörü ile nesne yaratılamaz. Eğer `new` operatörü ile nesne yaratılmak istenirse derleme zamanında error oluşur. Örneğin:

```
public static void Main()
```

```
{
     Sample s; à geçerli

     s = new Sample() à error
}
```

Abstract sınıf türünden nesne, new operatörü ile neden yaratılamamaktadır? Eğer yaratılabilseydi olmayan bir fonksiyonun çağrılması gibi potansiyel bir durum oluşurdu.

Bir abstract sınıftan sınıf türetildiğinde, türemiş sınıf taban sınıfın tüm abstract fonksiyonlarını override etmek zorundadır. Aksi halde türemiş sınıfta abstract olur. Türemiş sınıf türünden de new operatörü ile nesne yaratılamaz. Türemiş sınıf bildiriminin önüne de abstract anahtar sözcüğünü yerleştirmek gerekir. Örneğin:

```
abstract class A
{
     public abstract void Foo();
}

class B : A
{
     public override void Foo()
     {
          //...
     }
}
```

Bu örnekte A sınıfı, abstract sınıftır. Fakat B sınıfı, abstract sınıf değildir. Çünkü B sınıfı, taban A sınıfının tüm abstract fonksiyonlarını override etmiştir. B sınıfı türünden new operatörü ile yeni bir nesne yaratılabilir. Fakat A sınıfı türünden nesne yaratılamaz. Örneğin:

```
A a = new B();

a.Foo(); à B.Foo fonksiyonu çağrılacak.
```

Şüphesiz sanal fonksiyonu abstract yapmak yerine virtual yapıp boş bir gövde yazmak işlevsel olarak aynı etkiye yol açacaktır. Fakat tetris ve powerpoint örneklerinde olduğu gibi eğer en tepedeki taban sınıf türden bağımsız işlevler için kullanılıyorsa sınıfın abstract yapılması bu kullanım vurgusuna yol açar. Biz bir abstract sınıf gördüğümüzde, o sınıf türünden nesne yaratılamayacağını, o sınıfın tamamen türden bağımsız işlemler yapmak amacıyla kullanılacağını anlamalıyız. Örneğin tetris ve powerpoint uygulamalarında en tepedeki taban sınıf bu anlamda abstract yapılmalıdır.

Anahtar Notlar

Abstract sözcüğü bu bağlamda soyut anlamına gelmektedir. Soyut sınıf fiziksel olarak varolmayan fakat kavramsal olarak varolan sınıf anlamına gelmektedir.

Taban sınıfın pek çok `abstract` elemanı olabilir. Türemiş sınıfta bunların hepsi `override` edilmedi ise türemiş sınıfta `abstract` olur. `A` isimli taban `abstract` sınıfın, bir grup `abstract` elemanı türemiş `B` sınıfında `override` edilmiş olsun. Geri kalan `abstract` elemanları da `B` sınıfından türetilmiş, `C` sınıfında `override` edilmiş olsun. Bu durumda `B` `abstract` sınıftır. Fakat `C` sınıfı `abstract` sınıf değildir. Örneğin:

```
abstract class A
{
     public abstract void Foo();
     public abstract void Bar();
     //...
}

abstract class B : A
{
     public override void Foo()
     {
          //...
     }
}

class c : B
{
     public override void Bar()
     {
          //...
     }
}
```

Bir sınıfın hiçbir `abstract` elemanı olmadığı halde yine de sınıf bildiriminin başına `abstract` anahtar sözcüğü getirilerek, sınıf `abstract` yapılabilir. Bu durumda da `new` operatörü ile yine sınıf nesnesi yaratılamaz.

Abstract PROPERTY ELEMANLAR

Sınıfın property elemanları, aslında `get` ve `set` fonksiyonları olarak değerlendirilebilir. Property elemanın `abstract` olması, aslında bu `get` ve `set` bölümlerin `abstract` olması anlamındadır.

Bir property eleman `abstract` yapılırken ana bloğun içerisinde `get` ve `set` anahtar sözcükleri noktalı virgül ile kapatılır. Örneğin:

```
abstract public int x
{
     get;
     set;
```

```
}
```

Yine türemiş sınıfta bu property elemanlar override edilmelidir. Örneğin:

```
abstract class A
{
     abstract public int x
     {
          get;
          set;
     }
}
class B : A
{
     public override int x
     {
          get
          {
               //...
          }
          set
          {
               //...
          }
     }
}
```

HOCANIN TAM ÖRNEĞİ

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            A a = new B();

            a.X = 10;
        }
    }

    abstract class A
    {
        abstract public int X
        {
            get;
            set;
        }

        //...
```

```
    }

    class B : A
    {
        public override int X
        {
            get
            {
                Console.WriteLine("get");
                return 0;
            }
            set
            {
                Console.WriteLine("set");

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

ARAYÜZLER

Bir arayüz, `interface` anahtar sözcüğü ile arayüz ismi belirtilerek oluşturulur. Örneğin:

```
interface IX
{
     //...
}
```

Arayüz isimleri geleneksel olarak, `I` harfi ile başlanarak isimlendirilir. Bir arayüz, adeta tüm elemanları `abstract` olan ve veri elemanına sahip olmayan sınıflara benzemektedir. Arayüz oluşturmanın şu kısıtları vardır:

- **Arayüz elemanlarında erişim belirleyici (public) elemanlar kullanılamaz. Zaten default `public` durum anlaşılır.**
- **Arayüz elemanlarında `virtual`, `abstract` gibi belirteçler kullanılmaz. Elemanlar zaten `abstract` gibi ele alınır.**
- **Arayüz fonksiyonları ve propertyleri gövde içermez.**
- **Arayüzler veri elemanı içeremez. Örneğin:**

```
   interface IX
   {
     void Foo();
     int Bar(int a);
     int X
     {
          get;
          set;
     }
```

```
}
```

Arayüz türünden referanslar tanımlanabilir. Fakat new operatörü ile nesneler yaratılamaz. Örneğin:

```
IX ix;    à geçerli

ix = new IX();  à error
```

Bir arayüz bir sınıfın ya da yapının taban listesinde belirtilebilir. Bu duruma ilgili sınıf ya da yapının ilgili arayüzü desteklemesi denilmektedir. Bir sınıf tek bir sınıftan türetilebilir fakat birden fazla arayüzü destekleyebilir. Örneğin:

```
class A : B, IX, IY
{
     //...
}
```

Burada A sınıfı, B sınıfından türetilmiş ve IX ve IY arayüzünü desteklemektedir. Taban listesinde(:'den sonraki yazılanlar) arayüzler istenildiği sırada belirtilebilir. Fakat eğer taban listesinde hem taban sınıf hem de arayüzler varsa taban sınıfın önce belirtilmesi zorunludur.

Anımsanacağı gibi yapılar türetme işleminde kullanılamamaktadır. Fakat yapılar bir yada birden fazla arayüzü desteklemektedir. Örneğin:

```
stract A : IX, IX
{
     //...
}
```

Bir sınıf yada yapı bir arayüzü destekliyorsa o sınıf yada yapıda o arayüz elemanlarının public olarak tanımlanması zorunludur. Aksi halde işlem error ile sonuçlanır. Örneğin:

```
interface IX
{
     void Foo();
}
class A : IX
{
     public void Foo()
     {
          //...
     }
     //...
}
```

Burada A sınıfı, Foo() fonksiyonunu tanımlamazsa derleme zamanında error oluşur. Tanımlama sırasında override anahtar sözcüğü belirtilemez ve tanımlama public olarak yapılmak zorundadır.

