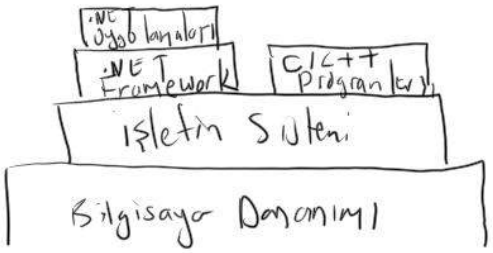
C# Programama Dİli Ve Nesne Yönelimli Programlamaya Giriş Kurs Notları (Nisan-2016)
Eğitmen: Kaan ASLAN

.NET Nedir?

.NET Microsoft'un yeni kuşak, modern bir uygulama geliştirme ve çalıştırma ortamıdır. Microsoft .NET'in ilk resmi duyurusunu 2000 yılında yaptı. Bu ortamın ilk versiyonu 2002 yılında oluşturuldu (.NET Framework 1.0). Bundan sonra 2003'te Framework 1.1 ve 2005'te Framework 2.0 piyasaya sürüldü. Bunu 2007'de .NET Framework 3.0, 2009'da 3.5 ve 2010'da 4.0 izledi. Bugün Framework 4.5 bu sistemin en son sürümü olarak kullanılmaktadır.

.NET bir işetim sistemi değildir. İşletim sisteminin üzerine kurulan bir katmadır.



.NET amaç bakımından Java ortamına çok benzemektedir. Java denildiğinde hem bir ortam (Java Framework) hem de bir programlama dili anlaşılır. (Bu nedenle yalnızca Java demek yerine “Java Framework” ya da “Java Programming Language” demek daha uygundur.)

Anahtar Notlar: Framework sözcüğünün ne anlam ifade ettiği tartışmalı bir konudur. Fakat pek çokları bir olgunun framework olması için aşağıdaki bazı olnakaları sağlıyor olması gerektiğini düşünmektedir:

- Framework’ler programcıların işini kolaylaştıracak alt sistemlere sahiptir. Kolay kullanım amacı framework’lerin en önemli özelliklerindendir

- Framework’ler programın akışını programcıdan alarak programcı içim onun işini kolaylaştıracak işlemleri yaparlar

- Framework’ler genişletilebilir bir yapı sunarlar.

C# Nedir?

C# Microsoft'un .NET ortamı için tasarladığı nesne yönelimli ve çok modelli (multi paradigm) bir programlama dilidir. Dil Anders Hejlsberg ve dört arkadaşı tarafından tasarlanmıştır. Hejlsberg daha önce Borland firmasında çalışıyordu ve Delphie ürününün sorumlusuydu.

C#'taki # müzikteki # işaretinden gelmektedir. (Yani C'nin daha yüksek seviyelibir versiyonu gibi bir espri yapılmıştır.)

C# dil olarak %70 oranında Java'ya benzemektedir. Tasarım Java'dan alınmıştır. Fakat C++'a daha fazla yaklaştırılmıştır. Yani C# adeta dil olarak Java'nın çok iyileştirilmiş bir biçimidir.

C# ECMA ve ISO tarafından standardize edilmiş bir programa dilidir. (C#’ın ECMA standartlarındaki kod numarası ECMA-334'tür).

.NET'in Temel Özellikleri

1) Arakodlu Çalışma Sistemi: .NET ortamında derlenen bir program gerçek makine komutlarını (yani doğal kodları) içermez. İsmine CIL (Common Intermediate Language) denilen yapay bir arakod içerir. (Halbuki C gibi bir dilde derleme yaptığımızda derlenmiş olan program gerçek makina komutlarını içermektedir.) Arakod içeren programlar doğrudan çalıştırılamazlar. Bunların çalıştırılması için .NET ortamının o bilgisayarda kurulu olması gerekir. İşte böyle bir program çalıştırılmak istendiğinde .NET ortamının CLR (Common Language

Runtime) denilen alt sistemi devreye girer. Bu yapay arakodları o anda gerçek makine komutlarına dönüştürüp çalıştırır. CLR'nin yaptığı bu işleme "JIT Derlemesi (Just In Time Compilation)" denilmektedir. Şüphesiz doğal kodlu çalışmaya göre burada bir yavaşlık söz konusudur. (Microsoft'un verilerine göre bu yavaşlık %18 civarı. Fakat bunun bir önemi olduğu söylenemez.)

Doğal kod içeren bir program hem işlemciye hem de işletim sistemine bağlıdır. Yani o programı ancak biz o işlemcinin ve o işletim sisteminin bulunduğu makinalarda çalıştırabiliriz. (İşlemci ve işletim sisteminin oluşturduğu birlikteliğe "platform" da denilmektedir.)

Arakodlu çalışmanın en önemli avantajı derlenmiş programın taşınabilirliğini (binary portability) sağlamasıdır. Yani biz bir C# programını derleyip .exe yaptığımızda işletim sistemi ve işlemci (processor) ne olursa olsun eğer .NET ortamı o sistemde kuruluysa program orada çalışabilir.

Aynı durum Java platformunda da tamamen benzerdir. Bir java programı derlendiğinde .class dosyası oluşur. Bu dosya C#'taki .exe dosya gibidir ve arakod içermektedir. Java dünyasında bu arakoda "Java Byte Code" denilmektedir. Java dünyasında CLR'nin karşılığı ise JVM (Java Virtual Machine)'dir.

2) Dillerarası Entegrasyon (Language Interoperability): Doğal kodlu sistemde bir programlama dilinde yazılmış olan kodun başka bir dilden kullanılması problemli bir konudur. Microsoft Windows sistemleri için farklı dillerle birarada çalışmayı sağlamak üzere COM (Component Object Module) denilen bir spesifikasyon oluşturmuştu. COM'lar bugün hala kullanılmaktadır. Ancak COM sisteminin de ayrı sorunları vardır. İşte .NET ortamıyla bu sorun tamamen ortadan kaldırılmıştır.

.NET ortamında pek çok dil kullanılabilmektedir. Tabii C# tamamen sil baştan bu ortama yönelik tasarlandığı için .NET ortamının birincil dilidir. Microsoft .NET ortamı için şu dillerin derleyicilerini bizzat kendisi yazmıştır:

- VB.NET (Eski Visual Basic'in .NET versiyonu)
- J# (Java'nın CIL kodu üreten versiyonu)
- F# (Microsoft'un yeni fonksiyonel modele uygun dili)
- C++/CLI (C++'ın .NET'leştirilmiş hali)

Bunun dışında başkaları tarafından .NET ortamı için yazılmış derleyiciler ve diller de vardır.

İşte .NET'te biz .NET uyumlu dilleri aynı projede beraber kullanabilmekteyiz. Çünkü bu dillerin derleyicileri aynı arakodu üretmektedir.

3) Geniş Bir Sınıf Kütüphanesi (Framework Class Library): .NET ortamında geniş bir sınıf kütüphanesi vardır. Yani pek çok işi yapan sınıflar hazır olarak zaten bulunmaktadır. Bu sınıf kütüphanesi yalnızca C#'tan değil tüm .NET dillerinden ortak olarak kullanılmaktadır. Kütüphanenin çeşitli bölümleri çeşitli isimlerle anılır. Örneğin:

- Kütüphanenin GUI işlemleri için kullanılan kısmına "Forms" ve "WPF" denilmektedir.
- Kütüphanenin grafik çizimlerini yapan kısmına "GDI+" denilmektedir.
- Kütüphanenin veritabanı işlemlerini yapan kısmına "ADO.NET" denilmektedir.
- Kütüphanenin Web sayfası yapmak için kullanılan kısmına "ASP.NET" denilmektedir.

4) Hızlı Uygulama Geliştirme Ortamı: .NET bir hızlı uygulama geliştirme (rapid application development) ortamı sunar. Bu ortam birtakım görsel araçların kullanımına destek vermektedir. Dilin öğrenilmesi için ve ürün ortaya çıkartılması için gereken süre kısaltılmıştır.

5) Güvenli Bir Uygulama Geliştirme Ortamı: .NET ortamı daha güvenli bir çalışma sunmaktadır. .NET ortamı için yazılmış programlara virüs girme olasılığı doğal kodlu ortamlara göre daha zayıftır. Birtakım malware kodlar program çalışırken CLR tarafından tespit edilebilmektedir. Bozuk yazılmış bir programın sisteme zararverme olasılığı daha düşüktür.

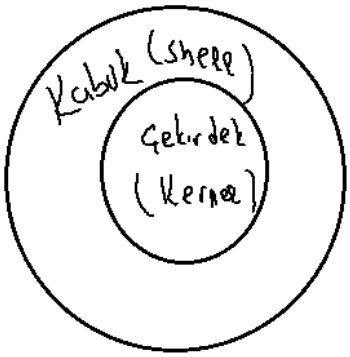
.NET ve CLI (Common Language Infrastructure)

.NET aslında bir ürünün ismidir. .NET ortamının genel standardı ECMA-335'te tanımlanmıştır ve bu standarda CLI (Common Language Infrastructure) denilmektedir. (Yani örneğin aslında CLI kağıt mendilse, .NET Selpak gibidir.) CLI standartlarının .NET dışındaki en önemli gerçekleştirimi Mono'dur (www.mono-project.org). UNIX/Linux ve Mac OS X sistemlerinde Mono kullanılmaktadır. (Yani ".NET'in Linux karşılığı Mono'dur" denilebilir). Mono'nun Windows versiyonu da vardır. .NET Microsoft'un bir malıdır, Mono ise ,"open source" bir

projedir.

Temel Kavramlar

İşletim Sistemi (Operating System): İşletim sistemi makinenın donanımını yöneten, makine ile kullanıcı arasında arayüz oluşturan temel bir sistem programıdır. İşletim sistemi olmasa daha biz bilgisayarı açtığımızda bile bir şeyler göremeyiz. İşletim sistemleri iki katmandan oluşmaktadır. Çekirdek (kernel), makinanın donanımını yöneten kontrol kısmıdır. Kabuk (shell) ise kullanıcıyla arayüz oluşturan kısımdır. (Örneğin Windows'ta masaüstü kabuk görevindedir. Biz çekirdeği bakarak göremeyiz.) Tabii işletim sistemi yazmanın asıl önemli kısmı çekirdeğin yazımıdır. Çekirdek işletim sistemlerinin motor kısmıdır.



Bugün çok kullanılan bazı işletim sistemleri şunlardır:

- Windows
- Linux
- BSD'ler
- Mac OS X
- Android
- IOS
- Windows Mobile
- Solaris
- QNX

İşletim sistemlerinin bir kısmı açık kaynak kodlu bir kısmı da mülkiyete bağlıdır. Örneğin Linux, BSD açık kaynak kodlu olduğu halde Windows mülkiyete sahip bir işletim sistemidir. Mac OS X sistemlerinin çekirdeği açıktır (buna Darwin deniyor) ancak geri kalan kısmı kapalıdır.

Bazı işletitim sistemleri diğer sistemlerin kodları değiştirilerek gerçekleştirilmiştir. Bazıları sıfırdan (orijinak kod tabanına sahip) yazılmışlardır. Orijinal kod tabanaına sahip olan yani sıfırdan yazılmış olan işletim sistemlerinden bazıları şunlardır:

- Microsoft Windows
- Linux
- BSD’ler
- Solaris

Andoid Linux işletim sistem sisteminin çekirdek kodları alınarak bazı modüllerin atılması ve mobil cihazlara yönelik bazı işlevselliklerin eklenmesiyle gerçekleştirilmiş bir sistemdir. Benzer biçimde IOS da Mac OS X kodlarından devşirilmiştir. Darwin çekirdeği Free BSD ve Mach isimli çeekirdeklerin birleştirilmesiyle oluşturulmuş hibrit bir çekirdektir.

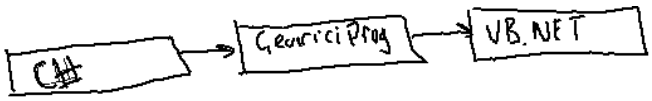
Çok kullanılan masaüstü işletim sistemlerinin mobil versiyonları da vardır. Örneğin Windows'un mobil versiyonu Windows CE (türevlerinden biri Windows Mobile)'dir. MAC OS X'in mobil versiyonu IOS'tur. Android Linux çekirdeğinin mobil hale getirilmiş bir versiyonudur.

Maalesef her mobil ortamın doğal program geliştirme ortamı vardır. Windows mobil aygıtlarının doğal geliştirme ortamı .NET'tir. Android'in Java'dır. IOS'un da Objective-C ve Swift'tir. Ancak C# ile Android (Monodroid ortamı) ve IOS'ta (Mono Touch ortamı) geliştirme yapılabilir.

Bugün masaüstü (laptoplar da dahil olmak üzere) işletim sistemlerinde Windows %70-%80 arası bir kullanıma sahiptir. Mac OS X'in kullanım oranı %10 civarındadır. Linux'un gündelik yaşamda kişisel bilgisayar olarak kullanım oranı %1 civarlarındadır. Ancak server dünyasında UNIX türevi işletim sistemlerinin payları %60'ın yukarısındadır. Yani UNIX/Linux sistemleri server dünyasında en çok kullanılan sistemlerdir. 2015 yılı itibarıyla Android akıllı telefonlarda %80'in yukarısında bir yaygınlığa sahiptir. IOS %13 civarıyla Andorid'i izlemektedir. Windows Mobile sistemlerinin kullanım oranı %2.5 civarındadır.

Gömülü Sistemler (Embedded Systems): Asıl amacı bilgisayar olmayan fakat bilgisayar devresi içeren sistemlere genel olarak gömülü sistemler denilmektedir. Örneğin elektronik tartılar, biyomedikal aygıtlar, GPS cihazları, turnike geçiş sistemleri, müzik kutuları vs. birer gömülü sistemdir. Gömülü sistemlerde en çok kullanılan programlama dili C'dir. Ancak son yıllarda Raspberry Pi gibi, Banana Pi gibi, Orange Pi gibi güçlü ARM işlemcilerine sahip kartlar çok ucuzlamıştır ve artık gömülü sistemlerde de doğrudan kullanılır hale gelmiştir. Bu kartlar tamamen bir bilgisayarın işlevselliğine sahiptir. Bunlara genellikle Linux işletim sistemi ya da Andrid yüklenir. Böylece gömülü yazılımların güçlü donanımlarda ve birt işletim sistemi altında çalışması sağlanabilmektedir. Örneğin Raspberry Pi'a biz Mono'yu yükleyerek C#'ta program yazıp onu çalıştırabiliriz.

Çevirici Programlar (Translators), Derleyiciler (Compilers) ve Yorumlayıcılar (Interpreters): Bir programlama dilinde yazılmış olan programı eşdeğer olarak başka bir dile dönüştüren programlara çevirici programlar (translators) denilmektedir. Çevirici programlarda dönüştürülmek istenen programın diline kaynak dil (source language), dönüşüm sonucunda elde edilen programın diline hedef dil (target/destination language) denilmektedir. Örneğin:



Burada kaynak dil C#, hedef dil VB.NET'tir.

Eğer bir çevirici programda hedef dil aşağı seviyeli bir dil ise (saf makina dili, arakod ve sembolik makine dilleri alçak seviyeli dillerdir) böyle çevirici programlara derleyici (compiler) denir. Her derleyici bir çevirici programdır fakat her çevirici program bir derleyici değildir. Bir çeviri programa derleyici diyebilmek için hedef dile bakmak gerekir. Örneğin arakodu gerçek makina koduna dönüştüren CLR bir derleme işlemi yapmaktadır. Sembolik makina dilini saf makina diline dönüştüren program da bir derleyicidir.

Bazı programlar kaynak programı alarak hedef kod üretmeden onu o anda çalıştırırlar. Bunlara yorumlayıcı (interpreter) denilmektedir. Yorumlayıcılar birer çevirici program değildir. Yorumlayıcı yazmak derleyici yazmaktan daha kolaydır. Fakat programın çalışması genel olarak daha yavaş olur. Yorumlayıcılarda kaynak kodun çalıştırılması için onun başka kişilere verilmesi gerekir. Bu da kaynak kod güvenliğini bozar.

Bazı diller yalnızca derleyicilere sahiptir (C, C++, C#, Java gibi). Bazıları yalnızca yorumlayıcılara sahiptir (PHP, Perl gibi). Bazılarının hem derleyicileri hem de yorumlayıcıları vardır (Basic gibi). Genel olarak belli bir alana yönelik (domain specific) dillerde çalışma yorumlayılar yoluyla yapılmaktadır. Genel amaçlar diller daha çok derleyiciler ile derlenerek çalıştırılırlar.

IDE (Integrated Development Environment): Derleyiciler komut satırından çalıştırılan programlardır. Bir programlama faaliyetinde program editör denilen bir program kullanılarak yazılır. Diske save edilir. Sonra komut satırından derleme yapılır. Bu yorucu faaliyettir. İşte yazılım geliştirmeyi kolaylaştıran çeşitli araçları içerisinde barındıran (integrated) özel yazılımlara IDE denilmektedir. IDE'nin editörü vardır, menüleri vardır ve çeşitli araçları vardır. IDE'lerde derleme yapılırken derlemeyi IDE yapmaz. IDE derleyiciyi çağırır. IDE yardımcı bir araçtır, mutlak gerekli bir araç değildir.

Microsoft'un ünlü IDE'sinin ismi "Visual Studio"dur. Apple'ın "X-Code" isimli IDE'si vardır. Bunların dışında başka şirketlerin malı olan ya da "open source" olan pek çok IDE mevcuttur. Örneğin "Eclipse" ve "Netbeans"

yaygın kullanılan cross-platform “open source” IDE'lerdir. Linux altında Mono'da “Mono Develop” isimli bir IDE tercih edilmektedir.

VisualStudio'nun “Express Edition" ya da "Community Edition ” isimli bedava bir sürümü de vardır. Bu bedava sürüm bu kurstaki gereksinimleri tamamen karşılayabilir. Visual Studio'nun bugün için son versiyonu "Visual Studio 2015"tir.