Arayüzler çokbiçimli etkiye sahiptir. Bir arayüz referansına o arayüzü destekleyen sınıf referanslar atanabilir. Sonra bu arayüz referansı ile arayüz fonksiyonlar çağrılırsa, dinamik türe ilişkin sınıf ya da yapının fonksiyonları çağrılır. Örneğin:

```
IX ix = new A();

ix.Foo();  à A.Foo() çağrılır
```

HOCANIN TAM ÖRNEĞİ

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            IX ix;

            ix = new A();

            ix.Foo();

            IY iy = new A();
            iy.Bar();
        }
    }

    interface IX
    {
        void Foo();
    }

    interface IY
    {
        void Bar();
    }

    class A : IX, IY
    {
        public void Bar()
        {
            //...
        }
        public void Foo()
        {
            //...
        }
    }
}
```

Her arayüzün yine `System.Object` türünden türetildiği varsayılır.

`B` sınıfı, `A` sınıfından türetilmiş olsun. `A` sınıfı da örneğin `IX` isimli bir arayüzü destekliyor olsun.

```
interface IX
{
     void Foo();
}


class A : IX
{
     public void Foo()
     {
          //..
     }
}
class B : A
{
     //...
}
```

Burada `B` sınıfı `IX` arayüzünü desteklememektedir. `A` sınıfı desteklemektedir. Bu durumda `Foo()`, `B` sınıfında tanımlanmak zorunda değildir.

Türemiş sınıf bir arayüzü desteklemiyor fakat taban sınıf destekliyorsa yine biz türemiş sınıf türünden bir nesnenin referansını arayüz referansına atayabiliriz. Bu arayüz referansı ile arayüz fonksiyonu çağrılabilir. Örneğin:

```
IX ix = new B();

ix.Foo(); à A.Foo çağrılır.
```

Arayüz konusu çoklu türetme kavramını basit bir şekilde oldukça kısıtlı ölçüde dile sokmak için düşünülmüştür. Yani örneğin bir sınıf birden fazla taban sınıfa sahip olamaz. Fakat birden fazla arayüzü destekleyebilir. C++ da arayüz konusu yoktur. Bu dilde çoklu türetme olduğu için arayüz gereksinimi zaten `abstract` sınıflarla karşılanabilmektedir.

`IX` ve `IY` gibi iki arayüzde aynı isimli ve aynı parametrik yapıya sahip fonksiyonlar olabilir. Örneğin:

```
interface IX
{
     void Foo();
     //...
}

interface IY
{
     void Foo();
```

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

Bu iki arayüz destekleyen A gibi bir sınıf tek bir tanımlama ile her iki arayüzü destekleme gereksinimini de karşılar. Örneğin.

```
class A : IX, IY
{
    public void Foo()
    {
        //...
    }
    //...
}
```

Örneğin:

```
A a = new A();

IX ix = a;
ix.Foo(); à A.Foo çağrılır

IY iy = a;
iy.Foo(); à A.Foo çağrılır.
```

.Net'te pek çok standart arayüz de tanımlanmıştır. Gördüğümüz sınıfların çoğu bu arayüzleri desteklemektedir.

Bir arayüz başka bir arayüzden türetilebilir. Fakat bir sınıf ya da yapıdan türetilemez. Örneğin:

```
interface IX
{
    void Foo();
}

interface IX : IY
{
    void Bar();
}
```

Anahtar Notlar:

Destekleme tanımlama zorunluluğunu anlatan bir kavramdır. Yani bir sınıf yada yapı bir arayüzü desteklerse o arayüzün fonksiyonlarını tanımlamak zorundadır. Halbuki türetme türemişin taban elemanlarını içermesi anlamına gelmektedir.Bu nedenle arayüzün arayüzü desteklemesi yerine türetmesi sözkonusu olmaktadır.

Bir sınıf ya da yapı bir türemiş arayüzü destekliyorsa hem türemiş arayüzün fonksiyonlarını hem de taban arayüzün fonksiyonlarını tanımlamak zorundadır. Örneğin:

```
class A : IY
{
     public void Foo()
     {
          //...
     }
     public void Bar()
     {
          //...
     }
}
```

Burada A sınıfı, hem Foo() hem de Bar() fonksiyonlarını tanımlamak zorundadır.

Sınıf ya da yapının taban listesinde türemiş arayüzün yanı sıra taban arayüzde belirtilebilir. Taban arayüzün belirtilmesinin hiçbir gerekliliği ve özel anlamı yoktur. Bu belirtme geçerlidir. Fakat taban arayüzün hiç belirtilmemesi ile arasında hiçbir fark yoktur. Örneğin:

```
class A : IY
{
     //...
}
```

ile

```
class A : IY, IX
{
     //...
}
```

tamamen aynıdır.

Burada belirtme büyük ölçüde dikkat çekmek için yapılmaktadır.

Türemiş arayüz referansı taban arayüz referansına atanabilir. Fakat tersi olan durum yalnızca tür dönüştürme operatörü ile yapılır. Örneğin:

```
IY iy = new A();
IX ix iy;

ix.Foo(); à A.Foo çağrılır.
```

Anımsanacağı gibi bir arayüz fonksiyonu public olarak yazılmak zorundadır. Fakat arayüz fonksiyonlarının normal tanımlamanın yanısıra açıkça tanımlanması da mümkündür. Hem normal tanımlama hem de açıkça tanımlama aynı anda yapılabilir. Açıkça tanımlamada erişim belirleyici anahtar

sözcükler kullanılmaz ve arayüz fonksiyonu, arayüz ismi belirtilerek tanımlanır. Örneğin;

```
Interface IX
{
     void Func();
}

class Sample:IX
{
     public void Func ()      //Normal tanımlama
}
void IX.Func()                //Açıkça (explicit) tanımlama
{
     //..
}
```

Arayüz fonksiyonları yanlızca normal olarak tanımlanabilir, yanlızca açıkça tanımlanabilir ya da hem normal hem de açıkça tanımlanabilir.

Açıkça tanımlanmış olan versiyon, yanlızca arayüz referansıyla çağırılabilir. Normal versiyon ise hem nesne referansıyla hem de arayüz referansıyla çağırılabilir.

Arayüz fonksiyonunun hem normal hem de açıkça tanımlanmış olduğunu varsayalım. Şimdi biz bu fonksiyonu nesne referansıyla çağırırsak (yani aşağıdan çağırırsak) normal tanımlanmış olan versiyon, arayüz referansıyla çağırırsak açıkça tanımlanmış versiyon çağırılır.

Arayüz fonksiyonu yanlızca açıkça tanımlanabilir. Bu durumda arayüz fonksiyonu aşağıdan yani sınıf referansı kullanılarak çağırılamaz. Fakat yukarıdan çağırılabilir.

ARAYÜZ DÖNÜŞTÜRMELERİ

Eğer bir sınıf ya da yapı, bir arayüzü destekliyorsa o sınıf türünden referans ya da yapı türünden değişken, o arayüz referansına ya da o arayüzün taban arayüz referanslarına doğrudan atanabilir.

Eğer taban sınıf bir arayüzü destekliyorsa türemiş sınıfın da o arayüzü desteklediği kabul edilmektedir. Bu durumda örneğin taban sınıf bir arayüzü destekliyor olsun fakat türemiş sınıf taban listesinde o arayüzü belirtmesin. Bu durumda türemiş sınıfın da yine o arayüzü desteklediği kabul edilir ve türemiş sınıf türünden referans, o arayüz referansına doğrudan atanabilir.

```
interface IX
{
     void Func();
}

class A:IX
```

```
{
     //..
}

class B:A
{
     //..
}

//..