Mülkiyete Sahip Yazılımlar, Özgür ve Açık Kaynak Kodlu Yazılımlar

Yazılımların çoğu bir firma tarafından ticari amaçla yazılırlar. Bunlara mülkiyete bağlı yazılım (proprieatary) denilmektedir. 1980'li yılların ortalarında Richard Stallman tarafından “Özgür Yazılım” hareketi başlatılmıştır. Bunu daha sonra “Open Source” ve türevleri izlemiştir. Bunların çoğu birbirine benzer lisanslara sahiptir. Özgür yazılımın ve açık kaynak kodlu yazılımın temel prensipleri şöyledir:

- Program yazıldığında yalnızca executable dosyalar değil, kaynak kodlar da verilir
- Kaynak kodlar sahiplenilemez.
- Bir kişi bir kaynak kodu değiştirdiğinde ve geliştirdiğinde o da ürününü açmak zorundadır.
- Program istenildiği gibi dağıtılıp kullanılabilir.

Bugün Linux dünyasındaki ürünlerin çoğu bu kapsamdadır. Biz bir yazılımı ya da bileşeni kullanırken onun lisansına dikkat etmeliyiz.

Bugün özgür yazılım ve açık kaynak kod hareketi çok ilerlemiştir. Neredeyse popüler pek çok ürürünün açık kaynak kodlu bir versiyonu vardır.

Bit ve Byte Kavramları

Bilgisayarlar 2'lik sistemi kullanırlar. Bu nedenle bilgisayarın belleğinde, diskinde vs. her şey ikilik sistemde bulunmaktadır. İkilik sistemde sayıları yazarken yalnızca 1'ler ve 0'lar kullanılır. Böylece bilgisayarın içerisinde yalnızca 1'ler ve 0'lar vardır. Her şey 1'lerden ve 0'lardan oluşmaktadır. 2'lik sistemdeki her bir basamağa bit (binary digit'ten kısaltma) denilmektedir. Bu durumda en küçük bellek birimi bit'tir. Bit çok küçük olduğu için 8 bit'e 1 byte denilmiştir. Bellek birimi olarak byte kullanılır. Bilgisayar bilimlerinde Kilo 1024 katı anlamına gelir. Örneğin 1KB = 1024 byte. Mega da kilonun 1024 katıdır. Örneğin, 1MB=1024KB'tır.

Dil Nedir?

Dil karmaşık bir olgudur. Tek bir cümleyle tanımını yapmak pek mümkün değildir. Fakat kısaca “iletişim için kullanılan semboller kümesidir” denebilir. Bir dilin tüm kurallarına gramer denir. Gramerin en önemli iki alt alanı sentaks (syntax) ve semantik (semantic)'tir. Bir dili oluşturan en yalın öğelere atom ya da sembol (token) denilmektedir. Örneğin doğal dillerde atomlar sözcüklerdir.

Bir olgunun dil olabilmesi için asgari sentaks ve semantik kurallara sahip olması gerekir. Sentaks doğru dizilime ve doğru yazıma ilişkin kurallardır. Örneğin:

“I school to am going”

Burada İngilizce için bir sentaks hatası vardır. Sözcükler doğrudur fakat dizilimleri yanlıştır. Örneğin:

“Herkez çok mutluydu”

Burada da bir sentaks hatası vardır. Türkçe'de “herkez” biçiminde bir sözcük (yani sembol) yoktur. Örneğin:

“if a > 10)”

Burada da C#'ça bir sentaks hatası yapılmıştır.

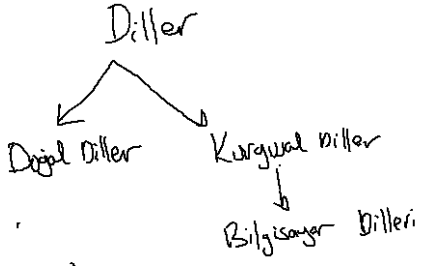
Semantik doğru yazılmış ve dizilmiş öğelerin ne anlam ifade ettiğine ilişkin kurallardır. Yani bir şey doğru

yazılmıştır fakat ne anlama gelmektedir? Örneğin:

“ I am going to school”

sentaks bakımından geçerlidir. Fakat burada ne denmek istenmiştir? Bu kurallara semantik denilmektedir.

Diller doğal ve kurgusal (ya da yapay) olmak üzere ikiye ayrılabilir. Doğal dillerde sentaksın tam bir fomülasyonu yoktur. Kurgusal diller insanlar tarafından formüle edilebilecek biçimde tasarlanmış dillerdir. Bilgisayar dilleri kurgusal dillerdendir.



Kurgusal dillerde istisnalar ya yoktur ya da çok azdır. Sentaks ve semantik tutarlıdır. Doğal dillerde pek çok istisna vardır. Doğal dillerin zor öğrenilmesinin en önemli nedenlerinden birisi de istisnadır.

Bilgisayar Dilleri ve Programlama Dilleri

Bilgisayar Bilimlerinde kullanılan dillere bilgisayar dilleri (computer languages) denir. Bir bilgisayar dilinde akış varsa ona aynı zamanda programlama dili de (programming language) denilmektedir. Örneğin HTML bir bilgisayar dilidir. Fakat HTML'de bir akış yoktur. Bu nedenle HTML bir programlama dili değildir. HTML'de de sentaks ve semantik kurallar vardır. Oysa C#'ta bir akış da vardır. C# bir programlama dilidir.

Programlama Dillerinin Sınıflandırılması

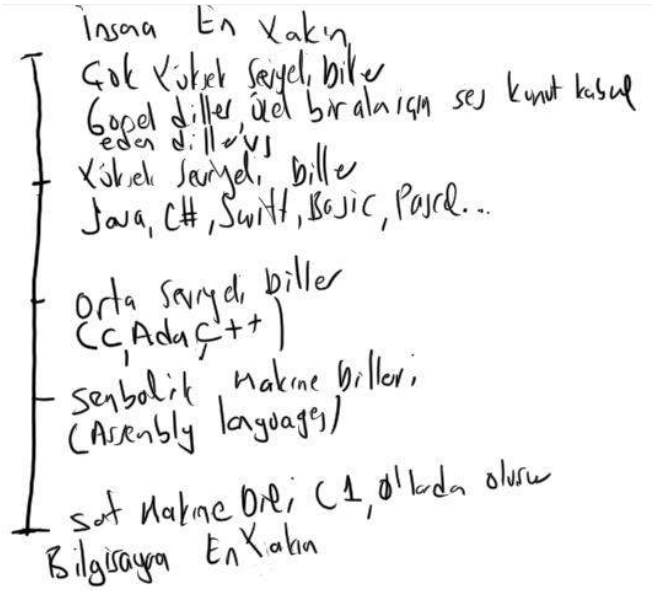
Programlama dilleri üç biçimde sınıflandırılabilir:

1) Seviyelerine Göre sınıflandırma
2) Kullanım Alanlarına Göre Sınıflandırma
3) Programlama Modeline Göre Sınıflandırma

Seviyelerine Göre Sınıflandırma: Seviye (level) bir programlama dilinin insan algısına yakınlığının bir ölçüsüdür. Yüksek seviyeli diller kolay öğrenilebilen insana yakın dillerdir. Alçak seviyeli diller bilgisayara yakın dillerdir. Olabilecek en alçak seviyeli diller 1'lerden 0'lardan oluşan saf makina dillerdir. Bunun biraz yukarısında sembolik makina dilleri (assembly languages) bulunur. Biraz daha yukarıda orta seviyeli diller bulunur. Daha yukarıda yüksek seviyeli, en yukarıda da “çok yüksek seviyeli diller” bulunmaktadır. Örneğin:

- Java, C#, Pascal, Basic “yüksek seviyeli” dillerdir.
- C “orta seviyeli” bir dildir.
- C++ orta ile yüksek seviye arasındadır.

Tabi aslında dillerin seviyelerini sürekli çizgi üzerinde noktalar biçiminde düşünebiliriz. İki yüksek seviyeli dil bu çizgide aynı yerde bulunmak zorunda değildir. Seviyelerine göre dilleri sınıflandırırken genellikle bir seviye çizgisinden faydalanılır:



Kullanım Alanlarına Göre Sınıflandırma: Bu sınıflandırma biçimi dillerin hangi amaçla daha çok kullanıldığına yöneliktir. Tipik sınıflandırma şöyle yapılabilir:

- Bilimsel ve Mühendislik Diller: C, C++, Java, C#, Fortran, Pascal, Matlab gibi...
- Veritabanı Yoğun İşlemlerde Kullanılan Diller: SQL, Foxpro, Clipper, gibi...
- Web Dilleri: PHP, C#, Java
- Animasyon Dilleri: Action Script gibi...
- Yapay Zeka Dilleri: Lisp, Prolog, C, C++, C#, Java gibi...
- Sistem Programlama Dilleri: C, C++, Sembolik Makina Dilleri
- Genel Amaçlı Diller: C, C++, Pascal, C#, Java, Python, Basic gibi...

Programlama Modeline Göre Sınıflandırma: Program yazarken hangi modeli (paradigm) kullandığımıza yönelik sınıflandırmadır. Altprogramların birbirlerini çağırmasıyla program yazma modeline "prosedürel programlama modeli (procedural programming paradigm)" denilmektedir. Bir dilde yalnızca alt programlar oluşturabiliyorsak, sınıflar oluşturamıyorsak bu dil için "prosedürel programlama modeline uygun olarak tasarlanmış" bir dil diyebiliriz. Örneğin klasik Basic, Fortran, C, Pascal prosedürel dillerdir. Bir dilde sınıflar varsa ve program sınıflar kullanılarak yazılıyorsa böyle dillere "nesne yönelimli diller (object oriented languages)" denilmektedir. Eğer program formül yazar gibi yazılıyorsa bu modele fonksiyonel model (functional paradigm), bu modeli destekleyen dillere de fonksiyonel diller denilmektedir. Bazı dillerde program görsel olarak fare hareketleriyle oluşturulabilmektedir. Bunlara görsel diller denir. Bazı diller birden fazla programlama modelinin kullanılmasına olanak sağlayacak biçimde tasarlanmıştır. Bunlara da çok modelli diller (multiparadigm languages) denilmektedir. Örneğin C++ gibi. C#'a son yıllarda Microsoft tarafından eklenen bazı özellikler ona belli oranda fonksiyonel programlama yeteneği de kazandırmıştır. Bu durumda C# için de belki çok modellidir denilebilir. Yine Apple'ın yeni tasarladığı Swift dili de çok modellidir.

Tüm bunlar ışığında C# için şunlar söylenebilir: C# yüksek seviyeli, bilimsel ve mühendislik uygulamalarda, Web programlamada, yapay zeka uygulamalarında kullanılabilen genel amaçlı, nesne yönelimli ve fonksiyonel özellire de olan bir programlama dilidir.

Klavyedeki Karakterlerin İngilizce İsimleri

Sembol	İsim
+	plus
-	minus, hyphen, dash
*	asterisk
/	slash
\	back slash
.	period, dot

,	comma
:	colon [ko:lın]
;	semicolon
“	double quote [dabıl kvot]
'	single quote
(...)	paranthesis [pıran(th)isi:s] left, right, opening, closing
[...]	(square) bracket left, right, opening, closing
{...}	brace [breys] left, right, opening, closing
=	equal sign [i:kvıl sayn]
&	ampersand
~	tilda
@	at
<...>	less than, greater than, angular bracket
^	caret [k(ea)rıt]
|	pipe [payp]
_	underscore [andırsko:r]
?	question mark
#	sharp, number sign, hashtag
%	percent sign [pörsınt sayn]
!	exclamation mark [eksklemeyşın mark]
$	dollar sign [dalır sayn]
...	ellipsis [elipsis]

Merhaba Dünya Programı

Ekrana “Merhaba Dünya” yazısını çıkartan bir C# programı şöyle yazılabilir:

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

Program notepad gibi bir editörde yazılır ve uygun yere .cs uzantısıyla save edilir (örneğin Sample.cs olsun). Bir C# programını IDE olmadan derlemek için öncelikle komut satırına geçmek gerekir. Komut satırı cmd.exe isimli programdır. Fakat komut satırı programı olarak Start Menü'den Visual Studio/Visual Studio Tools menüsündeki program kullanılmalıdır. İkide bir bu menüyü kullanmak yerine masaüsüne kısayol oluşturulabilir.

Komut satırında DOS komutları uygulanabilmektedir. Sürücü değiştirmek için sürücü ismi ve ‘:’ kakterlerini yazıp ENTER tuşuna basarız. Örneğin:

```
e:
```

Bu işlemle e sürücüsüne geçilecektir. Dizin değiştirmek için cd komutu kullanılır. Örneğin:

```
cd CSharp-Subat-2015
```

Bir üst dizine geçmek için cd .., aynı sürücünün kök dizinine geçmek için cd \ komutları kullanılır.

Klasörün (dizinin) içerisindekileri görmek için dir komutu uygulanabilir:

```
dir
```

Bir dosyanın içini görüntülemek için ise type komutundan faydalanılır. Örneğin:

```
type Sample.cs
```

Programı derlemek için şunlar yapılmalıdır:

```
csc <dosya ismi>
```

Örneğin:

```
csc Sample.cs
```

İşlem başarılı olursa .exe uzantılı bir dosya oluşacaktır. Bunun ismini yazıp ENTER tuşuna basarsak prgram çalışır.

Atom (Token) Kavramı

Bir programlama dilinde kendi başına anlamı olan en küçük yazısal birime atom (token) denir. Atomlar daha fazla anlamlı parçaya ayrılamazlar. Aslında bir program atomların belli bir kurala göre yan yana getirilmesiyle oluşur (buna da sentaks denilmektedir). Örneğin "Merhaba Dünya" programı atomlarına şöyle ayrılabilir:

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

Derleyici de önce kaynak programı atomlarına ayırmaktadır.

Atomlar 6 gruba ayrılmaktadır:

1) Anahtar Sözcükler(Keywords/Reserved Words): Dil için özel anlamı olan, derleyici tarafından doğrudan tanınan, değişken olarak kullanılması yasaklanmış sözcüklerdir. Örneğin if, public, namespace gibi...

2) Değişkenler (Identifiers/Variables): İsmini bizim (ya da kodu yazanın) istediğimiz gibi verebildiğimiz atomlardır. Örneğin App, CSD, count, x, y gibi...

3) Operatörler (Operators): Bir işleme yol açan ve o işlem sonucunda bir değer üretilmesini sağlayan +, -, * gibi sembollere operatör denilmektedir. Örneğin:

```
a = b + c;
```

Burada a, b ve c değişken atom, = ve + operatör atomdur.

4) Sabitler (Literals/Constants): Bir değer ya bir değişkenin içerisindedir ya da doğrudan yazılmıştır. Doğrudan yazılan sayılara sabit denir. Örneğin:

a = b + 10;

Burada 10 sabit bir atomdur.

5) String'ler (Strings): İki tırnak içerisindeki yazılar iki tırnaklarıyla birlikte tek bir atomdur. Bunlara string denir. Örneğin:

```
“Bugün hava çok güzel”
```

6) Ayıraçlar (Delimeters/Punctuators): İfadeleri ayırmak için kullanılan atomlara ayıraç denir. Örneğin';', '{', '}' birer ayrıraç atomdur.

Merhaba Dünya proghramındaki atomların türleri şöyledir:

namespace	Anahtar sözcük
CSD	Değişken
{	Ayıraç
class	Anahtar Sözcük
App	Değişken
{	Ayıraç
public	Anahtar Sözcük
static	Anahtar Sözcük
void	Anahtar Sözcük
Main	Değişken
(	Operatör
)	Operatör
{	Ayıraç
System	Değişken
.	Operatör
Console	Değişken
.	Operatör
WriteLine	Değişken
(	Operatör
"Merhaba Dünya"	String
)	Operatör
;	Ayıraç
}	Ayıraç
}	Ayıraç
}	Ayıraç

Boşluk Karakterleri (White Space)

Boşluk duygusu oluşturmak için kullanılan karakterlere boşluk karakterleri denir. En tipik boşluk karakterleri SPACE, TAB ve ENTER karakterleridir. Ancak klavyeden çıkartılmıyor olsa da başka boşluk karakterleri de vardır. (Örneğin "Vertical TAB" karakteri gibi.)

C#'ın Yazım Kuralları

C#'ın yazım kuralları iki maddeyle özetlenebilir:

1) Atomlar arasında istenildiği kadar boşluk karakterleri bulundurulabilir.

2) Atomlar istenildiği kadar bitişik yazılabilirler. Fakat anahtar sözcükler ve/veya değişkenler arasında en az bir boşluk karakteri bulundurulmak zorundadır.

Fakat programlar bakıldığında anlaşılabilecek biçimde (readable) yazılmalıdır. Örneğin aşağıdaki prgram geçerlidir:

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

Fakat bu yazım okunabilir (readable) değildir. Aşağıdaki program da geçerlidir:

```
namespace
CSD

       {class
App
{

               public static
               void Main
(

){
System         .
       Console.      WriteLine

(
"Merhaba Dunya"
)
               ;
}
       }
               }
```

Bu yazım da okunabilir değildir.

Programlama Dillerinde Sentaks'ın İfade Edilmesi

Programlama dillerinin sentakslarını formel olarak ifade edebilmek için çeşitli teknik notasyonlar geliştirilmiştir. Bunların en yaygını BNF (Backus-Naur Form) notasyonudur. Programlama dillerinin standartlarında genellikle BNF notasyonu ya da bunun türevleri kullanılmaktadır. ISO BNF notasyonunu EBNF (Extended BNF) ismiyle genişleterek standardize etmiştir. Fakat kursumuzda sentaks açıklamak için bunun yerine açısal parantez köşeli parantez tekniği kullanılacaktır. Bu tekniğin özeti şöyledir:

- Açısal parantez içerisindeki öğeler yazılması zorunlu öğeleri belirtir.
- Köşeli parantez içerisindeki öğeler ise yazılması isteğe bağlı (optional) öğeleri belirtir.
- Bunlar dışındaki tüm atomlar aynı sırada ve aynı biçimde bulundurulmak zorundadır. Örneğin:

```
class <sınıf ismi>
{
       //...
```

}

- Gösterimimizde ayrıca anahtar sözcüklerin altı çizilecektir.

Ayrıca genel gösterimlerdeki //... ifadesi “burada başka birşeyler var, fakat biz onunla şimdilik ilgilenmiyoruz” anlamına gelmektedir.

Merhaba Dünya Programının Açıklaması

Bir C# programı kabaca isim alanlarından (namespace'lerden), isim alanları sınıflardan (class'lardan), sınıflar da metotlardan oluşur. Bir isim alanı bildiriminin genel biçimi şöyledir:

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

Sınıf bildirimin genel biçimi de şöyledir:

```
class <sınıf ismi>
{
      //...
}
```

Metot bildirimlerinin ise genel biçimi ise şöyledir:

```
[erişim belirleyici] [static] <geri dönüş değerinin türü> <metot ismi>([parametre bildirimi])
{
      //...
}
```

Erişim belirleyicisi aşağıdaki anahtar sözcüklerden biri olabilir:

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

Erişim belirleyicisinin hiç yazılmaması ile oraya private yazılması aynı anlama gelmektedir (yani erişim belirleyicisinin default durumu private'tir.). Fakat biz kursumuzda erişim belirleyicileri konusu ele alınana kadar erişim belirleyicisini hep public alacağız.

Bir metot static olabilir ya da olmayabilir. static olmayan metotlara C# standartlarında İngilizce "instance method" denilmektedir. Biz şimdilik hep static metot kullanacağız.

C#'ta iki küme parantezi arasındaki bölgeye blok (block) denir. Pek çok sentaktik yapı blok içermektedir. Örneğin isim alanlarının, sınıfların ve metotların blokları vardır.

C#'ta altprogramlara metot (method) denilmektedir. İç içe metotlar bildirilemez.

Anahtar Notlar: Altprogramlara bazı dillerde “procedure”, bazı dillerde “function”, bazı dillerde “subroutine”, bazılarında ise “method” denilmektedir. Fakat kavram aynıdır.

C# programları Main isimli özel bir metottan çalışmaya başlar. Main metodu herhangi bir isim alanındaki herhangi bir sınıfın içerisinde bulunabilir. Main metodu static olmak zorundadır fakat public olmak zorunda değildir. Şüphesiz bir programda "bir ve yalnızca bir tane" Main metodu bulunmak zorundadır.

Bir C# programı Main metodundan çalışmaya başlar. Main bitince program da biter.

Bir metodun bildirilmesi (declaration) demek onun bizim tarafımızdan yazılması demektir. Çağrılması (calling) demek onun çalıştırılması demektir. Şüphesiz var olan yani bildirilmiş olan metotlar çağrılabilir. static metot çağırmanın genel biçimi şöyledir:

```
[isim alanı ismi][.][sınıf ismi][.]<metot ismi>([argüman listesi]);
```

Bir metot çağrıldığında akış metoda gider. O metodun kodları soldan sağa, yukarıdan aşağıya çalışır, metot bitince akış kalınan yerden (çağrılma işleminin yapıldığı yerden) devam eder.

Merhaba Dünya programında System isim alanı içerisindeki Console sınıfının WriteLine isimli metodu çağrılmıştır. Console sınıfı .NET'in sınıf kütüphanesinde zaten yazılmış olan ve hazır bulunan bir sınıftır. WriteLine metodu iki tırnak içerisindeki yazıyı ekrana basmaktadır.

Bir C# programında istenildiği kadar çok isim alanı, bir isim alanında istenildiği kadar çok sınıf ve bir sınıfta istenildiği kadar çok metot bulunabilir. Fakat programın akışı her zaman Main isimli metottan başlayacaktır. Örneğin:

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

namespace Test
{
	class Sample
	{
		public static void Foo()
		{
			System.Console.WriteLine("Foo");
		}
	}
}
```

WriteLine metodu imlecin (cursor) bulunduğu yere yazıyı yazdırdıktan sonra imleci aşağı satırın başına geçirir. Böylece sonraki yazılacak yazılar aşağıda görülecektir. Console sınıfının ayrıca bir de Write metodu vardır. Bu metot imleci yazının sonunda bırakmaktadır. Örneğin:

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

Eğer çağrılan metot aynı isim alanı içerisindeki bir sınıftaysa çağrılma sırasında isim alanı ismi belirtilmeyebilir (tabii istenirse belirtilebilir de). Örneğin:

```
namespace CSD
```

```
{
	class App
	{
		public static void Main()
		{
			System.Console.WriteLine("Merhaba Dunya");
			Sample.Foo();
		}
	}

	class Sample
	{
		public static void Foo()
		{
			System.Console.WriteLine("Foo");
		}
	}
}
```

Aynı sınıfın içerisindeki bir metodu çağırmak için sınıf ismi de belirtilmeyebilir. Örneğin:

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

		public static void Bar()
		{
			System.Console.WriteLine("Bar");
		}
	}

	class Sample
	{
		public static void Foo()
		{
			System.Console.WriteLine("Foo");
		}
	}
}
```

Tersten gidersek,

```
Foo();
```

gibi çağrıda Foo'nun aynı sınıfn bir metodu olduğu sonucunu çıkartırız. Benzer biçimde:

```
Sample.Foo();
```

gibi bir çağrıda Sample sınıfının çağırma işlemini yaptığımız sınıfın içinde bulunduğu isim alanı ile aynı isim alanı içerisinde olduğu sonucunu çıkartabiliriz.

Anahtar Notlar: Programlarımızda kullandığımız Foo, Bar, Tar gibi isimlerin herhangi bir anlamı yoktur. Bunlar öylesine uydurulmuş metot isimleridir. Bu isimler pek çok yazar tarafından "herhangi bir isim niyetiyle" kullanılmaktadır.

Derleyicilerin Hata Mesajları

Derleyiciler kaynak kodu akur, anlamlandırır ve programcının yapılmasını istediği şeyleri sağlayacak bir kod

üretir. Ancak derleme işlemi sırasında kodu inceleyen derleyici bazı mesajlarla kodumuz hakkında bize bildirimlerde bulunmaktadır.i Bunlara derleyicilerin hata mesajları denir. Derleyicilerin hata mesajları 3 kısma ayrılmaktadır:

1) Uyarı Mesajları (Warnings): Uyarılar programcının yapmış olabileceği olası mantık hatalarına dikkat çekmek için verilirler. Programcı dilin sentaks ve semantik kurallarına uymuştur. Ancak derleyici onun anlamsız bazı şeyler yaptığını düşünmektedir. Ve duruma dikkat çekmek istemektedir. Programdaki uyarılar .exe dosya oluşturulmasını engellemezler. Tabii yine de programcı uyarıları dikkatlice gözden geçirmelidir. Örneğin:

```
e:\Dropbox\Kurslar\CSharp-Subat-2015\Src>csc sample.cs
Microsoft (R) Visual C# Compiler version 12.0.21005.1
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

Sample.cs(7,8): warning CS0168: The variable 'x' is declared but never used

e:\Dropbox\Kurslar\CSharp-Subat-2015\Src>
```

2) Gerçek Hata Mesajları (Errors): Gerçek hata mesajları (yani error'ler) dilin sentaks ve semantik kurallarına uyulmaması nedeniyle verilirler. Başka bir deyişle derleyici bizim yazdığımız kodu hiç anlamamıştır. Yani biz onu kurallara uygun yazmamışızdır. Dolayısıyla .exe dosya oluşturulamaz. Bunların mutlaka düzeltilmeleri gerekir. Bir programda tek bir error olsa bile program başarılı olarak derlenemez.

3) Ölümcül Hata Mesajları (Fatal Errors): Bunlar derleme işleminin devamını engeleyecek derece ciddi hatalardır. Bir program normal olarak baştan sona derleyici tarafından gözden geçirilir. Bütün hata mesajları işlemin sonunda rapor edilir. Ancak bir ölümcül hatayla karşılaşıldığında tüm işlemler sonlandırılmaktadır. Ölümcül hatalar genellikle sistemdeki anormal durumlar yüzünden oluşurlar (örneğin bellek yetersizliği, diskte yeteri kadar boş alan omaması gibi.)

Hatalara yol açan kaynaklar aynı olsa da her derleyicinin hata mesajları birbirinden farklı olabilir. Örneğin biz metot çağırırken sonuna ; koymazsak bu bir sentaks hatasıdır. Buna her C# derleyicisi error vermek zorundadır. Fakat hepsinin mesaj metni farklı olabilir. Yani dilin kuralları çeşitli kurumlar tarafından standardize edilmiştir ancak hata mesajları için bir standart yoktur.

Anahtar Notlar: Neden tüm programı Main içerisinde yazıp bitirmiyoruz da metotlarla yazıyoruz? Bir işi parçalara ayırmak onu kavramsal olarak kolaylaştırır. Buna bilgisayar dünyasında "böl ve yönet (divide conquer)" denilmektedir. Ayrıca metotlar kod tekrararını engellemek için de kullanılırlar. Örneğin bir programda aynı şeyleri farklı yerlerde yapacak olalım. Aynı kodları yeniden yazmak yerine o işi yapan bir metot yazıp farklı yerlerden o metodu çağırabiliriz. Büyük bir proje böyle metotların birbirini çağırmasıyla yazılıyorsa bu tekniğe prosedürel programlama tekniği (procedural programming paradigm) denilmektedir. C programlama dili bu tekniğin kullanılmasına olanak sağlamaktadır. Eğer bir işi metotların ötesinde sınıflara bölerek, sınıflarla gerçekleştiriyorsak bu tekniğe de nesne yönelimli model (object oriented paradigm) denilmektedir. C++, C#, Java nesne yönelimli teknik uygulansın diye tasarlanmışlardır.

C#'ın Temel Türleri

Tür (type) bir değişkenin bellekte kaç byte yer kapladığını, ona hangi formatta ve hangi sınırlarda değer atanabileceğini belirten önemli bir bilgidir. C#'ta her değişkenin ve ifadenin bir türü vardır. C#'ın türleri aşağıdaki tabloda açıklanmaktadır.

Tür Belirten Anahtar Sözcük	Byte Uzunluğu	Yapı Karşılığı	Sınır Değerler
int	4	System.Int32	[-2147483648, +2147483647]
uint	4	System.UInt32	[0, +4294967295]
short	2	System.Int16	[-32768, +32767]

ushort	2	System.UInt16	[0, +65535]
long	8	System.Int64	[-9223372036854775808, +9223372036854775807]
ulong	8	System.UInt64	[0,+ 18446744073709551615]
sbyte	1	System.SByte	[-128, +127]
byte	1	System.Byte	[0, +255]
char	2	System.Char	[0, +65535]
float	4	System.Single	[$\pm 3.6*10^{-38}$, $\pm 3.6*10^{+38}$]
double	8	System.Double	[$\pm 1.6*10^{-308}$, $\pm 1.6*10^{+308}$]
decimal	16	System.Decimal	±28 digit mantis
bool	1	System.Boolean	true, false

- int türü 4 byte uzunlukta işaretli bir tamsayı türüdür. Bir tamsayı işaretli (signed) ise hem negatif hem pozitif değerleri tutabilir. Ancak işaretsizse (unsigned) yalnızca sıfır ve pozitif değerleri tutabilir. Tamsayılar (integer) noktası olmayan sayılardır. (Bu türü int anahtar sözcüğü temsil etmektedir. int sözcüğü "integer" sözcüğünden kısaltılmıştır. Fakat "integer" bir anahtar sözcük değildir.)

- Her işaretli tamsayı türünün bir de işaretsiz biçimi vardır. int türünün işaretsiz biçimi uint türüdür. İşaretsiz biçim işaretli biçimle aynı uzunluktadır yalnızca iki kat daha geniş pozitif değer tutar.

- short türü int türünün yarı uzunluğundadır. short türü de işaretli bir tamsayı türüdür.

- short türünün işaretsiz biçimi ushort türüdür.

- long türü int türünün 2 kat uzunluğundadır. O da işaretli bir tamsayı türüdür.

- long türünün işaretsiz biçimi ulong türüdür.

- byte türü 1 byte uzunlukta işaretsiz tamsayı türüdür.

- sbyte ise 1 byte uzunlukta işaretli tamsayı türüdür.

- Yazılar karakterlerden oluşmaktadır. Aslında karakterler bilgisayar belleklerinde ikilik sistemde sayılar biçiminde tutulur. Yani yazının bir karakteri bir sayı gibi bellekte tutulmaktadır. Hangi sayının hangi karakterlere karşılık geldiğini belirlemek için çeşitli tablolar oluşturulmuştur. Bu tablolardan bazıları bir byte'lık tablolardır. (Örneğin ASCII tablosu, EBCDIC tablosu gibi.) Bir byte'lık tablolarda toplam 256 değişik karakter kodlanabilmektedir. Halbuki 2 byte'lık tablolarda toplam 65536 karakter kodlanabilir. İşte C#'ta char türü bir karakterin UNICODE tablodaki sıra numarasını tutmak için düşünülmüş bir türdür. UNICODE tablo 2 byte'lık bir karakter dönüştürme tablosudur. UNICODE tabloda tüm ülkelerin karakterleri bulunmaktadır. Fakat C#'ta her ne kadar char türü karakterlerin tablodaki kodlarını saklamak için tasarlanmış olsa da aynı zamanda aritmetik işlemlerde de kullanılabilmektedir. char türünün işaretli ve işaretsiz biçimleri yoktur. char türü aritmetik olarak kullanılırsa işaretsiz sayı belirtir.

- C#'ta üç tane gerçek sayı türü (noktalı sayı türü) vardır. Bunlar float, double ve decimal türleridir. Bu türlerin işaretli ve işaretsiz biçimleri yoktur. Bunlar zaten her zaman işaretlidir.

- float ve double türleri yuvarlama hatalarına (rounding error) yol açabilen türlerdir. Yuvarlama hatası bazı noktalı sayıların tam olarak ifade edilemeyip ona yakın bir sayının ifade edilmesiyle oluşan hatalardır. float ve double formatında bazı sayılar tam olarak ifade edilemezler bunun yerine mecburen onlara en yakın sayılar kullanılmaktadır. Yuvarlama hataları sayı ilk kez depolanırken oluşabileceği gibi, bir işlem sonucunda da oluşabilir. Örneğin biz float türünden bir değişkene bir değer atadığımız zaman aslında o değişkenin içerisine o değer yerine ona çok yakın bir değer yerleştirilmiş olabilir. Yuvarlama hatası noktalı sayıların bellekte tutuluş formatıyla ilgilidir. Bugün ağırlıklı olarak işlemciler IEEE'nin 754 numaralı gerçek sayı formatını

kullanmaktadır. Yuvarlama hatası float ve double türleri için kaçınılmaz olarak ortaya çıkar. float türünün yuvarlama hatalarına direnci zayıfıtır. Bu nedenle C#'ta noktalı sayıları tutmak için programcılar genellikle double türünü tercih ederler.

- double türü float türünün yuvarlama hatalarına direnci yüksek olan iki kat geniş biçimidir. Yukarıda da belirtildiği gibi pratikte double türü daha yaygın kullanılmaktadır.

- decimal türü 28 digit mantise sahip noktalı sayıları yuvarlama hatası olmaksızın tutabilir. (Mantis noktalı bir sayıda noktanın kaldırılmasıyla elde edilen sayı dizilimidir.) Bu türün en önemli özelliği yuvarlama hatalarına maruz kalmamasıdır. Ancak decimal türü yapay bir türdür. Yani decimal sayılar üzerindeki işlemler daha uzun zaman içerisinde ve lagoritmalarla yapılmaktadır. Bu nedenle bu tür ancak yuvarlama hatalarının hiç istenmediği durumlarda tercih edilmelidir (örneğin finansal uygulamalarda). Örneğin biz decimal türünden bir değişkende 1.xxxx biçiminde 1'den sonra 27 basamak bir sayıyı tutabiliriz. Eğer sayı 12.xxxx biçiminde ise bu durumda noktadan sonra 26 basamak sayıyı tutabiliriz. Ya da örneğin nokta hiç olmadan 28 basamaklı bir sayı da decimal türünden bir değişken içerisnde tutulabilir.

- bool türü doğru-yanlış bilgisi tutan bir türdür. Şüphesiz bu tür aslında doğru-yanlış bilgisini de sayısal düzeyde tutar. Biz bu türle bir sayı tutamayız. Yalnızca bu türden bir değişkene "true" ya da "false" değerini atayabiliriz.

C#'ta bu kadar çok tür olmasına karşın tamsayı işlemleri için programcının ilk tercih edeceği tür int, gerçek sayı işlemleri için ise double olmalıdır. Programcı içerisine yerleştireceği değerler küçük bile olsa tekil değişkenler için ilk olarak bu türleri kullanmalıdır. Ancak önemli bir gerekçe varsa diğer türleri tercih etmelidir. (Örneğin biz bir kişinin yaşını tutacak bir değişken bildirmek isteyelim. Kişinin yaşı için byte türü yeterlidir. Ancak yine programcı int türünü tercih etmelidir. Tabii eğer milyonlarca kişinin yaşları bir dizide tutulmak isteniyorsa bu durumda byte tercih edilebilir.)

Bildirim İşlemi

Katı tür kontrolünün uygulandığı dillerde bir değişkeni kullanmadan önce onu derleyiciye tanıtmak gerekir. Kullanılmadan önce değişkenlerin derleyiciye tanıtılması işlemine bildirim (decalarion) denilmektedir. C#'ta bir değişkeni ancak bildirdikten sonra kullanabiliriz. Bildirim işleminin genel biçimi şöyledir:

```
<tür> <değişken listesi>;
```

Eğer değişken listesi birden fazla değişkeni içeriyorsa araya ',' atomu konulmalıdır. Örneğin:

```
int a;
long b, c, d;
```

Bildirimler C#'ta üç yerde yapılabilir:

1) Metotların içerisinde (yani metotların ana bloğunun içerisinde). Örneğin:

```
class Sample
{
	public static void Foo()
	{
		int a;
		//...
	}
}
```

2) Metotların dışında fakat sınıf bildiriminin içerisinde. Örneğin:

```
class Sample
{
	int x;

	public static void Foo()
	{
		//...
```

```
	}
}
```

3) Metotların parametre paranztezleri içerisinde. Örneğin:

```
class Sample
{
	public static void Foo(int a)
	{
		//...
	}
}
```

Metotların içerisinde yapılan bildirimlere yerel bildirimler (local declarations), bildirilen değişkenlere de yerel değişkenler (local variables) denilmektedir. Sınıfların içerisinde bildirilen değişkenlere sınıfın veri elemanları (class fields) denir. Parametre parantezlerinin içerisinde bildirilen değişkenlere de "parametre değişkeneri" denilmektedir. Biz şimdilik yerel bildirimlerle ilgileneceğiz.

Anahtar Notlar: Console sınıfının Write ve WriteLine metotlarına argüman olarak değişken ismi verilirse bu metotlar o değişkenlerin içerisindkei değerleri ekrana yazdırırlar.