B b=new B();

IX ix=b;   //geçerli
ix.Func(); //A'daki func çağırılır
```

Taban sınıf bir arayüzü destekliyor olsun arayüz türemiş sınıfın taban listesinde yeniden belirtilirse türemiş sınıf bu arayüzü ayrıca yeniden destekleyebilir. Fakat bu durumda türemiş sınıf arayüz fonksiyonunu yeniden tanımlamalıdır. Örneğin ;

```
interface IX
{
     void Func();
}

class A:IX
{
     //..
}

class B:A,IX
{
     //..
}

//..

B b=new B();

IX ix=b;   //geçerli
ix.Func(); //B'deki func çağırılır
```

Arayüzlerden sınıflara yani yukarıdan aşağıya doğru da tür dönüştürme operatörü ile dönüştürme yapılabilir. Tabi bu dönüştürmenin derleme aşamasında kabul edilebilmesi için dönüştürülen sınıfın ya da yapının o arayüzü destekliyor olması gerekir.Tabi ayrıca yine denetim çalışma zamanı sırasında dönüşümün haklı olup olmadığını belirlemek için yapılacaktır. Örneğin:

```
interface IX
{
     //...
```

```
}
class A :IX
{
     //...
}
class B : A
{
     //...
}
B b = new B();
IX ix = b; à geçerli
A a = (A) ix;  à geçerli
```

Bir sınıf referansı tür dönüştürme operatörü ile bir arayüze dönüştürülebilir. Bu dönüştürme her zaman derleme aşamasında kabul edilir. Fakat programın çalışma zamanı sırasında sınıf referansının dinamik türünün, o arayüzü destekleyip desteklemediğine bakılır ve böylece denetim uygulanır. Bu tür durumlarla şöyle bir senaryo biçiminde karşılaşılabilir:

A isimli sınıf, IX arayüzünü destekliyor olsun. Biz A türünden nesne referansını, Object türünden bir referansa atamış olalım. Şimdi bu object referansını, tür dönüştürme operatörü ile IX türüne dönüştürebiliriz. Çünkü object referansının dinamik türü A dır. A da IX arayüzünü desteklemektedir. Örneğin:

```
interface IX
{
     //...
}
class A : IX
{
     //...
}
A a = new A();
Object o = a;
IX ix = (IX)o à geçerli
```

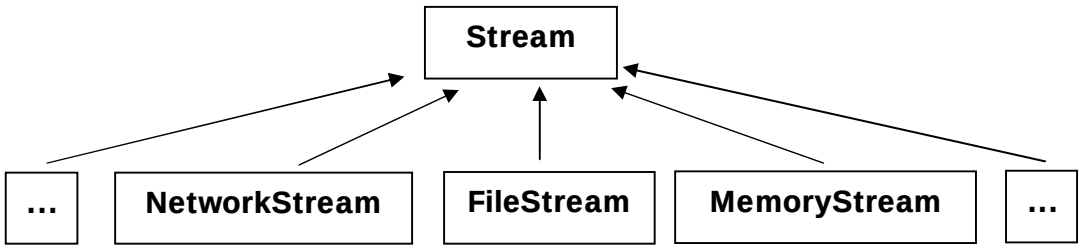
Özetle bir sınıf referansı bir arayüz referansına tür dönüştürme operatörü ile dönüştürülmek istenirse, bu durum her zaman derleme aşamasında kabul edilir. Fakat çalışma zamanı sırasında, CLR referansın dinamik türüne ilişkin sınıfın, bu arayüzü destekleyip desteklemediğine bakar.

DOSYA İŞLEMLERİ

Dosya işlemleri System.IO isim alanındaki çeşitli sınıflarla yapılmaktadır. .Net ortamında dosya gibi işlem gören tüm kavramlar, çokbiçimli bir etki yaratmak için ortak bir biçimde ele alınmıştır. Her ne kadar dosya denince

akla tipik olarak diskteki dosyalar anlaşılsa da soketteki bilgiler, bellekteki bilgilerde çokbiçimli etki yaratmak için dosya gibi değerlendirilmiştir.

Dosya işlemlerinde dosyamsı olan kavramlar `Stream` sınıfı ile temsil edilmiştir. `Stream` sınıfı `abstract` bir sınıftır ve dosya gibi işlem gören farklı kaynaklara ortak taban sınıflık yapmaktadır. `Stream` sınıfından bazı sınıflar türetilmiştir. Şüphesiz bu sınıflar `abstract` olmadığı için, `Stream` sınıfının `abstract` fonksiyonlarını `override` etmiştir.



`FileStream` sınıfı, disk tabanlı gerçek dosyaları temsil etmektedir. `NetworkStream` sınıfı, bir soketi dosya gibi temsil etmektedir. `MemoryStream` sınıfı, bellekteki bir bilginin dosya gibi ele alınmasını sağlamaktadır.

Yukarıdaki türetme şeması, çokbiçimli kullanıma olanak vermektedir. Yani biz dosyanın ne tabanlı bir dosya olduğunu bilmeden, onunla işlem yapabiliriz. Örneğin; bir kaynaktan bilgi okuyarak onları ayrıştıran bir `Parser` sınıfı yazacak olalım. `Parser` sınıfı, hiçbir değişiklik yapılmadan disk tabanlı dosyayı da, bir soketi de, bellekteki bir bilgiyi de kaynak olarak kullanıp parse yapabilsin. O halde `Parser` sınıfı bir `Stream` referansı alarak, `Stream` sınıfının sanal fonksiyonlarını çağırarak işlemlerini yapmaktadır.

```
class Parser
{
     private Stream m_s;

     public Parser (Stream s)
     {
          m_s = s;
     }
}
```

Biz sınıfı şöyle kullanabiliriz:

```
FileStream fs = new fileStream(...);
Parser p = new Parser(fs);
```

Şimdi `Parser` sınıfı bir disk dosyasındaki bilgileri parse edecektir. Ya da örneğin:

```
MemoryStream ms = new MemoryStream(...);
Parser p = new Parser(ms);
```

DİSK TABANLI DOSYALARIN FILESTREAM SINIFI İLE KULLANILMASI

Bir disk dosyası üzerinde işlem yapabilmek için, öncelikle onu açmamız gerekir. Dosyanın açılması sırasında bir takım ilk işlemler yapılır. `FileStream` sınıfının başlangıç fonksiyonları dosyayı açmaktadır. `FileStream` sınıfının birçok başlangıç fonksiyonu vardır. Ama açış için en çok kullanılan fonksiyon aşağıdaki gibidir:

```
public FileStream (string path, FileMode mode, FileAccess access);
```

Anahtar Notlar

Bir dosya herhangi bir sürücünün herhangi bir dizininde bulunabilir. Farklı dizinlerde aynı isimli dosyalar bulunabilir. O halde bir dosyayı belirtirken onun hangi dizinde olduğunu da belirtmemiz gerekebilir. Bir dosyanın yerini belirten yazısal ifadeye yol ifadesi(path) denir. Yol ifadesi mutlak ve göreli olmak üzere 2 ayrılır. Eğer yol ifadesinin ilk karakteri \ ile başlıyorsa mutlak yol ifadesi söz konusudur. Mutlak yol ifadesi kök dizinden itibaren yol belirtir. Eğer ilk karakter \ değilse buna göreli yol ifadesi denir. Göreli yol ifadesi prosesin çalışma dizininden itibaren yol belirtmektedir. Bu dizin default durumda .exe dosyanın bulunduğu dizindir. Tek başına bir dosya ismi göreli bir yol ifadesidir. O halde bu dosya exe dosyanın bulunduğu dizinde aranacaktır.

Fonksiyonun birinci parametresi, dosyanın yol ifadesini belirtmektedir. İkinci parametre, dosyanın açım modunu belirtmek için kullanılır. `FileMode` bir enum türüdür ve şu elemanlara sahiptir:

`Open`: Var olan dosyanın açılması için kullanılır. Eğer dosya yoksa exception oluşur.
`Create`: Dosya yoksa yaratılır ve açılır, varsa sıfırlanır ve açılır.
`OpenOrCreate`: Dosya varsa varolan dosyayı açar , yoksa dosyayı yaratır ve açar.
`CreateNew`: Dosya yoksa yaratır ve açar, dosya varsa exception oluşur dosyayı bozmaz.
`Trancute`: olan bir dosyayı sıfırlayarak açmak için kullanılır. Dosya yoksa exception oluşmaktadır.
`Append`: olan dosyayı açar yoksa dosyayı yaratır açar. Dosyanın herhangi bir yerinden okuma yapılabilir. Fakat tüm yazma işlemleri sona ekleme anlamındadır.

Fonksiyonun üçüncü parametresi, `FileAccess` isimli bir enum türündendir. Bu parametre dosyaya yapılacak operasyonu belirtir. Bu enum türünün üç elemanı vardır:

`Read`: Yalnızca okuma yapılabilir
`Write`: Yalnızca yazma yapılabilir.
`ReadWrite`: Hem okuma hem yazma yapılabilir.

Eğer dosya açılmazsa işlem exception oluşur.

Stream SINIFININ ÖNEMLİ ELEMANLARI

Sınıfın Length isimli abstract property elemanı, açılmış olan dosyanın uzunluğunun verir.

Sınıfın Read isimli abstract fonksiyonu, dosyadan n byte okumaktadır.

```
public abstract int Read (byte[] buffer, int offset, int count)
```

Fonksiyonun birinci parametresi, okunan byte'ların yerleştirileceği diziyi belirtir. İkinic parametresi, bu dizide bir offset belirtmektedir. Yani okunacak bilgilerin dizinin hangi indexinden itibaren yerleştirileceğidir. Bu parametre tipik olarak "0" biçiminde girilir. Son parametre okunacak byte sayısıdır.

Sınıfın Write isimli abstract fonksiyonu, ters olarak bir byte dizisi içerisindekileri dosyaya yazar.

```
public abstract void Write(byte[] buffer, int offset, int count)
```

Fonksiyonun birinci parametresi, yazdırılacak olan bilginin bulunduğu diziyi, ikinci parametresi dizide bir indexi(yazdırma bu indexten itibaren yapılacaktır) ve son parametresi de yazılacak byte sayısı belirtmektedir.

Ayrıca Stream sınıfının 1 byte yazan ve okuyan ReadByte ve WriteByte fonksiyonları da vardır:

```
public virtual int ReadByte()
```

```
public virtual void WriteByte (byte value)
```

DOSYA GÖSTERİCİSİ KAVRAMI

Dosya göstericisi, o anda dosyanın hangi offsetinden okuma ya da yazma yapılacağını belirten bir sayıdır. İlk byte "0" olmak üzere, dosyanın her bir byte'na bir offset numarası karşılık getirilmiştir. Okuma ve yazma işlemleri, o offset numarasından itibaren yapılır. Dosya göstericisinin konumu, otomatik olarak okunan ya da yazılan byte miktarı kadar ilerletilmektedir. Dosya göstericisinin, dosyadaki son byte'dan bir sonraki byte'ı göstermesi durumuna(yani olmayan byteyi göstermesi durumu) EOF denir. EOF durumunda okuma yapılamaz. Ama yazma yapılabilir(yazma hakkımız varsa). Yazma ekleme anlamına gelir.

Dosya göstericisinin konumlandırılması için Stream sınıfının Position property elemanı kullanılabilir. Bu property long türdendir ve Read Write propertydir.

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream fs = new FileStream("test.txt",
FileMode.Open, FileAccess.Read);

            fs.Position = 10;

            byte[] b = new byte[5];

            fs.Read(b, 0, 5);

            string s = Encoding.ASCII.GetString(b);

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

DOSYANIN KAPATILMASI

Dosyayı kapatmak için `Stream` sınıfının parametresiz `Close` fonksiyonu kullanılır. Açılan her dosya kapatılmalıdır.

BYTE DİZİSİNDEN DÖNÜŞTÜRME YAPAN FONKSİYONLAR

Elimizde bir `byte` dizisi varsa onun içerisindeki bilgileri istediğimiz türe dönüştürebiliriz. Bunun için `BitConverter` sınıfının `ToXXX(xxx=int,char vs...)` fonksiyonları kullanılır. Bu fonksiyon parametre olarak `byte` dizisini alırlar. Geri dönüş değeri olarak dönüştürülmüş değeri verirler. Bu işlemin tam tersi de yapılmak istenebilir. Yani örneğin elimizde bir `int` değer bulunabilir. Biz bunu `4` `byte` halinde bir `byte` dizisine yazmak isteyebiliriz. Bunun için `BitConverter` sınıfının `GetBytes` fonksiyonları kullanılmaktadır. Bu fonksiyonlar, parametre olarak byte'a dönüştürülecek tür türünden değer alırlar. Kendi içlerinde bir byte dizisi yaratırlar ve bu byte dizisi ile geri dönerler.

Anahtar Notlar

Elimizde int türden bir sayı olsun. Biz bu sayıyı bir dosyaya yazı olarak yada binary bir biçimde 4 byte lik bir dizi olarak yazabiliriz. Yazı olarak yazarsak bir editörle kolay bir biçimde görürüz. Fakat yazı olarak yazmak tercih

edilmeyebilir. Biz onu binary olarak yazarsak her zaman 4 byte olarak yer kaplayacaktır.

Örneğin bir dosyaya bir sayıyı yazıp okuyan program şöyle yazılabilir:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{

    class App
    {
        public static void Main()
        {
            FileStream fs = new FileStream("test.txt",
FileMode.Create, FileAccess.ReadWrite);

            int i = 123456;

            byte[] b = BitConverter.GetBytes(i);//int ten byte
dönüştürme

            fs.Write(b, 0, 4); // 4 byte yazdı
            fs.Position = 0; //dosya göstericisi tekrar başa alındı

            byte[] x = new byte[4];
            fs.Read(x, 0, 4); // x dizisine 4 byte bilgi okundu

            int val = BitConverter.ToInt32(x, 0);//byte int te
dönüştürüldü

            Console.WriteLine(val);

            fs.Close();
        }
    }
}
```

Bir byte dizisini string'e, bir string'i byte dizisine dönüştürmek için, yazının hangi formata ilişkin olduğunun bilinmesi gerekir. Yazı her bir karakteri 1 byte olan ASCII karakterlerden oluşabilir. Unicode UTF8 formatta ya da başka bir formatta da olabilir. O halde konu bir Encoding konusudur. İşte bu işlem için Encoding sınıfının Encoding türünden ASCII, UTF8 gibi elemanları kullanılır. Daha sonra dönüştürme işlemi bu sınıfın GetBytes ve GetString fonksiyonları ile yapılır. Örneğin:

```
using System;
using System.Collections;
using System.IO;
using System.Text;
```

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream fs = new FileStream("test.txt",
FileMode.Open, FileAccess.Read);
            byte[] b = new byte[10];

            fs.Read(b, 0, 10);

            string s = Encoding.ASCII.GetString(b);