Örneğin:

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

			x = 200;

			Foo();
			System.Console.WriteLine(x);
		}

		public static void Foo()
		{
			int a;

			a = 123;
			System.Console.WriteLine(a);
		}
	}
}
```

Değişkenlere İlkdeğer Verilmesi (Initialization)

Bir değişkene bildirim sırasında = operatörü ile ilkdeğer verebiliriz. Örneğin:

```
int a = 10;
int x, y = 20, z;
```

İlkdeğer verme ile ilk kez değer verme aynı anlamda değildir. İlkdeğer verme denildiğinde bildirim işlemi sırasında değer atama anlaşılmalıdır. Örneğin:

```
int a = 10;
```

Halbuki ilk kez değer verme daha sonra yapılan bir işlemdir. Örneğin:

```
int a;
a = 10;
```

Örneğin:

```
namespace CSD
{
```

```
	class App
	{
		public static void Main()
		{
			double a = 10.2, b, c = 30.23;

			b = 12.45;
			System.Console.WriteLine(a);
			System.Console.WriteLine(b);
			System.Console.WriteLine(c);
		}
	}
}
```

Yerel Değişkenlerin Faaliyet Alanları

Bildirilen bir değişken programın her yerinde kullanılamaz. Bir değişkenin kullanılabildiği program aralığına faaliyet alanı (scope) denilmektedir. Yerel değişkenler bildirildikleri noktadan itibaren bildirildikleri bloğun sonuna kadarki bölümde kullanılabilirler. Örneğin:

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

			x = 10;
			System.Console.WriteLine(x);
		}

		public static void Foo()
		{
			System.Console.WriteLine(x);	// error!
		}
	}
}
```

Bir metot en azından bir ana blok içermek zorundadır. Fakat bunun içerisinde istenildiği kadar iç içe ya da ayrık blok da içerebilir. Örneğin:

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

				x = 10;
			}

			System.Console.WriteLine(x);	// error!
		}
	}
}
```

Görüldüğü gibi iç bir yerel blokta bildirilen değişken dış blokta kullanılamaz. Tabi bir bloktaki değişken o bloğun kapsadığı bloklarda da kullanılabilir. Örneğin:

```
namespace CSD
{
	class App
	{
```

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

				x = 10;
				{

					System.Console.WriteLine(x); // geçerli
				}
			}

		}
	}
}
```

Bir yerel blokla aynı blokta ya da onun kapsadağı blokta aynı isimli değişkenler bildirilemez. Örneğin:

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

				{
					long x;	// error!

					//...
				}
			}

	}
}
```

Önce iç blokta sonra dış blokta aynı isimli değişkenlerin bildirilmesi de error oluşturur. Örneğin:

```
namespace CSD
{
	class App
	{
		public static void Main()
		{
			{
				int x = 10;
				System.Console.WriteLine(x);
			}

			int x; // error!
			//...
		}
	}
}
```

Ancak ayrık bloklarda aynı isimli değişkenler bildirilebilir:

```
namespace CSD
{
	class App
	{
		public static void Main()
		{
			{
				int x = 10;
				System.Console.WriteLine(x);
```

```
                }

                {
                        long x = 20;
                        System.Console.WriteLine(x);

                }
        }
    }
}
```

Farklı bloklardaki aynı isimli değişkenler aslında farklı değişkenlerdir. Yani yukarıdaki örnekte birinci iç bloktaki x ile ikinci x bloktaki x aynı isimli olmasına karşın farklı değişkenlerdir. Örneğin:

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

            Foo();
            System.Console.WriteLine(a);           // 10
        }

        public static void Foo()
        {
            long a = 50;        // geçerli

            System.Console.WriteLine(a);           // 50
        }
    }
}
```

İfade Kavramı (Expressions)

Değişkenlerin, operatörlerin ve sabitlerin herbir birleşimine ifade (expression) denir. Örneğin:

```
a
100
a + 100
a + b + 50
...
```

birer ifadedir. Tek başına sabit ve tek başına değişken ifade belirtir. Fakat tek başına operatör ifade belirtmez.

Metotların Geri Dönüş Değerleri (return value)

Bir metodu çağırdıktan sonra metodun çalışması bittiğinde metot bize bir değer verebilir. Bu değere metodun geri dönüş değeri (return value) denilmektedir. Metodun geri dönüş değerinin türü metot bildiriminde metot isminin soluna yazılır. Örneğin:

```
public static int Foo()
{
    //...
}
```

Burada Foo metodu int türden bir değer geri vermektedir. Metotların geri dönüş değerleri işlemlere sokulabilir. Örneğin:

```
x = Foo() * 2;
```

Burada önce Foo metodu çağrılmış, bundan bir geri dönüş değeri elde edilmiş, o değer de ikiyle çarpılıp

sonuç x'e atanmıştır.

Metotların geri dönüş değerleri return deyimiyle oluşturulur. return deyiminin genel biçimi şöyledir:

```
return [ifade];
```

Örneğin:

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

			result = Sample.Foo() * 2;
			System.Console.WriteLine(result);
		}
	}

	class Sample
	{
		public static int Foo()
		{
			System.Console.WriteLine("Foo");

			return 100;
		}
	}
}
```

return deyiminin yanında ifade tanımına uyan herhangi bir ifade bulunabilir. Örneğin:

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

			result = Sample.Foo() * 2;
			System.Console.WriteLine(result);
		}

	}


	class Sample
	{
		public static int Foo()
		{
			int a = 10;

			System.Console.WriteLine("Sample.Foo");

			return a + 10;
		}
	}
}
```

return deyiminin iki işlevi vardır:

1) Metodu sonlandırır.

2) Metodun geri dönüş değerini oluşturur.

Metodun geri dönüş değeri yoksa yani metot bize sonlandığında bir değer vermiyorsa bu durumda geri dönüş değerinin türü yerine void anahtar sözcüğü yazılır. void bir metotta return anahtar sözcüğü kullanılabilir fakat yanına ifade yazılamaz. void bir metotta return kullanılmazsa metot ana blok bittiğinde sonlanır. void olmayan metotlarda return anahtar sözcüğünün yanında bir ifade bulunmak zorundadır.

void olmayan metotlarda return kullanılması zorunludur. Aksi taktirde error oluşur. Ayrıca programın tüm mümkün akışlarında return deyiminin görülmesi gerekir.

void metotların geri dönüş değerleri olmadığına göre onları sanki biz geri dönüş değeri varmış gibi kullanamayız. Örneğin Foo void bir metot olsun:

```
result = Foo() * 2;           // error!
```

void metotları sade bir biçimde eaşağıdaki gibi çağırmalıyız:

```
Foo();
```

Bir metodun geri dönüş değerinin olması onu kullanmayı zorunlu hale getirmez. Örneğin Foo metodunun bir geri dönüş değeri olsun:

```
Foo();            // geçerli
```

Geri dönüş değerleri istenirse kullanılır, istenir kullanılmaz.,

Metot çağırmak için kullanılan parantezler de bir operatördür. Dolayısıyla aşağıdaki gibi return ifadesi olabilir:

```
return Bar();
```

Burada Bar netodu çağrılır, onun geri dönüş değeri ile geri dönülür. Örneğin:

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

			result = Sample.Foo();
			System.Console.WriteLine(result);
		}
	}

	class Sample
	{
		public static int Foo()
		{
			System.Console.WriteLine("Sample.Foo");

			return Bar() * 2;
		}

		public static int Bar()
		{
			System.Console.WriteLine("Sample.Bar");

			return 100;
		}
	}
}
```

Metotların Geri Dönüş Değerlerine Neden Gereksinim Duyulmaktadır?

Metotların geri dönüş değerleri metodu çağırana ekstra bir bilgi vermek için kullanımaktadır. Örneğin bir metot parametresiyle aldığı değerin karekökünü geri dönüyor olabilir. Biz bu metodu bir değerle çağırıp geri dönüş değeri olarak onun karekökünü elde edebiliriz. Başka bir metot karışık birtakım işlemler yapıp bize o işlemlerle ilgili ekstra bir bilgiyi geri dönüş değeri olarak veriyor olabilir. Bir başka metot bir işlem yapıp işlemin başarısını bool bir değer olarak bize veriyor olabilir.

Klavyeden Değer Okunması

Klavyeden T türünden bir değer okumak için aşağıdaki kalıp kullanılır:

```
val = T.Parse(System.Console.ReadLine());
```

Örneğin:

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

			System.Console.Write("Bir deger giriniz:");
			val = int.Parse(System.Console.ReadLine());
			System.Console.WriteLine(val * val);
		}
	}
}
```

Sınıf Çalışması: a, b, ve c isimli double türden 3 değişken tanımlayınız. a ve b için klavyeden değer okuyunuz. Bunları çarparak sonucu c'ye yazınız ve c'yi yazdırınız.

Çözüm:

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

			System.Console.Write("Bir deger giriniz:");
			a = int.Parse(System.Console.ReadLine());


			System.Console.Write("Bir deger giriniz:");
			b = int.Parse(System.Console.ReadLine());

			c = a * b;
			System.Console.WriteLine(c);
		}
	}
}
```

Sınıf Çalışması: Sample sınıfının içerisinde Square isimli bir metot yazınız. Metodun geri dönüş değeri double türden olsun. Metot içerisinde klavyeden bir double değer isteyiniz ve o double değerin karesiyle metodu geri döndürünüz. Sonra Main metodundan Square metodunu çağırıp geri dönüş değerini yazdırınız.

Çözüm:

```
namespace CSD
```

```
{
	class App
	{
		public static void Main()
		{
			double result;

			result = Sample.Square();
			System.Console.WriteLine(result);
		}
	}

	class Sample
	{
		public static double Square()
		{
			double d;

			System.Console.Write("Bir deger giriniz:");
			d = double.Parse(System.Console.ReadLine());

			return d * d;
		}
	}
}
```

Metotların Geri Dönüş Değerleri Nasıl Oluşturuluyor?

Metotların geri dönüş değerleri önce geçici bir değişkene aktarılır, oradan alınarak kullanılır. Örneğin:

```
return 100;
```

gibi bir kodda bu 100 değeri geçici bir değişkene aktarılmaktadır. Derleyici tarafından yaratılıp yok edilen değişkenlre geçici değişkenler denilmektedir. Geçici değişkenlerin ismini biz bilemeyiz. Fakat temp olarak varsayarsak,

```
return 100;
```

gibi bir işlem aslında şu anlama gelmektedir:

```
temp = 100;
```

Geri dönüş değeri kullanılmak istendiğinde o geçici değişkenin içerisinden alınarak kullanılır. Kullanım bitince geçici değişken derleyici tarafından yok edilmektedir. Yani örneğin:

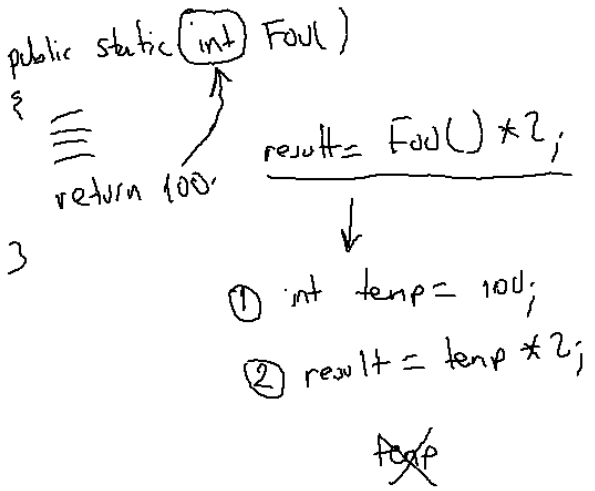
```
return 100;
```

gibi gibi bir return işlemi yapılmış olsun. Metot da şöyle kullanılmış olsun:

```
result = Foo() * 2;
```

Burada şu olaylar gerçekleşmektedir:

1) temp yaratılır
2) temp = 100;
3) result = temp * 2;
4) temp yok edilir.



return işlemi aslında geçici değişkene yapılan bir atama işlemi gibidir. Metodun geri dönüş değerinin türü de aslında bu geçici değişkenin türünü belirtir. Bu nedenle ileride atama işlemi için söyleyeceğimiz her şey return işlemi için de geçerli olacaktır.

Visual Studio IDE'sinin Kullanımı

VisualStudio IDE'siyle bir C# programı aşağıdaki gibi derlenerek çalıştırılır:

1) VisualStudio IDE'si açıldığında karşımıza bir StartPage sayfası gelir. Burada bazı kısa yol işlemleri ve haberler vs. bulunmaktadır. Bu sayfa kapatılabilir.

2) Çalışma için öncelikle bir projenin yaratılması gerekir. Proje yaratmak için "File/New/Project" seçilir. Buradan da "Visual C#/Empty Project" seçilir. Daha sonra "Name" kısmına Proje ismi girilir. "Location" proje dizininin yaratılacağı taban dizini belirtmektedir. Aslında projeler “Solution” denilen kaplar içerisinde bulunmaktadır. Dolayısıyla bir proje yaratılmak istendiğinde aslında aynı zamanda bir solution da yaratılır. İşte “Create directory for solution” çarpılanırsa solution dizini ve proje dizini ayrı ve iç içe yaratılır. Biz bunu çarpılamayacağız. (Ancak IDE'yi kurduğumuzda default olarak bu çarpılanmış gelmektedir.) Çarpılamazsa solution ve proje aynı dizin içerisinde yaratılır.

3) Solution ve projeler ismine “Solution Explorer” denilen bir pencereyle idare edilirler. Bu pencere "View/Solution Explorer" seçeneğiyle, araç çubuğundaki icon'a tıklanarak ya da Ctrl + Alt + L tuşlarıyla çıkartılabilirler. Bu dockable bir penceredir. Yani bu pencereyi biz istediğimiz köşeye yuvalayabiliriz.

4) Şimdi bizim proje içerisine bir kaynak dosya eklememiz gerekir. Bu işlem “Project/Add New Item” menüsüyle yapılabileceği gibi Solution Explorer'da proje üzerine gelinip bağlam menüsünden “Add/New Item” seçeneğiyle de yapılabilir. Burada karşımıza “Add New Item” dialog penceresi çıkar. Bu pencerede "C#/Windows" ve "Code File" seçilir. Sonra dosyay bir isim verilir. Sonra da kod bu dosyanın içerisine yazılır.

5) Derleme işlemi için “Build/Build Solution” ya da “Build/Build XXX” seçilir. Programı çalıştırmak için “Debug/Start Without Debugging” seçilmelidir. Zaten sonraki seçenek seçildiğinde öncekiler de yapılır. Yani aslında tek yapılacak şey “Debug/Start Without Debugging” menüsünü seçmektir. Bunun da kısa yol tuşu Ctrl+F5'tir.

6) Projeyi yeniden açmak için File/open/Project-Solution seçilir. Bizden solution dosyasını seçmemiz istenecektir. Biz de proje dizinine gelerek .sln uzantılı solution dosyasını seçeriz. Aslında Visual Studio'yu hiç çalıştırmadan ilgili dizine gelip .sln dosyasını seçersek bu uzantı ilişkilendirildiği için zaten Visual Studio tarafından açılacaktır.

Sabitler

Program içrisinde doğrudan yazılan sayılara sabit denir. Yalnızca değişkenlerin değil sabitlerin de türleri vardır. Örneğin:

```
100
200
0
```

ifadeleri birer belirtir. Negatif sayılar bu anlamada sabit belirtmezler. Çünkü negatif sayıları yazmak için kullandığımız '-' sembolü operatör görevindedir.

C#'ta sabit oluşturma kuralları şöyledir:

1) Sayı nokta içermiyorsa ve sayının sonunda hiçbir ek yoksa, sabit sırasıyla int, uint, long ve ulong türlerinin hangisinin sınırları içerisinde ilk kez kalıyorsa o türdendir. Sayı ulong sınırlarını da aşıyorsa error oluşur. Örneğin:

```
0  ---> int
100 ---> int
3000000000 --> uint (int sınırını aşmış)
5000000000 --> long (uint sınırını da aşmış)
500000000000000000000000000000 (error, ulong sınırını da aşmış)
```

2) Sayı nokta içermiyorsa ve sayının sonunda küçük harf ya da büyük hraf l (el) varsa sabit long ve ulong türlerinin hangisinin sınırları içerisinde ilk kez kalıyorsa o türdendir. Sayı ulong sınırlarını da aşıyorsa error oluşur. Örneğin:

```
100L --> long
0l ---> long
```

3) Sayı nokta içermiyosa ve sayının sonunda küçük harf ya da büyük harf u varsa sabit uint ve ulong türlerinin hangisinin sınırları içerisind eilk kez kalıyorsa o türdendir. Sayı ulong sınırlarını da aşıyorsa error oluşur. Örneğin:

```
100u ---> uint
5000000000u ---> ulong
```

4) Sayı nokta içermiyorsa ve sayının sonunda büyük harf ya da küçük harf ul ya da lu varsa (örneğin Ul, Lu, LU,UL gibi) sabit ulong türdendir. Sayı ulong sınırlarını da aşıyorsa error oluşur. Örneğin:

```
1ul ---> ulong
100UL ---> ulong
```

5) Sayı nokta içeriyorsa ve sonunda ek yoksa ya da sayı nokta içerisin veya içermesin sonunda küçük harf ya da büyük harf d varsa sabit double türündendir.

```
3.2 ---> double
3D ---> double
4.3d ---> double
```

6) Sayı nokta içermesin ya da içermesin sayının sonunda küçük harf ya da büyük harf f varsa sabit float türdendir. Örneğin:

```
10F --> float
20.3f --> float
```

7) Sayı nokta içersin ya da içrmesin sayısının sonunda küçük harf ya da büyük harf m varsa sabit decimal türdendir. Örneğin:

```
10m --> decimal
100.3M --> decimal
```

8) Bir karakter tek tırnak içerisine alınırsa o artık bir sayı belirtir hale gelir. Tek tırnak içerisindeki karakterler UNICODE tabloda sıra numarası belirtmektedir. Böyle elde edilen sabitler char türdendir. Örneğin:

```
'a' ---> char türden sabit değeri 97
'1' --> char türdensabit değeri 49
'?' ---> char türdn sabit
```

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            char ch = 'a';
            int result;

            result = ch + 1;
            System.Console.WriteLine(result);
        }
    }
}
```

Tek tırnak içerisine yalnızca tek bir karakter yerleştirilebilir.