            Console.WriteLine(s);

            fs.Close();
        }
    }
}
```

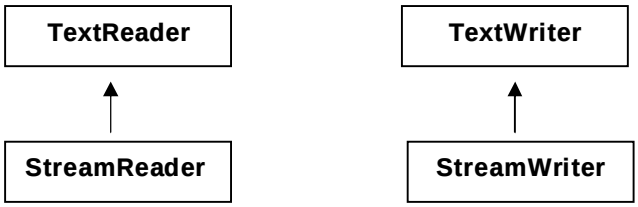
Görüldüğü gibi FileStream sınıfı ile byte dizisi okuyup byte dizisi yazabiliriz. Yani bu sınıf bize çok rahat bir çalışma olanağı sunmaz.

DOSYA İŞLEMLERİNİ KOLAYLAŞTIRAN YARDIMCI SINIFLAR

Aslında her türlü dosya işlemi byte okuyup yazan Read ve Write fonksiyonları ile yapılabilir. Fakat programcının bu fonksiyonları kullanarak fazlaca kod yazması gerekir. Halbuki bir takım adaptör sınıflar bir Stream referansı alarak çokbiçimli bir biçimde Read ve Write fonksiyonlarını kullanarak bu işlemleri yapabilmektedir.

-StreamReader ve StreamWriter Sınıfları

StreamReader ve StreamWriter sınıfları, bir Stream nesnesini alarak yetenekli text düzeyinde okuma ve yazma işlemlerini yapar. Bu sınıflar TextReader ve TextWriter sınıflarından türetilmiştir.



Bir StreamReader nesnesi aşağıdaki başlangıç fonksiyonları ile oluşturulabilir:

```
public StreamReader (Stream stream)
```

Bu durumda nesne şöyle oluşturulabilir:

```
FileStream fs = new FileStream("test.txt", FileMode.Open,
FileAccess.Read);
```

```
StreamReader sr = new StreamReader(fs);
```

StreamReader ve StreamWriter sınıflarının Close fonksiyonları, kullandıkları gerçek Stream kaynağını da kapatmaktadır:

```
sr.Close();
```

Bu işlemle kullanılan FileStream nesnesi de kapatılır.

StreamReader sınıfının string geri dönüş değerine sahip ReadLine isimli fonksiyonu dosya göstericisinin gösterdiği yerden itibaren bir satırlık bilgiyi okur. Eğer dosya sonuna gelinirse fonksiyon null değeri geri döner. Bu durumda bir text dosyayı satır satır okuyup ekrana yazdıran program şöyle olabilir:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{

    class App
    {
        public static void Main()
        {
            FileStream fs = new FileStream("test.txt",
FileMode.Open, FileAccess.Read);
            StreamReader sr = new StreamReader(fs);

            string line;

            while ((line = sr.ReadLine()) != null)
                Console.WriteLine(line);

            sr.Close();
        }
    }
}
```

StreamReader sınıfının ReadToEnd isimli fonksiyonu dosya göstericisinin gösterdiği yerden itibaren dosya sonuna kadar tüm karakterleri okur ve bir string biçiminde geri verir:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
```

```
{
    class App
    {
        public static void Main()
        {
            FileStream fs = new FileStream("test.txt",
FileMode.Open, FileAccess.Read);
            StreamReader sr = new StreamReader(fs);

            string s = sr.ReadToEnd();

            Console.WriteLine(s);

            sr.Close();
        }
    }
}
```

StreamReader default olarak dosyadaki karaterlerin Unicode UTF8 kodlamasına ilişkin olduğunu varsaymaktadır. Tabi bir yazıdaki karakterler hep ingilizce karakterlerden oluşursa Unicode UTF8 kodlaması ile ASCII kodlaması arasında bir fark kalmaz. İstenirse StreamReader ve StreanmWriter sınıflarının iki parametreli başlangıç fonksiyonları ile encoding belirtilebilir:

```
StreamReader sr = new StreamReader(fs, Encoding.ASCII);
```

StreamWriter sınıfı benzer amaçla kullanılmaktadır. StreamWriter sınıfının her türden parametreli Write ve WriteLine fonksiyonları vardır. Bu fonksiyonlar, sayısal değerleri yazı biçiminde dosyaya yazar. Örneğin:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream fs = new FileStream("test.txt",
FileMode.Create, FileAccess.ReadWrite);
            StreamWriter sw = new StreamWriter(fs);

            for (int i = 0; i < 100; ++i)
                sw.WriteLine(i);

            sw.Close();
        }
    }
}
```

Şüphesiz string parametreli Write ve WriteLine fonksiyonları da vardır. Örneğin:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream fs = new FileStream("test.txt",
FileMode.Create, FileAccess.ReadWrite);
            StreamWriter sw = new StreamWriter(fs);

            sw.WriteLine("Ankara");
            sw.WriteLine("Izmir");

            sw.Close();
        }
    }
}
```

BinaryReader ve BinaryWriter SINIFLARI

Bu sınıflar çeşitli sayısal değerleri dosyaya yazı olarak değil binary biçimde bir byte dizisi olarak yazıp okurlar. Bu sınıflar da bir Stream referansı alarak aslında Stream sınıfının Read ve Write fonksiyonlarını kullanır. Benzer biçimde bu fonksiyonların Close fonksiyonları da aldıkları string nesnesini kapatmaktadır. Örneğin; BinaryReader nesnesi şöyle yaratılabilir:

```
FileStream fs = new FileStream(...);
BinaryReader br = new BinaryReader();
```

BinaryReader sınıfının ReadXXX biçiminde parametresiz fakat geri dönüş değeri XXX türünden olan bir grup fonksiyonu vardır. Bu fonksiyonlar dosyadaki byte topluluğunu okuyup, onu XXX türüne dönüştürerek vermektedir. Benzer biçimde BinaryWriter sınıfının da her türden parametre alan Write fonksiyonları vardır. Bunlar tıpkı StreamWriter gibi yazma yaparlar. Fakat yazmayı yazı olarak değil Binary düzeyde gerçekleştiriler. Örneğin:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
```

```
{
    class App
    {
        public static void Main()
        {
            FileStream    fs    =    new    FileStream("test.dat",
FileMode.Open, FileAccess.ReadWrite);
            BinaryReader br = new BinaryReader(fs);

            int val = br.ReadInt32();

            Console.WriteLine(val);

            br.Close();
        }
    }
}
```

Bazen bir sınıfın tüm elemanlarını dosyaya yazıp okumak gerekebilir. Bu işlemi seri hale getirmek dışında, pratik olarak yapmanın bir yolu yoktur. Tipik olarak sınıfın içerisine, Read ve Write gibi iki fonksiyon yerleştirilebilir Bu fonksiyonlar okuma ve yazmayı yapabilirler.

Hocanın Örneği:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream    fs    =    new    FileStream("test.dat",
FileMode.Create, FileAccess.ReadWrite);
            BinaryWriter bw = new BinaryWriter(fs);
            BinaryReader br = new BinaryReader(fs);

            Person a = new Person("Kaan Aslan", 123);
            a.Write(bw);

            fs.Position = 0;

            Person b = new Person();
            b.Read(br);

            b.Disp();

            bw.Close();
            br.Close();
        }
    }
```

```
    class Person
    {
        private string m_name;
        private int m_no;

        public Person()
        { }

        public Person(string name, int no)
        {
            m_name = name;
            m_no = no;
        }
        public void Disp()
        {
            Console.WriteLine("{0} {1}", m_name, m_no);
        }
        public void Write(BinaryWriter bw)
        {
            bw.Write(m_name);
            bw.Write(m_no);
        }
        public void Read(BinaryReader br)
        {
            m_name = br.ReadString();
            m_no = br.ReadInt32();
        }
    }
}
```

AÇMADAN YAPILAN DOSYA İŞLEMLERİ

System.IO isim alanındaki File isimli sınıfın pek çok static fonksiyonu vardır. Bu static fonksiyonlar dosya silme, isim değiştirme, taşıma, kopyalama gibi temel işlemleri yapmaktadır. Ayrıca File sınıfının static Create, Open gibi fonksiyonları da vardır. Bunlar FileStream sınıfını kullanarak dosya yaratımı yapmaktadır.

Exception İŞLEMLERİ

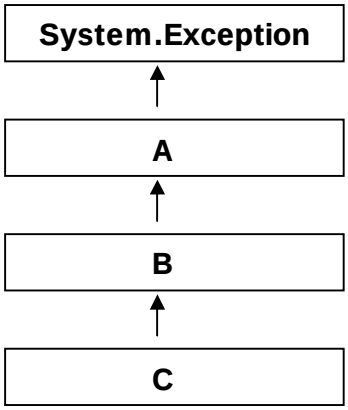
Exception terimi, beklemedik bir biçimde programın çalışma zamanı sırasında bir hata durumu ortaya çıkmasıdır. .Net ortamında sınıfların fonksiyonları, sorunlarla karşılaşınca exception oluşturmaktadır. Eğer bu exception ele alınmazsa, programın çalışmasın sonlanır. Exception işlemleri programın çeşitli yerlerinde oluşan hataların, belirli yerlerde kontrolünü sağlamak için kullanılır. Yani programcı hata nerede çıkmış olursa olsun, akışın belirli bir noktaya çekilip kontrolün orada yapılmasını sağlayabilir. Exception işlemleri için try, catch, finally, throw anahtar sözcükleri kullanılır.

Try anahtar sözcüğünü, bir blok izlemek zorundadır. Buna try bloğu denir. Try bloğunu bir ya da birden fazla catch bloğu izler. Catch anahtar sözcüğünden sonra parantezler içerisine, catch parametre bildirimi yapılır. Catch parametresi ilişkin tür bilgisi yazılmak zorundadır. Fakat catch parametre değişkeni yazılmak zorunda değildir.

```
try
{
     //...
}
catch(<tür>[parametre değişkeni])
{
     //...
}
catch(<tür>[parametre değişkeni])
{
     //...
}
```

Programın akışı try bloğuna girdikten sonra, çıkana kadar akış bakımından try bloğunun içerisindedir.

System.Exception sınıfı ya da bu sınıftan doğrudan ya da dolaylı olarak türetilen sınıflara, exception sınıflar denir. Örneğin:



Exception mekanizmasını tetikleyen, throw anahtar sözcüğüdür. Throw anahtar sözcüğünün genel biçimi şöyledir:

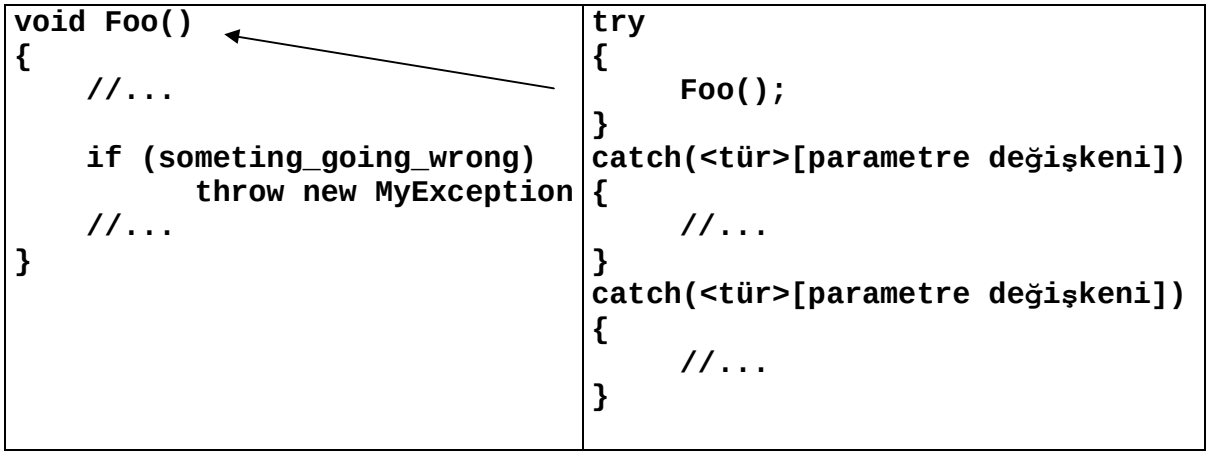
```
throw [exception sınıf referansı];
```

Örneğin; MyException isimli sınıf, Exception sınıfından türetilmiş olsun:

```
throw new MyException();
```

Programın akışı try bloğuna girdikten sonra, herhangi bir yerde throw işlemi oluşursa akış, bir goto işlemi gibi son girilen try bloğunun uygun catch bloğuna aktarılır. O catch bloğu çalıştırıldıktan sonra, diğer catch blokları

atlanarak akış, catch bloklarından sonra devam eder. Try bloğu ile catch blokları arasına başka bir deyim yazılamaz.



```
void Foo()
{
    //...

    if (someting_going_wrong)
          throw new MyException
    //...
}
```

```
try
{
     Foo();
}
catch(<tür>[parametre değişkeni])
{
     //...
}
catch(<tür>[parametre değişkeni])
{
     //...
}
```

Burada Foo() fonksiyonu içerisinde throw işlemi gerçekleşirse, akış goto işlemi gibi MyException parametreli catch bloğuna aktarılır.

.Net kütüphanesindeki sınıfların fonksiyonları sorunla karşılaşıldığında, hep exception sınıflarıyla throw işlemi yapılmaktadır. Bir fonksiyonun hangi durumda hangi sınıf ile throw yapacağı, MSDN dökümanlarında exception alt başlığında listelenmektedir. O halde bir kütüphane fonksiyonu çağrılırken hatayı ele alabilmek için çağrımı try-catch bloğu içerisinde yapmak gerekir.

Hocanın Örneği:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

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

            try
            {
                val = int.Parse(Console.ReadLine());

                Console.WriteLine(val);
            }
            catch (ArgumentNullException ane)
            {
                Console.WriteLine("Arguman null değerinde!..");
            }
            catch (FormatException fe)
            {
```

```
                Console.WriteLine("Bilgi  sayısal  karakterlerden
oluşmuyor!..");
            }
            catch (OverflowException of)
            {
                Console.WriteLine("Sayı sınır dışında!..");
            }
        }
    }

        class MyException : Exception
    {
        //...
    }

    class YourException : Exception
    {
        //...
    }
}
```

Oluşan bir exception için uygun bir catch bulunamazsa thread sonlandırılır. Bu durum tek thread'li programlarda, programın sonlandırılacağı anlamına gelir.

Programın akışı iç içe birden fazla try bloğuna girmiş olsun. İç bloklarda bir throw işlemi olursa, içten dışa doğru uygun catch blokları taranır. Yani önce en içteki try bloğunun catch bloklarına bakılır. Orada uygun catch bloğu bulunamazsa bir önce girilen try bloğunun catch bloklarına bakılır. Bu biçimde en yukarıya kadar çıkılır. Nihayet en dışta da uygun catch bloğu da bulunamazsa thread sonlandırılır.