Bazı karakterlerin görüntü karşılığı yoktur. Biz bunları ekrana yazdırmak istediğimizde birşey göremeyiz. Fakat bunlar bazı olaylara yol açarlar. Bu karakterlerin görüntü karşılığı olmadığı için biz bunları tek tırnak içerisine alamayız. İşte bazı görüntülenemeyen (non-printable) karakterleri ifade edebilmek için bir özel yöntem oluşturulmuştur. Tek tırnak içerisinde önce bir ters bölü, sonra bazı özel karakterler bazı görüntülenemeyen karakterleri temsil etmektedir. Bunların listesi şöyledir:

```
'\a' ---> alert (yazdırılmaya çalışılırsa ses çıkar)
'\b' ---> back space (yazdırılmaya çalışılırsa imleç sola gider ve soldaki karakter silinir)
'\f' ---> form feed (yazdırılmaya çalışılırsa imleç bir sayfa atlar)
'\n' ---> new line (yazdırılmaya çalışılırsa imleç aşağı satırın başına geçer)
'\r' ---> carriage return (yazdırılmaya çalışılırsa imleç bulunduğu satırın başına geçer)
'\t' ---> tab (yazdırılmaya çalışılırsa imleç bir tab atar)
‘\v’ ---> vertical tab (düşey tab işlemi yapar)
```

Ters bölü karakterlerini iki tırnak içerisinde de aynı anlamda kullanabiliriz. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            System.Console.WriteLine("ali\nveli\tselami");
        }
    }
}
```

Aşağıdaki iki çağrı tamamen aynı etkiye yol açar:

```
System.Console.WriteLine("ali");
System.Console.Write("ali\n");
```

Tek tırnak karakterinin kendisine ilişkin karakter sabiti '\'' biçiminde belirtilir. Ancak iki tırnak içerisinde tek tırnak soruna yol açmaz. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            System.Console.WriteLine("ali'nin evi");        // geçerli
            System.Console.WriteLine("ali\'nin evi");       // geçerli
            System.Console.WriteLine('\'');                 // geçerli
```

```
        }
    }
}
```

Tek tırnak içerisinde iki tırnak soruna yol açmaz. Ancak iki tırnak içerisinde iki tırnak kullanamayız. İki tırnak karakteri hem tek tırnak içerisinde hem de iki tırnak içerisinde \" biçiminde belirtilebilir. Örneğin:

```
System.Console.WriteLine("\"Ankara\"");
```

Ters bölü karakterinin kendisi '\\' ile belirtilir. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            System.Console.WriteLine("c:\temp\a.dat");
            System.Console.WriteLine("c:\\temp\\a.dat");
        }
    }
}
```

9) bool türden iki sabit vardır: true ve false

10) C#'ta byte, sbyte, short ve ushort türden sabit kavramı yoktur. Yani char türünü bir yana bırakırsak C#'ta int türünden küçük türlere ilişkin sabit kavramı yoktur.

Gerçek Sayıların Üstel Biçimde İfade Edilmesi

C#'ta çok büyük ve çok küçük sayıları kolay yazmak için üstel biçim de kullanılabilmektedir. Üstel biçimin genel biçimi şöyledir:

```
<sayı><e/E>[+/-]<üstel kısım>
```

Örneğin:

```
1e10
2.3e-20
123.23E+16
```

Yine bu sayıların sonunda hiçbir ek yoksa sabitler double türdendir. Tabi biz üstel biçimdeki sabitlerin float türünden olmasını sonuna büyük harf ya da küçük harf 'f' getirerek sağlayabiliriz:

```
2.3e5F
1E4F
```

Sayı üstel biçimde yazılmışsa zaten türü hiçbir zaman tamsayı türlerine ilişkin olamaz. Örneğin 1e3 sayısı 1000 anlamına geliyor olsa da double türdendir. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            double d = 1e10;
            System.Console.WriteLine(d);
        }
    }
}
```

Anahtar Notlar: C#'ta tamsayılar 0x ile başlatılarak 16'lık sistemde belirtilebilirler. Ancak C#'ta 8'lik sistemde (oc tal system) bir gösterim yoktur. Halbuki C'de sayının başına 0 getirilirse sayının 8'lik sistemde yazıldığı anlaşılmaktadır. Örneğin:

```
a = 012;              // Dikkat Ocatal 12 değil, decimal 12
```

Anahtar Notlar: Fortran zamanından beri gelen bir gelenek C ve C++'ta devam ettirilmiştir. Gerçek sayılarda noktanın sağı ya da solu boş bırakılırsa orada sıfır olduğu varsayılmaktadır. C#'ta noktanın solunu boş bırakabiliriz. Ancak sağını boş bırakamayız. Örneğin:

```
x = .12;              // C, C++, Java ve C#'ta geçerli
x = 12.;              // C, C++ ve Java'da geçerli fakat C#'ta geçerli değil
```

C#'ta noktanın sağının boş bırakılamamasının nedeni temel türlerin de bir yapı olmasından kaynaklanmaktadır. Nokta aynı zamanda yapılarda elemana erişim operatörü olarak kullanılmaktadır.

Metotların Parametre Değişkenleri

Metotlar parametre değişkenleri yoluyla dış dünyadan değerler alıp onları kullanabilirler. Parametre değişkenleri metot bildiriminde parametre parantezinin içerisinde aralarına ',' atomu kullanılarak bildirilmektedir. Örneğin:

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

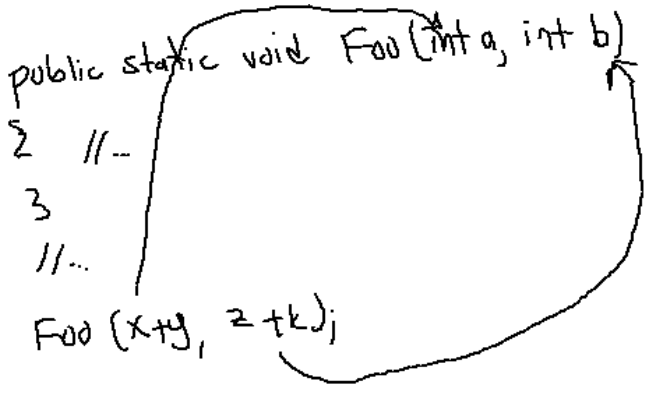
Burada metodun iki parametre değişkeni vardır. Birincisi int türden ikincisi long türdendir. Parametre değişkenleri aynı türden olsa bile tür belirten anahtar sözcük her defasında yazılmak zorundadır. Örneğin:

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

Parametreli bir metot çağrılırken parametre sayısı kadar argüman girilmek zorundadır. Argümanlar ',' atomu ile birbirlerinden ayrılırlar. Argüman olarak herhangi geçerli bir ifade kullanılabilir. Örneğin:

```
Foo(x, y);
Foo(10, 20);
Foo(a + 10, b - 20);
...
```

Parametreli bir metot çağrıldığında önce argümanların değerleri hesaplanır, sonra argümanlardan parametre değişkenlerine karşılıklı bir atama yapılır. Sonra da programın akışı metoda aktarılır.



Örneğin:

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

```
            int x = 10, y = 20;

            Sample.Foo(x, y);
            Sample.Foo(x + 10, y - 5);
            Sample.Foo(100, 200);
        }
    }

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

Parametre değişkenleri ilkdeğerlerini metot çağrılırken alırlar. Fakat bunlara biz daha sonra değer atayabiliriz. Tabi bunlara değer atamamız çağrılan ifadedeki değişkenleri etkilemez. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int x = 10, y = 20;

            Sample.Foo(x, y);

            System.Console.Write(x);
            System.Console.Write(", ");
            System.Console.WriteLine(y);
        }
    }

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

            a = 100;
            b = 200;

            System.Console.Write(a);
            System.Console.Write(", ");
            System.Console.WriteLine(b);
        }
    }
}
```

Metotların parametre değişkenleri yalnızca o metotta tanınabilmektedir. Bunlar faaliyet alanı bakımından sanki metodun ana bloğunun başında tanımlanmış değişkenler gibi değerlendirilirler. Böylece iki metodun parametre değişkenleri aynı isimde olsa bile bunlar birbirlerine karışmazlar. Bunlar aslında farklı değişkenleri belirtirler. Örneğin:

```
public static void Foo(int a)    // sorun yok
{
      //...
}

public static void Bar(int a)    // sorun yok
{
```

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

Fakat bir parametre değişkeni ile aynı isimli o metotta başka bir yerel değişken bildirilemez. Örneğin:

```
public static void Foo(int a)
{
	//...
	{
		long a;			// error!
		//...
	}
	//...
}
```

Anahtar Notlar: Parametre (parameter) ve argüman (argument) terimleri farklı anlamlarda kullanılmaktadır. Parametre denildiğinde metotların parametre değişkenleri anlaşılır. Argüman denildiğinde metotları çağırırken yazdığımız ifadeler anlaşılır.

Bazen metotlar bizden parametre yoluyla birtakım değerleri alır, onlar üzerinde birtakım işlemleri gerçekleştirir ve bir değerle geri dönerler. Örneğin:

```
public static int Add(int a, int b)
{
	return a + b;
}
```

Örneğin:

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

            result = MathOp.Add(10, 20);
            System.Console.WriteLine(result);

            result = MathOp.Mul(10, 20);
            System.Console.WriteLine(result);
        }
    }

    class MathOp
    {
        public static double Add(int a, int b)
        {
            return a + b;
        }

        public static double Mul(int a, int b)
        {
            return a * b;
        }
    }
}
```

Örneğin:

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

```
            double result;

            result = Sample.Div(2, 10);
            System.Console.WriteLine(result);

            result = Sample.Square(10);
            System.Console.WriteLine(result);
        }
    }

    class Sample
    {
        public static double Div(double a, double b)
        {
            return a / b;
        }

        public static double Square(double a)
        {
            return a * a;
        }
    }
}
```

Anahtar Notlar: Visual Studio IDE'sinde programı satır satır çalıştımak için F10 ve F11 tuşları kullanılır. F11 tuşu metotların içerisine girerek çalıştırma yapar. F10 ise metodun içerisine girmeden çalıştırır. Yani F10 tuşu metodu yine çağırmaktadır fakat bizi içeriye sokmaz. Kodun geri kalanını hızlı bir biçimde çalıştırmak için F5 tuşu kullanılır. Programı belli bir noktaya kadar çalıştırdıktan sonra o noktadan itibaren adım adım çalıştırmak isteyebiliriz. Bunu sağlamak için ilgili satıra "kırma noktası (break point)" yerleştirilir. Program F5 ile çalıkştırılırken kırma noktasına kadar tam sürat çalışır. Kırma noktasına gelindiğinde orada durur. Biz F10 ve F11 ile devam edebiliriz. Visual Studio'da kırma noktası F9 tuşuyla yerleştirilip kaldırılabilir.

Bazı Matematiksel Metotlar

System isim alanı içerisindeki Math isimli sınıfın static metotları önemli bazı matematiksel işlemleri yapmaktadır. Bunlardan bazılarını gözden geçirelim:

- Sqrt metodu double bir değerin karekökünü alıp bize onu geri dönüş değeri olarak verir. Parametrik yapısı şöyledir:

```
public static double Sqrt(double val)
```

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            double val, result;

            System.Console.Write("Lutfen bir sayı giriniz:");
            val = double.Parse(System.Console.ReadLine());

            result = System.Math.Sqrt(val);
            System.Console.WriteLine(result);
        }
    }
 }
```

- Pow isimli metot kuvvet alma işlemi yapar. Metodun iki double türden parametresi vardır. Metot birinci parametresiyle belirtilen değerin ikinci parametresiyle belirtilen kuvvetini alır ve geri dönüş değeri olarak verir.

```
public static double Pow(double a, double b)
```

Örneğin:

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

            result = System.Math.Pow(2, 10);
            System.Console.WriteLine(result);       // 2 üzeri 10 = 1024
        }
    }
 }
```

- Sin, Cos, Tan, ASin, ACos, ATan trigonometrik işlemleri yapar. Bunlar double parametre alıp double değer verirler. Parametre radyan cinsinden açı belirtir.

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

            result = System.Math.Sin(3.14159265358979323846264643 / 6);
            System.Console.WriteLine(result);
        }
    }
 }
```

- Math sınıfının Log metodu e tabanına göre logaritma, Log10 metodu 10 tabanına göre logartima hesaplar. Örneğin:

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

            result = System.Math.Log10(1000);
            System.Console.WriteLine(result);       // 3

            result = System.Math.Log(1000);
            System.Console.WriteLine(result);
        }
    }
 }
```

Sqrt kullanımına bir örnek:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            double a, b, c;

            System.Console.Write("a:");
            a = double.Parse(System.Console.ReadLine());

            System.Console.Write("b:");
```

```
            b = double.Parse(System.Console.ReadLine());

            System.Console.Write("c:");
            c = double.Parse(System.Console.ReadLine());

            Root.DispRoots(a, b, c);
        }
    }

    class Root
    {
        public static void DispRoots(double a, double b, double c)
        {
            double delta;
            double x1, x2;

            delta = b * b - 4 * a * c;
            x1 = (-b + System.Math.Sqrt(delta)) / (2 * a);
            x2 = (-b - System.Math.Sqrt(delta)) / (2 * a);

            System.Console.WriteLine(x1);
            System.Console.WriteLine(x2);
        }
    }
}
```

Operatörler

Bir işleme yol açan ve işlem sonucunda belli bir değerin üretilmesini sağlayan atomlara operatör denir. Operatörler üç biçimnde sınıflandırılabilirler:

1) İşlevlerine Göre Sınıflandırma
2) Operand Sayılarına Göre Sınıflandırma
3) Operatörün Konumuna Göre Sınıflandırma

1) İşlevlerine Göre Sınıflandırma

Bu sınıflandırma operatörün hangi amaçla kullanıldığına yöneliktir. Tipik sınıflandırma şöyledir:

- Artimetik Operatörler (Arithmetic operators): Bunlar tipik toplama, çarpma, bölme gibi işlemleri yapan operatörlerdir.

- Karşılaştırma Operatörleri (Comparison operators): Bunlar karşılaştırma amacıyla kullanılan operatörlerdir. Örneğin >, <, >=, <= gibi...

- Mantıksal Operatörler (Logical Operators): Bunlar AND, OR ve NOT gibi mantıksal işlem yapan operatörlerdir.

- Bit Operatörleri (Bitwise Operators): Bunlar sayıların karşılıklı bitleri üzerinde işlem yapan operatörlerdir.

- Özel Amaçlı Operatörler (Special Purpose Operators): Belli bir konuya ilişkin işlem yapan, diğer gruplara girmeyen operatörlerdir.

2) Operand Sayılarına Göre Sınıflandırma

Operatörün işleme soktuğu ifadelere operand denir. Örneğin a + b ifadesinde + operatördür, a ve b onun operandlarıdır. Operand sayılarına göre operatörler üçe ayrılmaktadır:

- Tek opeandlı Operatörler (Unary Operators): Bunlar tek operand alırlar. Örneğin !a ifadesinde ! operatördür a bunun operandıdır.

- İki Operandlı Operatörler (Binary Operators): Bunlar iki operand alırlar. Örneğin a * b işleminde * operatördür. a ve b bunun iki operandıdır.

- Üç Opeandlı Operatörler (Ternary Operators): C#'ta yalnızca bir tane üç operand alan operatör vardır. a ? b : c buna örnek verilebilir.

3) Operatörün Konumuna Göre Sınıflandırma

Bu sınıflandırma operatörün operandlarının neresine yerleştirilebileceğine göre yapılmaktadır. Operatör operandlarının önüne, sonuna ya da arasına getirilebilir:

-Önek Operatörler (Prefix Operators): Bu operatörlerde operand operatörünün önüne getirilir. Örneğin !a ifadesinde ! operatördür a ise bunun operandıdır.

- Araek Operatörler (Infix Operators): Bu operatörlerde operatör operandlarının arasına getirilmek zorundadır. Örneğin a * b ifadesinde * operandların arasına getirilmiştir.

- Sonek Operatörler (Postfix Operators): Burada operatör operandlarının sonuna getirilir. Örneğin a++ ifadesinde ++ operatörü operandının sonuna getirilmiştir.

Bir operatörü teknik olarak tanımlamak için bu üç sınıflandırmada da nereye dönüştüğünü belirtmek gerekir. Örneğin "* operatörü iki operandlı araek (binary infix) bir artitmetik operatördür" ya da "! operatörü tek opeandlı önek (unary prefix) bir mantıksal operatördür" gibi.

Opeatörler Arasındaki Öncelik İlişkisi (Operator Precedency)

Operatörler arasında bir öncelik ilişkisi vardır. İşlemci makine komutlarını sırasıyla çalıştırır. Derleyicimiz ifadeleri makine komutlarına dönüştürür (C#'ta ara koda) bunlar da sırasıyla işlemci tarafından yapılmaktadır. Örneğin:

```
a = b + c * d;

İ1: c * d
İ2: b + İ1
İ3: a = İ2
```

Operatörler arasındaki öncelik ilişkisi ismine "Operatörlerin Öncelik Tablosu" denilen bir tabloyla betimlenmiştir. Öncelik tablosunun birkaç operatörü içine alan basit bir örneği şöyle verilebilir:

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

Öncelik tablosu satırlardan oluşmaktadır. Üst satırdaki operatörler alt satırdaki operatörlerden daha önceliklidir. Aynı satırdaki operatörler ise eşit önceliklidir. Ancak aynı satırdaki operatörlerde o satırın sağında "Soldan-Sağa" yazıyorsa ifade içerinde operatör solda ise onun öceliği vardır. "Sağdan-Sola" ise "ifade içrisinde o grupta sağda olan önce yapılır" anlamına gelir. Örneğin:

a = b / c * d;

Burada / ile * soldan-sağa eşit önceliklidir. İfade içerisinde (tabloda değil) / solda olduğu için o önce yapılacaktır:

```
İ1: b / c
İ2: İ1 * d
İ3: a = İ2
```

En üst satırdaki parantezler hem metot çağırma operatörünü hem de öncelik parantezini beliritir. Örneğin:

```
a = Foo() * Bar();
```

```
İ1: Foo()
İ2: Bar()
İ3: İ1 * İ2
İ4: a = İ3
```

***, /, + ve - Operatörleri**

Bunlar iki operandlı araek (binary infix) aritmetik operatörlerdir. Buınlar klasik dört ilmei yaparlar.