Hocanın Örneği:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                Foo();
                //...
            }

            catch (OverflowException of)
            {
```

```
                Console.WriteLine("Sayı sınır dışında!..");
            }
        }

        public static void Foo()
        {
            try
            {
                int val = int.Parse(Console.ReadLine());

                Console.WriteLine(val);
            }
            catch (ArgumentNullException ane)
            {
                Console.WriteLine("Arguman null değerinde!..");
            }
            catch (FormatException fe)
            {
                Console.WriteLine("Bilgi  sayısal  karakterlerden
oluşmuyor!..");
            }
        }
    }

        class MyException : Exception
    {
        //...
    }

    class YourException : Exception
    {
        //...
    }
}
```

Nasıl türemiş sınıf referansı, taban sınıf referansına atanabiliyorsa, türemiş sınıf türünden `throw` işlemi yapıldığında, bu `throw` işlemi taban sınıf türünden bir `catch` bloğu ile yakalanabilir. Yani örneğin biz oluşacak tüm exceptionları aslında, exception parametreli bir `catch` bloğu ile yakalayabiliriz. .Net kütüphanesindeki `Exception` sınıfları, bir türetme ağacına sahiptir. Örneğin; dosya işlemleri sırasında fırlatılan tüm `Exception` sınıfları `System.IO` sınıfından türetilmiştir. `System.IO` sınıfı da dolaylı olarak `System.Exception` sınıfından türetilmiştir.

Taban sınıf ve türemiş sınıfa ait `catch` blokları bir arada kullanılabilir. Bu durumda türemiş sınıf türünden exception fırlatıldığında, bunu türemiş sınıf `catch` bloğu yakalayacaktır. Fakat derleyici, catch bloklarını sırasıyla taradığı için ve ilk uygun `catch` bloğu bulunduğunda işlemi sonlandırdığı için sıralama önemlidir. Türemiş sınıfa ilişkin `catch` bloğunun taban sınıfa ait `catch` bloğunun daha yukarısında oluşturulması zorunludur. Aksi durumda derleme zamanında `error` oluşur.

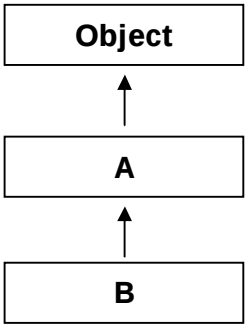
Hocanın Örneği:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

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

            try
            {
                int val = int.Parse(Console.ReadLine());

                string s = null;

                int n = s.Length;

                Console.WriteLine(val);
            }

            catch (ArgumentNullException ane)
            {
                Console.WriteLine("Arguman null değerinde!..");
            }
            catch (FormatException fe)
            {
                Console.WriteLine("Bilgi  sayısal  karakterlerden
oluşmuyor!..");
            }
            catch (OverflowException of)
            {
                Console.WriteLine("Sayı sınır dışında!..");
            }

            catch (Exception e)
            {
                Console.WriteLine("diğer...");
            }
        }
    }
    class MyException : Exception
    {
        //...
    }
    class YourException : Exception
    {
        //...
    }
}
```

SIK KARŞILAŞILAN exception SINIFLARI

Bir referansa, henüz değer atamadan referans kullanılamaz. Fakat referansa `null` değer atayıp referans kullanılırsa, program başarılı olarak derlenir. ancak çalışma zamanı sırasında, akış kullanım noktasına geldiğinde `NullReferanceException` oluşur.

Anımsanacağı gibi aralarında türetme ilişkisi olmayan sınıflar arasında, tür dönüştürme operatörü ile bile dönüştürme yapılamaz. Fakat türetme ilişkisi olduğu durumda derleme aşaması başarılı bir biçimde sonlanır. Fakat programın çalışma zamanı sırasında haklılık kontrolü yapılır. İşte dönüştürme haksız ise `InvalidCastException` isimli exception fırlatılmaktadır. Örneğin:




```
object o = new A();
B b = (B)o;  à  InvalidCastException
```

finally BLOĞU

`finally` bloğu parametre içermez. Öneğin:

```
finally
{
     //...
}
```

`finally` bloğu kullanılacaksa, `catch` bloklarının sonunda olmak zorundadır. Örneğin:

```
try
{
     //...
}
catch(Exception e)
{
     //...
}
finally
{
     //...
```

```
}
```

try bloğundan sonra, hiç catch bloğu bulunmadan finally bloğu bulunabilir. Örneğin:

```
try
{
     //...
}
finally
{
     //...
}
```

finally bloğu her zaman çalıştırılır. Akışın try bloğuna girdiğini düşünelim. Fakat hiç exception oluşmamış olsun. Bu durumda finally bloğu çalıştırıldıktan sonra akış devam eder.

Hocanın Örneği:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                Console.WriteLine("try block");
            }
            catch (Exception e)
            {
                Console.WriteLine("diğer...");
            }
            finally
            {
                Console.WriteLine("finally");
            }

            Console.WriteLine("ends...");
        }
    }

    class MyException : Exception
    {
        //...
    }

    class YourException : Exception
```

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

Try bloğuna girdikten sonra bir exception oluşsun ve oluşan exceptionun bir catch bloğu tarafından yakalandığını düşünelim. Catch bloğu çalıştırıldıktan sonra finally bloğu da çalıştırılır.

Hocanın Örneği:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                int.Parse("ggjhgj");
            }
            catch (Exception e)
            {
                Console.WriteLine("diğer...");
            }
            finally
            {
                Console.WriteLine("finally");
            }

            Console.WriteLine("ends...");
        }
    }

    class MyException : Exception
    {
        //...
    }

    class YourException : Exception
    {
        //...
    }
}
```

finally bölümün catch bölümlerinden sonra yazılanlardan ne farkı vardır? Yani madem ki finally bloğu her zaman çalıştırılacaktır burada yapılacakları catch bloklarından sonraya yerleştiremez miyiz? İç try bloğunda bir exception oluşmuş olsun. Fakat bu exception iç try bloğunun catch

blokları tarafından yakalanmamış olsun. İşte bu durumda dış `try` blokların catch blokları taranmadan önce iç `try` bloğunun `finally` bloğu çalıştırılır.

Hocanın Örneği:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                Func();
            }

            catch (Exception e)
            {
                Console.WriteLine("Dış catch");
            }

            Console.WriteLine("ends...");
        }

        public static void Func()
        {
            try
            {
                int.Parse("ggjhgj");
            }
            finally
            {
                Console.WriteLine("iç finally");
            }
        }
    }

    class MyException : Exception
    {
        //...
    }

    class YourException : Exception
    {
        //...
    }
}
```

Finally bölümü tipik olarak exception oluştuğunda, bir takım işlemleri geri almak için gerçekleştirilir. Örneğin bir dosyanın kapatılması işlemi tipik olarak finally bölümünde yapılmalıdır.

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream fs = null;

            try
            {
                fs = new FileStream("test.dat", FileMode.Open,
FileAccess.Read);
                //..
            }
            catch (Exception e)
            {
                //...
            }
            finally
            {
                if (fs != null)
                    fs.Close();
            }
        }
    }
}
```

Burada try bloğunun içerisinde bir exceptoion oluşsa akış, dış catch bloklarına gitse bile finally bloğu çalıştırılacağı için dosya kapatılır. Tabi exception başlangıç fonksiyonu içerisinde de oluşabilirdi. Bu durumda, fs'nin içerisinde null değeri olduğu için açılmamış doya kapatılamayacaktır.

Anımsanacağı gibi bir değişkene değer atanmadan değişken kullanılamaz. Aşağıdaki durumda derleme zamanında error oluşur:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
```

```
        public static void Main()
        {
            FileStream fs;

            try
            {
                fs  =  new  FileStream("test.dat",  FileMode.Open,
FileAccess.Read);
                //..
            }
            catch (Exception e)
            {
                //...
            }
            finally
            {
                fs.Close();
            }
        }
    }
}
```

Burada FileStream sınıfının başlangıç fonksiyonu içerisinde exception oluştuğunu varsayalım. Bu durunda fs değer almamış olacaktır. Bu da error ile sonuçlanacaktır. Özetle finally bölümüne, exception oluşsa da oluşmasa da her zaman yapılması gereken şeyler yerleştirilmelidir.

Try bloğunda goto, return gibi akışı başka bir bölgeye aktaran deyimler kullanılsa bile yine finally bölümü çalıştırılır.

Hocanın Örneği:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                int val = int.Parse(Console.ReadLine());
                if (val < 0)
                    return;
            }

            finally
            {
                Console.WriteLine("finally");
            }
```

```
        EXIT:
            Console.WriteLine("The end...");
        }
    }
}
```

PARAMETRESİZ catch BLOĞU

Parametresi olmayan özel bir catch bloğu ancak diğer catch bloklarının en sonuna fakat finally bloğundan önce yerleştirilebilir. Örneğin:

```
try
{
     //...
}
catch(Exception e)
{
     //..
}
catch
{
     //...
}
finally
{
     //...
}
```

Parametresiz catch bloğu tüm exceptionları yakalar. Peki exception parametreli catch bloğu da tüm exceptionları yakalayamaz mı?