% Operatörü

Bu operatör iki operandlı araek bir aritmektik operatördür. Sol tarafındaki operandın sağ tarafındaki operanda bölümünden elde edilen kalan değerini üretir. Örneğin:

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

            result = 20 % 3;
            System.Console.WriteLine(result);       // 2
        }
    }
 }
```

% operatörü öncelik tablosunda * ve / ile soldan sağa aynı düzeyde bulunmaktadır:

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

Örneğin:

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

            result = 20 % 3 * 2;
            System.Console.WriteLine(result);       // 4
        }
    }
}
```

% opeatöründe operandlar negatif olabilir. Örneğin -20 % 3 bize -2 değerini verir. Opeandlar gerçek sayı türlerine ilişkin de olabilirler. % operatörü “tek mi çift mi testi” için, “tam bölünüyor mu” testi için sık kullanılmaktadır.

Sınıf Çalışması: Klavyeden int bir sayı isteyiniz. Sayının kendisine en yakın fakat ondan küçük eşit olan 4'ün katını yazdırınız. Örneğin 15 için 12, 9 için 8, 6 için 4, 8 için 8 yazdırılmalıdır.

Çözüm:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int val, result;

            System.Console.Write("Bir sayı giriniz:");
            val = int.Parse(System.Console.ReadLine());

            result = val - val % 4;
            System.Console.WriteLine(result);
        }
    }
}
```

İşaret + ve İşaret - Operatörleri

İşaret + ve işaret - operatörleri tek operandlı önek aritmetik operatörlerdir. Bunların aynı sembolle gösteriliyor olsa da toplama ve çıkartma operatörüyle bir ilgisi yoktur. Örneğin:

```
a = -b - c;
```

işleminde ilk - işaret - operatörü diğeri çıkartma operatörüdür. Bu operatörler öncelik tablosounun ikinci düzeyinde bulunurlar:

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

İşaret - operatörü operandının negatif değerini üretir. İşaret + operatörü operandıyla aynı değri üretir (yani aslında birşey yapmaz.) Aşağıdaki örnekteki ilk - çıkartma operatörü diğerleri işaret - operatörleridir:

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

            result = 8 - - - - - - - - - - - - - - - 5;
            System.Console.WriteLine(result);
        }
    }
}
```

++ ve -- Operatörleri

++ operatörüne artırma (increment), -- operatörüne de eksiltme (decrement) operatörü denilmektedir. Bu operatörler tek operandlı önek ya da sonek kullanılabilen operatörlerdir. ++ operatörü operandıyle belirtilen değişkendeki değeri 1 artırır, -- operatörü de 1 eksiltir. Örneğin:

```
int a = 10;

--a;  // a = 9
```

++ ve -- operatörleri öncelik tablosunun ikinci süzeyinde sağdan sola grupta bulunur:

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

Bu operatörlerin önek ve sonek kullanımları arasında bir fark vardır. Her iki kullanımda da artırım ya da eksiltim tabloda belirtilen sırada yapılır. Önek kullanımda sonraki işleme artırılmış ya da eksiltilmiş değer sokulurken, sonek kullanımda artırılmamış ya da eksiltilmemiş değer sokulur. Örneğin:

```
int a = 3;
int b;

b = ++a * 3;

/* a = 4, b = 12 */
```

Burada a önce artılıp 4 olacak. Sonraki işleme a'nın artırılmış değeri olan 4 sokulacak. Örneğin:

```
int a = 3;
int b;

b = a++ * 3;

/* a = 4, b = 9 */
```

Burada a artılıp 4 olacak. Sonraki işleme a'nın artırılmamış değeri olan 3 sokulacak. Örneğin:

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

            a = 3;
            b = 2;

            c = ++a * b--;

            System.Console.WriteLine(a);        // 4
            System.Console.WriteLine(b);        // 1
            System.Console.WriteLine(c);        // 8

        }
    }
}
```

Örneğin:

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

```
            a = 3;
            b = ++a;

            System.Console.WriteLine(a);        // 4
            System.Console.WriteLine(b);        // 4

            a = 3;
            b = a++;

            System.Console.WriteLine(a);        // 4
            System.Console.WriteLine(b);        // 3
        }
    }
}
```

Şüphesiz bu operatörler tek başlarına kullanıldığında bunların önek ve sonek kullanımları arasıında bir fark oluşmaz. Yani örneğin:

++a;

ile,

a++;

arasında bir fark yoktur. Fark başka operatörler de varsa oluşur.

++ ve -- operatörlerinin operandları değişken olmak zorundadır.

C'de C++'ta bir değişken bir ifadede ++ ve -- operatörleriyle kullanımışsa artık o değişkenin aynı ifadede bir daha gözükmemesi gerekir. Aksi halde tanımsız davranış (undefined behavior) oluşur. Halbuki C#'ta bu durum tamamen tanımlanmıştır. Örneğin aşağıdaki işlem C#'ta geçerli ve tanımlıdır. Fakat tabi böylesi karışık ifadelerden kaçınmak gerekir:

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

a = 3;
b = ++a + a;
// a = 4, b = 8

a = 3;
b = a++ + a;
// a = 4, b = 7
```

Metot çağrısında argümnanda bir değişken ++ ve -- operatörleriyle kullanılmışsa parametre değişkenine onun artırım ya da eksiltimden önceki veya sonraki değeri atanabilir. Örneğin:

```
Foo(++a);
```

Burada a bir artırılır. Foo metoduna a'nın artırılmış değeri gönderilir. Fakat:

```
Foo(a++);
```

Burada a bir artırılır. Foo metoduna a'nın artırılmamış değeri gönderilir.

Karşılaştırma Operatörleri

C#'ta 6 karşılaştırma operatörü vardır:

<, >, <=, >=, ==, !=

“Eşit mi” sorusunun sorulduğu karşılaştırma operatörünün == ile, “eşit değil mi” aorusunun sorulduğu karşılaştırma operatörünün != ile temsil edildiğine dikkat ediniz. Öncelik tablosunda karşılaştırma operatörleri aritmetik operatörlerden daha düşük önceliklidir.

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

Örneğin:

```
result = a + b > c + d;

İ1: a + b
İ2: c + d
İ3: İ1 > İ2
İ4: result = İ3
```

Karşılaştırma operatörleri bool türden değer üretirler. Önerme doğruysa true, yanlışsa false değeri elde edilir. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int val;
            bool result;

            System.Console.WriteLine("Bir sayı giriniz:");
            val = int.Parse(System.Console.ReadLine());

            result = val % 2 == 0;
            System.Console.WriteLine(result);
        }
    }
}
```

Mantıksal Operatörler

C#'ta üç mantıksal operatör vardır:

```
! (NOT)
&& (AND)
|| (OR)
```

&& ve || operatörleri iki operandlı araek operatörlerdir. ! operatörü ise tek operandlı önek bir operatördür. Mantıksal operatörlerin operand'ları bool türden olmak zorundadır. Bu operatörler yine bool türden değer üretirler.

A	B	A && B	A || B
true	true	true	true

true	false	false	true
false	true	false	true
false	false	false	false

A	!A
true	false
false	true

! operatörü öncelik tablosunun ikinci düzeyinde sağdan-sola grupta bvulunur. && ve || operatörleri ise karşılaştırma operatörlerinden daha düşük önceliklidir:

( )	Soldan-Sağa
+ - ++ -- !	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
> < >= <=	Soldan-Sağa
== !=	Soldan-Sağa
&&	Soldan-Sağa
||	Soldan-Sağa
=	Sağdan-Sola

Karşılaştırma operatörleri bool değer ürettiği için genellikle mantıksal operatörler bunların çıktıları üzerinde işlem yapar. Örneğin:

```
result = a >= 10 && a =< 20;
```

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int val;
            bool result;

            System.Console.Write ("Bir sayı giriniz:");
            val = int.Parse(System.Console.ReadLine());

            result = val >= 10 && val <= 20;
            System.Console.WriteLine(result);
        }
    }
}
```

Örneğin:

De Morgan kuralına göre,

```
!(a && b)
```

ile,

```
!a || !b
```

eşdeğerdir. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int val;
            bool result;

            System.Console.Write("Bir sayı giriniz:");
            val = int.Parse(System.Console.ReadLine());

            result = !(val < 10 || val > 20);
            System.Console.WriteLine(result);
        }
    }
}
```

Örneğin:

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

            result = !!!!!!!!!!true;
            System.Console.WriteLine(result);
        }
    }
}
```

Aslında && ve || operatörleri klasik öncelik kuralına uymazlar. Bu operatörlerin kısa devre özelliği (short circuit) vardır. Bu özelliğe göre && ve || operatörlerinin sağ tarafında hangi öncelikli operatör olursa olsun önce sol tarafındaki ifade yapılır ve bitirilir. Duruma göre sağındaki ifade hiç yapılmaz. (&& operatöründe soldaki ifade false ise, || operatöründe soldaki ifade true ise sağ taraftaki ifade yapılmaz). Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int val;
            bool result;

            System.Console.Write("Bir sayı giriniz:");
            val = int.Parse(System.Console.ReadLine());

            result = val >= 0 || Foo();
            System.Console.WriteLine(result);
        }

        public static bool Foo()
        {
            System.Console.WriteLine("Foo");

            return false;
        }
```

```
    }
}
```

Örneğin:

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

            result = Foo() && Bar();
            System.Console.WriteLine(result);
        }

        public static bool Foo()
        {
            System.Console.WriteLine("Foo");

            return false;
        }

        public static bool Bar()
        {
            System.Console.WriteLine("Bar");

            return true;
        }

    }
}
```

&& ve || operatörleri bir arada kullanılırken soldaki operatörün sol tarafı (yani en soldaki ifade) önce yapılır. Örneğin:

```
result = Foo() || Bar() && Tar();
```

Burada Foo true ise başka birşey yapılmaz. Foo false ise Bar yapılır. Bar da false ise Tar yapılmaz. Örneğin:

```
result = Foo() && Bar() || Tar();
```

Burada önce Foo yapılır. Foo false ise Bar yapılmaz ama Tar yapılır. Foo true ise Bar yapılır. Bar da true ise Tar yapılmaz.

Kısa devre özelliği öncelik tablosunda belirtilen sırada işlemlerin yapılmasıyla farklı bir sonucun çıkmasına yol açmamaktadır. Yalnızca sonucun daha çabul elde edilmesine yol açmaktadır. Yani “kısa devre özelliğiyle elde edilen sonuç” "kısa devre özelliği olmasaydı" durumunda elde edilen sonuçla aynıdır. Örneğin:

```
result = Foo() || Bar() && Tar();
```

işleminde aslında Bar ile Tar'ın && işleminin sonucu Foo ile Or'lanmaktadır. Fakat bu sonuç önce Foo'nun yapılmasıyla daha çabuk elde edilir.

Bu konuyu şöyle de değerlendirebiliriz. Önce sanki hiç kısa devre özelliği yokmuş gibi işlem adımlarına yazalım:

```
result = Foo() && Bar() || Tar();

İ1: Foo() && Bar()
İ2: İ1 || Tar()
İ3: result = İ2
```

Burada Tar() ile İ1 || işlemine sokulmaktadır. || işleminde önce onun sol tarafıolan İ1 yapılacaktır. İ1 işleminde de onun sol tarafı olan Foo() işlemi yapılacaktır. O halde en önce Foo() yapılır. Foo() false ise BAr() yapılmaz fakat Tar() yapılır. Şimdi diğer örneğe bakalım:

```
result = Foo() || Bar() && Tar();
```

Burada kısa devre özelliği olmasaydı işlemler şu sırada yapıacaktı:

```
İ1: Bar() && Tar()
İ2: Foo() || İ1
İ3: result = İ2
```

Burada hem || operatörünün hem de && operatörünün sol tarafı önce yapılacaktır. Bunun tek yolu Foo()'nun yapılmasıdır. Eğer Foo() turue ise İ1 hiç yapılmaz. Fakat Foo() false ise Bar() yapılır. Bar() da false ise Tar() yapılmaz.

Ayrıca C#'ta & ve | biçiminde Bit AND ve Bit OR operatörleri de vardır. Bu operatörler her ne kadar aslında sayıların karşılıklı bitlerini AND'lemek ve OR'lamak için kullanılıyorsa da bool operandlarla da çalışabilmektedir. Böylece & ve | sanki && ve || gibi de kullanılabilmektedir. Fakat bir farkla: & ve | operatörlerinin kısa devre özelliği yoktur. Yani özetle C#'ta & operatörü && operatörünün kısa devre özelliği olmayan versiyonu olarak, | operatörü de || operatörünün kısa devre özelliği olmayan versiyonu olarak kullanılabilir. Örneğin:

```
result = Foo() | Bar() & Tar();
```

Burada Foo, Bar ve Tar işlemleri yapılır. Çünkü kısa devre özelliği yoktur. Örneğin:

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

            result = Foo() | Bar() & Tar();
            System.Console.WriteLine(result);
        }

        public static bool Foo()
        {
            System.Console.WriteLine("Foo");

            return true;
        }

        public static bool Bar()
        {
            System.Console.WriteLine("Bar");

            return false;
        }

        public static bool Tar()
        {
            System.Console.WriteLine("Tar");

            return false;
        }
    }
}
```

Atama Operatörü (Assignment Operator)

Atama operatörü iki operandlı araek özel amaçlı operatördür. Bu operatör sağdaki ifadenin değerini soldaki değişkene atamakta kullanılır. Atama operatörü de bir değer üretir. Atama operatörünün ürettiği değer soldaki değişkene atanmış olan değerdir. Biz onu diğer işlemlerde kullanabiliriz. Atama operatörü öncelik tablosunun en düşük öncelikli sağdan sola grubunda bulunmaktadır.

( )	Soldan-Sağa
+ - ++ -- !	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
> < >= <=	Soldan-Sağa
== !=	Soldan-Sağa
&&	Soldan-Sağa
||	Soldan-Sağa
=	Sağdan-Sola

Örneğin:

```
a = b = 10;

İ1: b = 10  => 10
İ2 = İ1
```

Burada hem a'ya hem b'ye 10 atanmış oluyor. Örneğin:

```
Foo(a = 10);
```

Burada a'ya 10 atanıyor. Aynı zamanda metot da bu 10 değeri ile çağrılıyor. Örneğin:

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

İ1: b = 10
İ2: İ1 + 20
İ3: a = İ2
```

Örneğin:

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

            a = b = 10;
            System.Console.WriteLine(b);        // 10
            System.Console.WriteLine(a);        // 10


            a = (b = 10) + 20;
            System.Console.WriteLine(b);        // 10
            System.Console.WriteLine(a);        // 30
        }
    }
}
```

Atama operatörünün sol tarafındaki operandın değişken olması gerekir. Örneğin:

```
a + b = 10;
```

ifadesi error oluşturur.

Bileşik Atama Operatörleri

Bazen bir değişkeni bir işleme sokup sonucu yine aynı değişkene atamak isteriz. Örneğin:

```
a = a + 2;
b = b * 3;
c = c / 10;
```

İşte bu işlemleri biz bileşik atama operatörleriyle yapabiliriz. +=, -=, *=, /=, %= operatörleri iki operandlı araek operaörlerdir. a <op>= b ifadesi tamamen a = a <op> b ile eşdeğerdir. Örneğin:

```
a = a + 2;
```

ile,

```
a += 2;
```

eşdeğerdir. Örneğin:

```
b = b * 3;
```

ile,

```
b *= 3;
```

eşdeğerdir. Bileşik atama operatörleri öncelik tablosunun en düşük düzeyinde atama operatörüyle sağdan sola aynı grupta bulunurlar:

( )	Soldan-Sağa
+ - ++ -- !	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
> < >= <=	Soldan-Sağa
== !=	Soldan-Sağa
&&	Soldan-Sağa
||	Soldan-Sağa
=, +=, -=, *=, /=, %=	Sağdan-Sola

Örneğin:

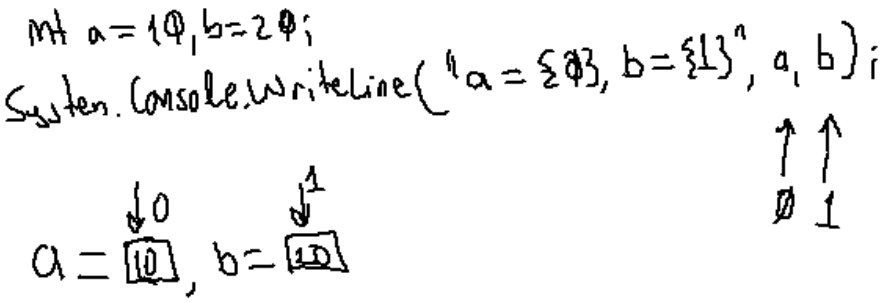
a *= 2 + 3;

İ1: 2 + 3
İ2: a = a * İ1

Bu operatörler de sol taraftaki atamanın yapıldığı değişken içerisinde bulunan değeri üretirler. Bu değer tıpkı atama operaötünde olduğu gibi işleme sokulabilir.

Console Sınıfının Write ve WriteLine Metotlarının Parametrik Kullanımı

Console sınıfının Write ve WriteLine metotları iki tırnak içerisindeki karakterleri tek tek ekrana yadırırlar. Fakat burada {n} biçiminde bir kalıp gördüklerinde bunu ekrana yazdırmazlar. Bunlar yer tutucudur. Bunların yerine n'inci indeksteki argümanın değeri yazdırılır. İki tırnaktan sonraki her argümanın ilki sıfır olmak üzere bir indeks numarası vardır. Örneğin:



Örneğin:

```
int a = 10, b = 20;

System.Console.WriteLine("a = {1}, b = {0}", a, b); // a = 20, b = 10
```

Örneğin:

```
int a = 10, b = 20;

System.Console.WriteLine("{1}{0}", a, b);           // 2010
```

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            double x, y, r;

            System.Console.Write("Lütfen derece cinsinden bir açı giriniz:");
            x = double.Parse(System.Console.ReadLine());

            r = x * System.Math.PI / 180;
            y = System.Math.Sin(r);

            System.Console.WriteLine("Sin({0})={1}", x, y);
        }
    }
}
```

Eğer yer tutucu ile belirtilen indekse bir argüman karşılık gelmiyorsa exception oluşur. Exception çalışan programda bir sorun çıkması durumu için kullanılan bir terimdir. Exception programın çalışma zamanına ilişkindir. Yani örneğin biz küme parantezlerinin içerisine büyük bir değer yazarsak program derlenir fakat çalışma sırasında çöker.

Write ve WriteLine metotlarında biz gerçekten '{' ve '}' karakterlerini yazdırmak isteyebiliriz. Bunu sağlamak için iki tane yan yana bu karakterlerden kullanılmalıaıdr. Örneğin:

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

```
            System.Console.WriteLine("{{0}}, {{1}}");
        }
    }
}
```

Parametrik yazdırmanın bazı ayrıntıları vardır. Bunlar ileride gereksinim duyuldukça ele alınacaktır.

Noktalı Virgülün İşlevi

Noktalı virgül bir ifadeyi sonlandırarak onu diğerlerinden ayırmak için kullanılmaktadır. Eğer noktalı virgül unutulursa derleyici önceki ifadeyle sonraki ifadeyi tek bir ifade sanır. Bu da error'e yol açar. Örneğin:

```
a = 10                  // dikkat, noktalı virgül unutulmuş!
b = 20;
```

derleyici sanki bunu a = 10 b = 20 gibi değerlendirecektir. Programlama dillerinde bu görevdeki atomlara sonlandırıcı (terminator) denilmektedir. Bazı dillerde sonlandırıcı olarak ENTER (line feed) karakteri kullanılmaktadır. (Örneğin BASIC'te, sembolik makina dillerinde vs.). Tabii bu tür dillerde her ifade ayrı bir satıra yazılmak zorundadır.

Etkisiz İfadeler

Program içerisinde hiçbir durum değişikliğine yol açmayan ifadelere etkisiz ifadeler denir. C#'ta etkisiz ifadeler error oluşturmaktadır. Örneğin:

```
10 + 20;          // error!  etkisiz ifade
a + b;            // error! etkisiz ifade
```

Metot çağrıları metotların içi boş olsa bile bir etki oluştururlar. C ve C++ dillerinde etkisiz ifadeler geçerli kabul edilirler. Ancak pek çok C ve C++ derleyicisi bu durumlar için uyarı mesajları vermektedir.

Exception Kavramı

Exception konusu ileride ayrıntılarıyla ele alınacaktır. Burada yalnızca kavramsal bir açıklama yapacağız. Programın çalışma zamanı sırasında (runtime), programın çökmesine yol açabilecek problemli durumlara exception denilmektedir.

Örneğin:

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

            System.Console.WriteLine("a = {1}", a);     // exception oluşur
        }
    }
}
```

Burada WriteLine metodunda {1} yer tutucusu için bir argüman belirtilmemiştir. Bu kod çalıştırıldığında exception oluşur ve program WriteLine metodunda çöker.

Exception derleme aşamasına (compile time), programın çalışma zamanına (runtime) ilişkin bir kavramdır. Exception'lar ele alınıp (handle edilip) işlenebilirler. Bu durumda program çökmez çalışma devam eder.

Deyimler (Statements)

Bir programdaki çalıştırma birimlerine deyim (statement) denir. Program deyimlerin çalışmasıyla çalışır. Deyimler 5 gruba ayrılmaktadır:

1) Basit Deyimler (Simple Statements): İfade kavramı noktalı virgülü içermez. İfadenin sonuna noktalı virgül konursa ifade deyim olur. Böyle deyimlere de basit deyim denir. Örneğin:

```
x = 10;          // basit deyim
Foo();           // basit deyim
```

Yani basit deyimler ifade; biçimindeki deyimlerdir.

2) Bileşik Deyimler (Compound Statements): Bir bloğun içerisine sıfır tane ya da daha fazla deyim yerleştirilirse tüm blok da bir deyim olur. Böyle deyimlere bileşik deyim denir. Bileşik deyimler adeta küçük kutuları içeren büyük kutular gibidir. Örneğin:

```
{
      a = 10;
      {
            b = 20;
            c = 30;
      }
}
d = 40;
```

Burada dışarıdan bakıldığında iki deyim vardır: Bir tane bileşik deyim ve bir tane basit deyim. Bileşik deyimin içerisinde de iki deyim vardır: Bir tane basit deyim ve bir tane bileşik deyim.

3) Kontrol Deyimleri (Control Statements): Program akışını kontrol etmek için kullanılan if, for, while gibi deyimlere kontrol deyimleri denir. Zaten bu bölümdeki ana konumuz kontrol deyimleridir.

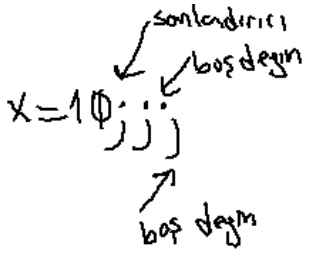
4) Bildirim Deyimleri (Declaration Statements): Bildirim işlmleri de birer deyim oluşturur. Bunlara bildirim deyimleri denilmektedir. Örneğin:

```
int a, b;        // bildirim deyimi
double x;        // bildirim deyimi
```

5) Boş Deyimler (Null statements): Solunda ifade olmadan kullanılan noktalı virgüller de bir deyim beliritir. Bunlara boş deyim denilmektedir. Boş deyimler "hiçbirşey;" biçiminde düşünülebilir. Örneğin:

```
x = 10;;;
```

Buradaki ilk noktalı virgül sonlandırıcıdır. Diğer ikisi ise boş deyimlerdir. Bu kod parçasına dışarıdan bakıldığında 3 deyim vardır.



Deyimlerin Çalışması

Bir program deyimlerin çalışmasıyla çalışmaktadır. Her deyim çalıştığında birşey olur:

1) Bir basit deyim çalıştırıldığında o deyimdeki ifade çalıştırılır.

2) Bir bileşik deyimin çalıştırıması demek onun içindeki deyimlerin sırasıyla çalıştırılması demektir. Örneğin:

```
{
      a = 10;
      {
            b = 20;
            c = 30;
      }
}
```

Burada bileşik çalıştığında önce a = 10; basit deyimi yapılacaktır. Sonra içteki bileşik deyim yapılacaktır. O da b = 20; ve c = 30; basit deyimlerinin yapılmasına yol açacaktır. Sonuç olarak bir bileşik deyim çalıştığında onun içerisindeki deyimler yukarıdan aşağıya doğru çalıştırılmış olur.

3) Kontrol deyimleri çalıştırıldığında ne olacağı zaten sonraki başlıklarda tek tek ele alınacaktır.

4) Bir bildirim deyimi çalıştırıldığında bellekte bildirilen değişkenler için yer ayrılır.

5) Bir boş deyim çalıştırıldığında hiçbir şey olamaz. Tabii "madem ki boş deyim çalıştırıldığında bir şey olmuyor, bu durumda boş deyime ne gerek var?" diye düşünebilirsiniz. Boş deyimin anlamlı ve gerekli olduğu durumlarla ileride karşılaşılacaktır.

Aslında metotlar birer bileşikdeyim belirtirler. Bir metot çağrıldığında o metodun ana bloğunun belrttiği bileşik deyimçalıştırılır. O halde bir C# programının çalıştırılması aslında Main metodunun çağrılması anlamına gelir.

Kontrol Deyimleri

Programın akışı üzerinde etkili olan deyimlere kontrol deyimleri denir. Bu bölümde if, while, for, switch ve goto deyimleri ele alınacaktır.

if Deyimi

if deyiminin genel sentaks biçimi şöyledir:

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

if anahtar sözcüğünden sonra parantezler içerisinde bool türden bir ifade bulunmak zorundadır. if deyimi iki kısma ayrılır: Doğruysa kısmı ve yanlışsa kısmı. Her iki kısımda da tek deyim bulunmak zorundadır. Eğer bu kısımlarda birden fazla deyimin bulunması isteniyorsa onun bileşik deyim biçimine (yani bloklu biçime) dönüştürülmesi gerekir.

if deyimi şöyle çalışır: Önce parantez içerisindeki ifadenin değeri hesaplanır. Bu ifade true ise (yani doğruysa) doğruysa kısmındaki deyim, false ise (yani yanlışsa) yanlışsa kısmındaki deyim çalıştırılır. Örneğin:

```
namespace CSD
{
    class App
    {
```

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

            System.Console.Write("Bir değer giriniz:");
            a = int.Parse(System.Console.ReadLine());

            if (a > 0)
                System.Console.WriteLine("Pozitif");
            else
                System.Console.WriteLine("Negatif ya da sıfır");

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

if deyiminin tamamı dışarıdan bakıldığında tek bir deyimdir.

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            double a, b, c;

            System.Console.Write("a:");
            a = double.Parse(System.Console.ReadLine());

            System.Console.Write("b:");
            b = double.Parse(System.Console.ReadLine());

            System.Console.Write("c:");
            c = double.Parse(System.Console.ReadLine());


            Root.DispRoots(a, b, c);
        }
    }

    class Root
    {
        public static void DispRoots(double a, double b, double c)
        {
            double delta;
            double x1, x2;

            delta = b * b - 4 * a * c;
            if (delta >= 0)
            {
                x1 = (-b + System.Math.Sqrt(delta)) / (2 * a);
                x2 = (-b - System.Math.Sqrt(delta)) / (2 * a);

                System.Console.WriteLine(x1);
                System.Console.WriteLine(x2);
            }
            else
                System.Console.WriteLine("Gerçek kök yok");
        }
    }
}
```

if deyiminin else kısmı olmak zorunda değildir. Eğer if deyiminin doğrusaya kısmını oluşturan deyimden sonra else anahtar sözcüğü gelmezse bu durumda "else kısmı olmayan bir if" söz konusu olur. Örneğin:

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

            System.Console.Write("Bir değer giriniz:");
            a = int.Parse(System.Console.ReadLine());

            if (a > 0)
                System.Console.WriteLine("Pozitif");
               System.Console.WriteLine("Son");
        }
    }
}
```

Tabi genel biçimden de görüldüğü gibi if deyiminin yalnızca yanlışsa kısmı olamaz. Bu durumda koşul ifadesini değiştirmek gerekir. Örneğin:

```
if (ifade1)           // error!
else
      ifade2;
```

Bunun eşdeğeri:

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

biçimindedir. Ya da aynı etki şöyle yaratılabilir:

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

if deyiminin doğruysa kısmının yanlışlıkla boş deyim ile kapatılması durumlarıyla karşılaşılmaktadır:

```
if (ifad1);
      ifade2;
```

Buradaki noktalı virgül boş deyimdir. Boş deyim için bir şey yapılmıyor olsa da boş deyim geçerli bir deyimdir. Böylece yukarıdaki örnekte if deyiminin doğruysa kısmını boş deyim oluşturmaktadır. Diğer deyim if dışında kalmıştır. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            double a, b, c;

            System.Console.Write("a:");
            a = int.Parse(System.Console.ReadLine());
            if (a > 0) ;  // dikkat!
                System.Console.WriteLine("Pozitif");
```

```
        }
    }
}
```

Bu tür durumlarda derleyici dikkat çekmek için uyarı mesajı verbilmektedir.

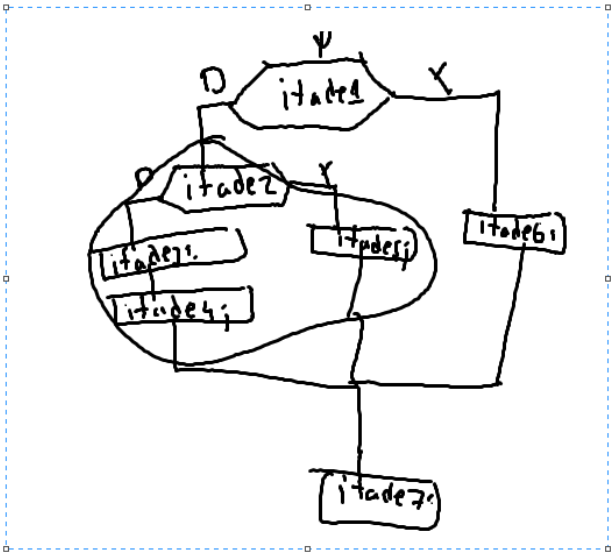
Aşağıdaki kodda nasıl bir error mesajı beklersiniz?

```
if (a > 0)
    System.Console.Write("Sayı ");
    System.Console.WriteLine("Pozitif");
else
    System.Console.Write("Sayı negatif ya da sıfır");
```

Derleyiciye göre burada bir else vardır, fakat ona karşı gelen bir if yoktur. Çünkü buradaki if deyimi "yanlışsa kısmı olmayan bir if" olarak ele alınmaktadır.

İç İçe if Deyimleri

Bir if deyiminin doğruysa ya da yanlışsa kısmında başka bir if deyimi bulunabilir. if deyiminin tamamı tek bir deyimdir.




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

Örneğin:

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

```
        {
            int a, b, c;

            System.Console.Write("a:");
            a = int.Parse(System.Console.ReadLine());

            System.Console.Write("b:");
            b = int.Parse(System.Console.ReadLine());

            System.Console.Write("c:");
            c = int.Parse(System.Console.ReadLine());

            if (a > b)
                if (a > c)
                    System.Console.WriteLine(a);
                else
                    System.Console.WriteLine(c);
            else
                if (b > c)
                    System.Console.WriteLine(b);
                else
                    System.Console.WriteLine(c);
        }
    }
}
```

Sınıf Çalışması: Klavyeden aint türden , b, c ve d değişkenleri için dört değer okuyunuz. Bunların en büyüğünü yazdırınız.

Çözüm:

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

            System.Console.Write("a:");
            a = int.Parse(System.Console.ReadLine());

            System.Console.Write("b:");
            b = int.Parse(System.Console.ReadLine());

            System.Console.Write("c:");
            c = int.Parse(System.Console.ReadLine());

            System.Console.Write("d:");
            d = int.Parse(System.Console.ReadLine());

            if (a > b)
                if (a > c)
                    if (a > d)
                        System.Console.WriteLine(a);
                    else
                        System.Console.WriteLine(d);
                else
                     if (c > d)
                         System.Console.WriteLine(c);
                     else
                          System.Console.WriteLine(d);

            else
                if (b > c)
```

```
                    if (b > d)
                        System.Console.WriteLine(b);
                    else
                        System.Console.WriteLine(d);

                else
                    if (c > d)
                        System.Console.WriteLine(c);
                    else
                        System.Console.WriteLine(d);
        }
    }
}
```

Alternatif çözüm şöyle olabilir:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int a, b, c, d;
            int max;

            System.Console.Write("a:");
            a = int.Parse(System.Console.ReadLine());

            System.Console.Write("b:");
            b = int.Parse(System.Console.ReadLine());

            System.Console.Write("c:");
            c = int.Parse(System.Console.ReadLine());

            System.Console.Write("d:");
            d = int.Parse(System.Console.ReadLine());

            max = a;
            if (a < b)
                max = b;
            if (max < c)
                max = c;
            if (max < d)
                max = d;
            System.Console.WriteLine(max);
        }
    }
}
```

Ayrık Koşullar

Bit koşul doğruyken diğerlerinin doğru olma olasılığı yoksa bu koşullara ayrık (discrete) koşullar denir. Örneğin:

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

Bu iki koşul ayrıktır. Örneğin:

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

Bu iki koşul ayrıktır. Örneğin:

```
a == 0
a == 1
a == 2
a == 4
```

Bu koşullar ayrıktır. Fakat örneğin:

```
a > 10
a > 20
```

bu iki koşul ayrk değildir.

Ayrık koşulların iki ayrık if ile ele alınması kötü bir tekniktir. Örneğin:

```
if (a > 0)
{
      //...
}

if (a < 0)
{
      //...
}
```

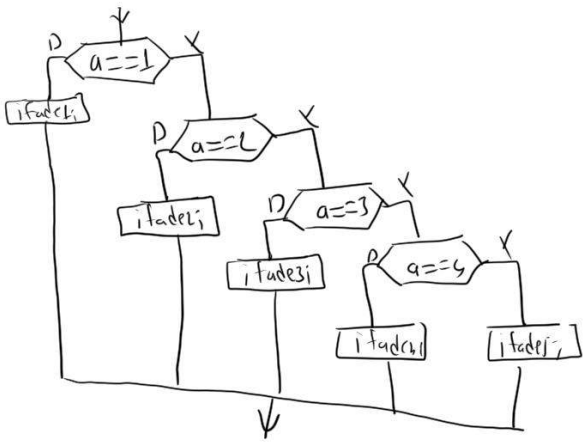
Burada a > 0 ise gereksiz bir biçimde hiç doğrulanmayacağı halde a < 0 karşılaştırması yapılmaktadır. Ayrık koşullar else-if ile ele alınmalıdır. Örneğin:

```
if (a > 0)
{
    //...
}
else
    if (a < 0)
    {
        //...
    }
```

Örneğin:

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

            System.Console.Write("Bir sayı giriniz:");
            a = int.Parse(System.Console.ReadLine());

            if (a > 0)
                System.Console.WriteLine("Pozitif");
            else
                if (a < 0)
                    System.Console.WriteLine("Negatif");
                else
                    System.Console.WriteLine("Sıfır");
        }
    }
}
```

Bazen else-if durumları bir merdiven oluşturabilir. Örneğin:




```
if (a == 0)
	ifade1;
else
	if (a == 1)
		ifade2;
	else
		if (a == 2)
			ifade3;
		else
			if (a == 3)
				ifade4;
			else
				ifade5;
```

Şüphesiz bu tür merdivenlerde olasılı yüksek koşulları daha yukarı yerleştirmek iyi bir tekniktir. Yazım bakımından sürekli kaymalar yerine aşağıdaki biçim de tercih edilmektedir:

```
if (a == 0)
	ifade1;
else if (a == 1)
	ifade2;
else if (a == 2)
	ifade3;
else if (a == 3)
	ifade4;
else
	ifade5;
```

Örneğin:

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

            System.Console.Write("Bir sayı giriniz:");
            a = int.Parse(System.Console.ReadLine());

            if (a == 1)
                System.Console.WriteLine("Bir");
            else if (a == 2)
                System.Console.WriteLine("İki");
             else if (a == 3)
                System.Console.WriteLine("Üç");
            else if (a == 4)
                System.Console.WriteLine("Dört");
            else if (a == 5)
```

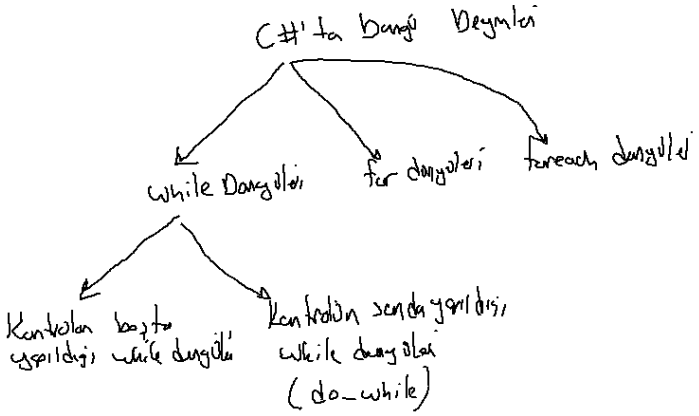
```
            System.Console.WriteLine("Beş");
        else
            System.Console.WriteLine("Diğer");

    }
  }
}
```

Döngü Deyimleri (Loop Statements)

Bir program parçasının yinelemeli olarak çalıştırılmasını sağlayan deyimlere döngü (loop) deyimleri denir. C#'ta üç döngü deyimi vardır:

1) while Döngüleri
2) for Döngüleri
3) foreach Döngüleri

while döngüleri de kendi aralarında "Kontrolün Başta Yapıldığı while Döngüleri" ve "Kontrolün Sonda Yapıldığı while Döngüleri (do-while Döngüleri)" olmak üzere ikiye ayrılmaktadır.