Exception sınıflarının System.exception sınıflarından türetilme zorunluluğu .Net geneline özgü değil, yalnızca C#'a özgüdür. Örneğin biz C++.Net'te herhangi bir sınıf ile throw işlemi yapabiliriz. İşte biz C++.Net'te yazılmış ve böyle bir sınıf ile throw eden bir fonksiyonu C#'tan çağırdığımızda bu exception, exception parametreli catch bloğu tarafından yakalanamaz. Ancak parametresiz catch bloğu tarafından yakalanır. Yani parametresiz catch bloğu, exception parametreli catch bloğunu kapsamaktadır.

YENİDEN throw İŞLEMİ

Oluşan bir exceptionu catch bloğunda yakalamış olalım. Eğer yakalamamış olsaydık, sırasıyla dış try bloklarının catch blokları da taranacaktır. İşte bazen bir exceptionu kısmen işleyip sanki onu hiç yakalamamış gibi bir durum oluşturmak isteriz. Yani exceptionu aynı şekilde yeniden fırlatıp, dış catch bloklarının işlemesini sağlamak isteyebilir. Bunun için throw anahtar sözcüğünün yanına hiçbir ifade yazılmadan ";" ile kapatılır.

Örneğin:

```
try
{
     Foo();
}
catch(MyException e)
{
     //...
     throw;
}
```

Bu durumda yine finally bloğu çalıştırılacaktır.

System.Exception SINIFI

Anımsanacağı gibi bütün exception sınıfları System.Exception sınıfından türetilmiştir. Bu sınıfın en önemli elemanı Message isimli property elemanıdır:

```
public virtual string Message {get;}
```

Görüldüğü gibi bu property virtual bir propertydir. Türemiş sınıflarda bu Message propertyleri kendi mesajlarını döndürecek şekilde override edilmiştir. Bu durumda biz bir exceptionu, exception sınıfı ile yakalarsak çok biçimli bir biçimde exceptionu oluşturan mesajı elde edebiliriz. Örneğin:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                int val = int.Parse("xxxx"); // FormatException
                //...
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);                    //
Format.Exception.Message çağrılır
            }

        }
```

```
    }
}
```

sealed SINIFLAR

Bir sınıf bildiriminin önüne `sealed` anahtar sözcüğü getirilebilir. Örneğin:

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

sealed sınıflarda türetme yapılamaz. Programcı türetmenin anlamlı olmadığı durumlarda sınıfı `sealed` yaparak okunabilirliği arttırabilir.

Static SINIFLAR

Bir sınıf, `static` anahtar sözcüğü getirilerek `static` yapılabilir. Örneğin:

```
Static class Sample
{
     //...
}
```

Static bir sınıf, yalnızca static elemanlara sahip olabilir. Örneğin `System.Math` sınıfının tüm elemanları statictir. Şüphesiz static sınıf türünden nesne `new` operatörü ile yaratılamaz.

sealed override FONKSİYONLAR

`sealed` anahtar sözcüğü, fonksiyonlar için de kullanılabilir. Ancak bu durumda fonksiyonun `override` bir fonksiyon olması gerekir. `sealed` anahtar sözcüğü `static`, `virtula` ya da `abstract` fonksiyonlarla kullanılamaz. Örneğin:

```
class A
{
     public virtual void Func()
     {
          //...
}
//...
}

class B : A
{
     public sealed override void Func()
     {
          //...
     }
}
```

sealed override bir fonksiyon, ilişkin olduğu sınıftan bir sınıf türetilse bile override edilemez. Örneğin:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                int val = int.Parse("xxxx"); // FormatException
                //...
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);                    //
Format.Exception.Message çağrılır
            }
        }
    }

    class A
    {
        public virtual void Func()
        {
            //...
        }
        //...
    }

    class B : A
    {
        public sealed override void Func()
        {
            //...
        }
    }

    class C : B
    {
        public override void Func()
        {
            //...
        }
    }
}
```

New belirleyicisi

Taban ve türemiş sınıflarda, aynı isimli elemanların bulunması yasaklanmamıştır. Bu durumda türemiş sınıf içerisinden ya da türemiş sınıf referansı ile yapılan erişimlerde, türemiş sınıftaki isim taban snıftaki ismi gizler. İşte bu tür durumlarda, derleyici bir uyarı mesajıyla durumu bildirmektedir. Örneğin:

```
class A
{
     public int m_x;
//...
}

class B : A
{
     public int m_x;     //uyarı!
     //...
}
```

Şüphesiz taban sınıftaki elemanın gizlenmesi ancak taban sınıftaki eleman `public` ya da `protected` ise söz konusudur. Örneğin:

```
class A
{
     private int m_x;
//...
}

class B : A
{
     public int m_x;     //uyarı yok !
     //...
}
```

B'deki elemanın A'dakini gizlemesi söz konusu olmadığı için bu durumda uyarı verilmeyecektir. Gizleme için taban sınıftaki elemanın `public` ya da `protected` bölümde olması gerekir.

`new` belirleyicisi, erişim belirleyicisi anahtar sözcükle ve `abstract` , `virtual` ve `static` belirleyicilerle aynı syntax grubundadır. `new` belirleyicisi gizleme durumunda, uyarıyı kesmek için kullanılır. Örneğin:

```
class A
{
     public int m_x;
//...
}

class B : A
{
```

```
     public new int m_x; //uyarı yok!
     //...
}
```

Bir gizleme durumu söz konusu değilse, new belirleyicisinin kullanımı da uyarıya yol açar. Taban sınıfla türemiş sınıfta aynı isimli fonksiyonların bulunması, her zaman gizleme oluşturmaz. Çünkü farklı parametrik yapılara ilişkin, aynı isimli fonksiyonlar bulunabilir. Örneğin:

```
class A
{
     public void Func(string a)
     {
          //...
     }
}

class B : A
{
     public void Func(int a)  //uyarı yok
     {
          //...
     }
}
```

Fakat taban sınıfta ve türemiş sınıfta aynı isimli ve aynı parametrik yapıya ilişkin fonksiyonlar varsa, burada bir gizleme söz konusudur. new belirleyicisi ile uyarıyı kesebiliriz. Örneğin:

```
class A
{
     public void Func(int a)
     {
          //...
     }
}

class B : A
{
     public new void Func(int a)   //uyarı yok
     {
          //...
     }
}
```

Taban sınıftaki virtual bir fonksiyon, türemiş sınıfta normal ya da virtual olarak bildirilirse, bu durum override etmek anlamına gelmez. Bu durumda türemiş sınıfta yeniden bir sanallık başlatılmış olur. Örneğin:

```
class A
{
     public virtual void Func(string a)
```

```
     {
          //...
     }
}

class B : A
{
     public virtual void Func(int a)    //uyarı var
     {
          //...
     }
}
```

Ya da örneğin:

```
class A
{
     public virtual void Func(string a)
     {
          //...
     }
}

class B : A
{
     public void Func(int a)  //uyarı var
     {
          //...
     }
}
```

Örneğin:

```
A a = new B();
a.Func();  à A daki Func çağrılır.
```