Kontrolün Başya Yapıldığı while Döngüleri

while döngüleri bir koşul sağlandığı sürece yinelenen döngülerdir. Kontrolün başya yapıldığı while döngülerinin genel biçimi şöyledir:

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

while döngüsü şöyle çalışır: Önce while parantezi içerisindeki ifadenin değeri hesaplanır. Bu değer true ise döngü içerisindeki deyim çalıştırılır ve başa dönülür. İfadenin değeri false ise while döngüsünün çalışması biter. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int i = 0;

            while (i < 10)
            {
                System.Console.WriteLine(i);
                ++i;
            }
            System.Console.WriteLine("Döngüden sonra i = {0}", i);
        }
    }
}
```

Örneğin:

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

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

Sınıf Çalışması: Tek bir while döngüsü kullanarak 0'dan 99'a kadar (99 dahil) sayıları her satırda 5 sayı olacak biçimde aşağıdaki gbi yazdırınız:

```
0 1 2 3 4
5 6 7 8 9
...
95 96 97 98 99
```

Çözüm:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int i = 0;

            while (i < 100)
            {
                System.Console.Write("{0} ", i);
                if (i % 5 == 4)
                    System.Console.WriteLine();
                ++i;
            }

        }
    }
}
```

Sınıf Çalışması: Önce klavyeden bir n sayısı isteyiniz. Sonra while döngüsünü kullanarak klavyeden int türden n tane sayı isteyip bunların toplamını yazdırınız.

Çözüm:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int n, i;
            int val, total;
```

```
            System.Console.Write("Bir sayı giriniz:");
            n = int.Parse(System.Console.ReadLine());

            i = 1;
            total = 0;
            while (i <= n)
            {
                System.Console.Write("{0}. Sayıyı giriniz:", i);
                val = int.Parse(System.Console.ReadLine());
                total += val;
                ++i;
            }

            System.Console.WriteLine("Girilen {0} sayının toplamı = {1}", n, total);
        }
    }
}
```

while parantezi içerisindeki ++ ve -- operatörlerine dikkat ediniz:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int i = 0;

            while (++i < 10)
                System.Console.Write("{0} ", i);
            System.Console.WriteLine();
        }
    }
}
```

Burada 1'den 9'a kadar (9 dahil) sayılar ekrana çıkar. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int i = 0;

            while (i++ < 10)
                System.Console.Write("{0} ", i);
            System.Console.WriteLine();
        }
    }
}
```

Burada 1'den 10'a kadar sayılar ekrana çıkar.

while döngüsünün yanlışlıkla boş deyim ile kapatılması durumlarıyla sık karşılaşılmaktadır. Örneğin:

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

            i = 0;
```

```
            while (i < 10) ;            // dikkat sonsuz döngü oluşur!
            {
                System.Console.WriteLine(i);
                ++i;
            }
        }
    }
}
```

Tabi bazen programcı isteyerek de (örneğin bir geciktirme oluşturmak için) döngüyü boş deyimle kapatabilir:

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

            i = 0;
            while (++i < 1000000000)
                ;
            System.Console.WriteLine("Son");
        }
    }
}
```

Bazen sonsuz döngüler oluşturmak isteyebiliriz. Bu döngülerden daha sonra göreceğimiz break deyimleriyle çıkabiliriz. Sonsuz döngü oluşturmak için while parantezinin içine true ifadesini yerleştirmek yeterlidir. Örneğin:

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

Bir while döngüsünün içerisinde başka bir while döngüsü bulunabilir. while döngülerinin de tamamı tek bir deyimdir. Örneğin:

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

            i = 0;
            while (i < 5)
            {
                k = 0;
                while (k < 5)
                {
                    System.Console.WriteLine("({0}, {1})", i, k);
                    ++k;
                }
                ++i;
            }
        }
    }
}
```

Kontrolün Sonda Yapıldığı while Döngüleri

Bu while döngülerine seyrek gereksinim duyulmaktadır. Burada kontrol noktası sondadır. Döngünün genel biçimi şöyledir:

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

Buradaki noktalı virgül boş deyim belirtmez. Sentaksın bir parçasıdır. do anahtar sözcüğü döngünün "kontrolün sonda yapıldığı while döngüsü" olduğunu belirlemek için gerekmektedir. Örneğin:

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

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

Kontrolün sonda yapılması döngü deyiminin en az bir kez çalıştırılacağı anlamına gelir. Döngünün sonda yapıldığı while döngülerine çok az gereksinim duyulmaktadır. Örneğin:

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

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

            if (ch == 'e')
                System.Console.WriteLine("Evet");
            else
                System.Console.WriteLine("Hayır");
        }
    }
}
```

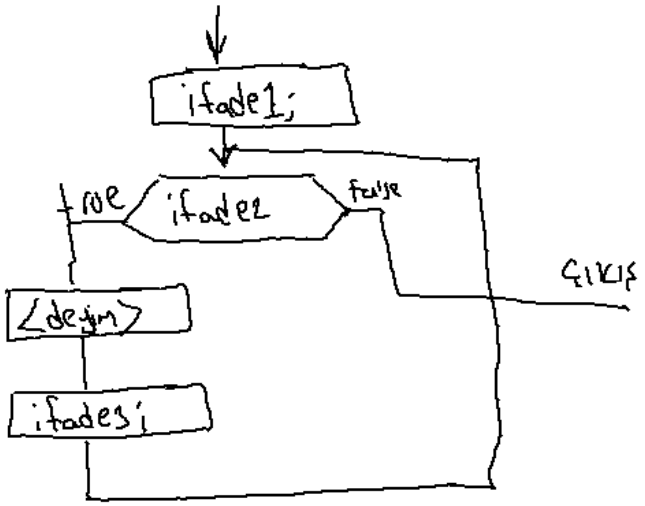
for Döngüleri

for döngüleri aslında while döngülerinin daha genel bir biçimidir. Yani while döngüsünde olan yetenek zaten for'da da vardır. Fakat for'un fazlalığı vardır. for döngülerinin genel biçimi şöyledir:

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

for anahtar sözcüğünden sonra parantezler içerisinde iki noktalı virgül bulunmak zorundadır. Bu iki noktalı

virgül döngüyü üç kısma ayırır. Döngünün ikinci kısmındaki ifade bool türden olmak zorundadır. Bunun dışında birinci ve üçüncü kısımdaki ifade herhangi bir biçimde oluşturulabilir. Döngü aşağıdaki gibi çalışmaktadır:



for döngüsünün birinci kısmındaki ifade döngüye girişte yalnızca bir kez yapılır. Bir daha da yapılmaz. Döngü ikinci kısımdaki ifade true olduğu sürece yinelenir. İkinci kısımdaki ifade her true olduğunda bir kez döngü deyimi yapılır sonra üçüncü kısımdaki ifade yapılır ve yeniden başa dönülerek ikinci kısımdaki ifade yapılır.

for döngülerinin en fazla karşılaşılan kalıbı aşağıdaki gibidir:

```
for (ilkdeğer; koşul; artırım)
      <deyim>
```

Örneğin:

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

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

Örneğin:

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

            for (i = 10; i >= 0; --i)
                System.Console.Write("{0} ", i);
            System.Console.WriteLine();
        }
```

```
    }
}
```

Örneğin:

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

            System.Console.Write("Bir sayı giriniz:");
            n = int.Parse(System.Console.ReadLine());

            total = 0;
            for (i = 1; i <= n; ++i)
                total += i;

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

Örneğin:

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

            for (x = 0; x < 2 * System.Math.PI; x += 0.1)
                System.Console.WriteLine("Sin({0:F2}) = {1:F2}", x, System.Math.Sin(x));

        }
    }
}
```

for döngüsünün birinci kısmındaki ifade hiç yazılmayabilir. Ya da aslında bu ifade oradan alınıp yukarıya konulsa da işlevsel olarak değişen bir şey olmaz. Örneğin:

```
ifade1;
for (; ifade2; ifade3)
    <deyim>
```

Örneğin:

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

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

```
    }
}
```

for döngüsünün üçüncü kısmı da yazılmayabilir. Örneğin:

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

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

Birinci ve üçüncü kısmı olmayan for döngüleri while döngüleri ile eşdeğerdir. Yani:

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

ile

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

tamamen eşdeğerdir.

while döngüsü ile for etkisi de yaratılabilir. Örneğin:

```
for (ifade1; ifade2; ifade3)
{
      //...
}
```

döngüsü eşdeğer olarak şöyle de yazılabilir:

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

for döngüsünün ikinci kısmı da bulundurulmayabilir. Eğer for döngüsünün ikinci kısmı bulundurulmamışsa koşulun sürekli sağlandığı kabul edilir. (Yani bu durum sonsuz döngü oluşturur). Daha açık bir biçimde:

```
for (ifade1; ; ifade3)
```

```
      <deyim>
```

işleminin eşdeğeri şöyle de yazılabilir:

```
for (ifade1; true; ifade3)
      <deyim>
```

Örneğin:

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

            for (i = 0; ; ++i)      // sonsuz döngü
                System.Console.WriteLine(i);
        }
    }
}
```

Nihayet for döngüsünün hiçbir kısmı olmayabilir. Tabii yine iki noktalı virgül parantezler içerisinde bulunmak zorundadır. Örneğin:

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

Aynı durum while döngüleriyle şöyle oluşturulabilir:

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

Örneğin:

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

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

for döngülerinin yanlışlıkla boş deyim kapatılması durumlarıyla karşılaşılmaktadır. Örneğin:

```
namespace CSD
```

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

            for (i = 0; i < 10; ++i) ;        // dikkat boş deyim!
                System.Console.WriteLine(i);
        }
    }
}
```

for döngüsünün birinci ve üçüncü kısmındaki ifadeler istenildiği gibi düzenlenebilir:

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

            i = 0;
            for (System.Console.WriteLine("Birinci kısım"); i < 3;
                        System.Console.WriteLine("Üçüncü kısım"))
            {
                System.Console.WriteLine("Döngü deyimi");
                ++i;
            }
        }
    }
}
```

for döngüsünün birinci ve üçüncü kısmı virgül atomu ile çoğaltılabilir. Örneğin:

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

            for (i = 0, k = 100; i + k > 50; ++i, k -= 2)
                System.Console.WriteLine("i = {0}, k = {1}", i, k);
        }
    }
}
```

for döngüsünün ikinci kısmı bu biçimde çoğaltılamaz.

for döngüsünün birinci kısmında bildirim yapılabilir. Ancak bildirilen değişkene ilkdeğer verilmesi zounludur. Örneğin:

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

```
    }
}
```

Burada aynı türden birden fazla değişkenin bildirimi de yapılabilir. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            for (int i = 0, k = 100; i + k > 50; ++i, k -= 2)
                System.Console.WriteLine("i = {0}, k = {1}", i, k);
        }
    }
}
```

for döngüsünün birinci kısmında bildirilen değişkenler yalnızca for döngüsü içerisinde kullanılabilirler. Daha açık bir biçimde ifade etmek istersek, standartlara göre:

```
for (bildirim; ifade2; ifade3)
      <deyim>
```

işleminin eşdeğeri şöyledir:

```
{
      bildirim;
      for (; ifade2; ifade3)
            <deyim>
}
```

Yani örneğin:

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

işleminin eşdeğeri şöyledir:

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

Buradan hareketle böyle bir döngüyü aşağıya kopyalarsak bir sorun oluşmaz. Örneğin:

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

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

Çünkü buradaki i'ler ayrı bloklardaymış gibi değerlendirileceklerdir.

Anahtar Notlar: Bir problemi kesin çözüme götüren adımlar topluluğuna algoritma (algorithm) denir. Bir problemi kesin çözüme değil fakat makul bir çözüme götüren adımlar topluluğuna “heuristic” denilmektedir. Algoritma hem sözel olarak hem de kod

biçiminde ifade edilebilmektedir. Genellikle her ikisi birarad kullanılarak algoritmalar açıklanır. Algoritma açıklamak için özel notasyonlar, akış diagramları vs. de kullanılabilmektedir. Ancak en yaygın kullanım sözel ve kod yazarak algoritmanın açıklanmasıdır. Algoritma "Ebû Ca'fer Muhammed bin Mûsâ el-Hârizmî" nin isminden gelmektedir.

Anahtar Notlar: Aynı işi yapan birden fazla algoritma söz konusu olduğunda bunlardan hangisinin daha iyi olduğu nasıl tespit edilir? İşte iki ölçüt çok kullanılmaktadır: Hız ve Kaynak Kullanımı. Fakat baskın ölçüt hıs'dır ve kıyaslama söz konusu olduğunda default bir biçimde hız ölçütü kastedilir. Algoritmaların hızını teknik olarak ölçmek basit bir konu değildir. Bu alanla uğraşan bölüme "Algoritma Analizi ("Analysis of Algorithms)" denilmektedir.

Anahtar Notlar: Birtakım problemleri çözmek için insanlar pek çok algoritmik yöntem düşünülmüştür. Bazen bu yöntemleri bilmek için zaman harcamak yerine doğrudan akla ilk geldiği gibi düz mantık (brute-force) çözüme de gidebiliriz.

Sınıf Çalışması: Klvayeden bir sayı isteyiniz. Onun asal çarpanlarını aşağıdaki gibi ekrana yazdırınız:

```
Lütfen bir sayı giriniz: 20
2 2 5
```

Çözüm:

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

            System.Console.Write("Lütfen bir sayı giriniz:");
            n = int.Parse(System.Console.ReadLine());

            i = 2;
            while (n != 1)
            {
                if (n % i == 0)
                {
                    System.Console.Write("{0} ", i);
                    n /= i;
                }
                else
                    ++i;
            }
            System.Console.WriteLine();
        }
    }
}
```

Asallık testi yapan basit bir metot yazabiliriz:

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

            System.Console.Write("Bir sayı giriniz:");
            n = int.Parse(System.Console.ReadLine());

            for (int i = 2; i <= n; ++i)
                if (Prime.IsPrime(i))
                    System.Console.Write("{0} ", i);
            System.Console.WriteLine();
        }
    }
```

```
    class Prime
    {
        public static bool IsPrime(int n)
        {
            for (int i = 2; i < n; ++i)
                if (n % i == 0)
                    return false;
            return true;
        }
    }
}
```

IsPrime metodunda n sayısının 2'den başlanarak kendisine kadar sayılara tam bölünüp bölünmediğine bakılmıştır. Eğer n herhangi bir sayıya tam bölünüyorsa metottan false ile, bölünmüyorsa true ile geri dönülmektedir.

Yukarıdaki algoritmayı daha iyi hale getirebiliriz: Asal olmayan her sayının karekökune kadar bir çarpanı vardır. Bu durumda bizim sayının karekökune kadar kontrol yapmamız yeterlidir. Ayrıca ikinin dışındaki çift sayıları kontrol etmeye de gerek yoktur:

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

            System.Console.Write("Bir sayı giriniz:");
            n = int.Parse(System.Console.ReadLine());

            for (int i = 2; i <= n; ++i)
                if (Prime.IsPrime(i))
                    System.Console.Write("{0} ", i);
            System.Console.WriteLine();
        }
    }

    class Prime
    {
        public static bool IsPrime(int n)
        {
            if (n % 2 == 0)
                return n == 2;

            for (int i = 3; i * i <= n; i += 2)
                if (n % i == 0)
                    return false;
            return true;
        }
    }
}
```

İç İçe Döngüler (Nested Loops)

Bir döngünün döngü deyimi başka bir döngüden oluşabilir. Örneğin:

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

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            for (int i = 0; i < 5; ++i)
                for (int k = 0; k < 5; ++k)
                    System.Console.WriteLine("({0}, {1})", i, k);
        }
    }
}
```

Sınıf Çalışması: Klavyeden bir sayı giriniz. Sıfırdan başlayarak her satırda beş tane olacak biçimde aralarına boşuk bırakarak sayıları yazdırınız. Örneğin:

Bir sayı giriniz: 22

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
20 21 22
```

Çözüm:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int n;
            System.Console.Write("bir sayı giriniz");
            n = int.Parse(System.Console.ReadLine());

            for (int i = 0; i <= n; ++i)
            {
                System.Console.Write("{0}", i);
                if (i % 5 == 4)
                    System.Console.WriteLine();
                else
                    System.Console.Write(" ");
            }
            System.Console.WriteLine();
        }
    }
}
```

Sınıf Çalışması: Klavyeden int türden bir n sayısı okuyunuz. Aşağaıdaki desenei çıkartınız:

```
*
**
***
****
*****
....
*****....**** (n tane)
```

Çözüm:

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

            System.Console.Write("Bir sayı giriniz:");
            n = int.Parse(System.Console.ReadLine());

            for (int i = 1; i <= n; ++i)
            {
                for (int k = 1; k <= i; ++k)
                    System.Console.Write("*");
                System.Console.WriteLine();
            }
        }
    }
}
```

Sınıf Çalışması: Klavyeden width ve height için iki int değer okuyunuz. Aşağıdaki kutucuğu oluşturunuz:

```
******
*    *
*    *
******
```

Burada width = 6, height = 4'tür.

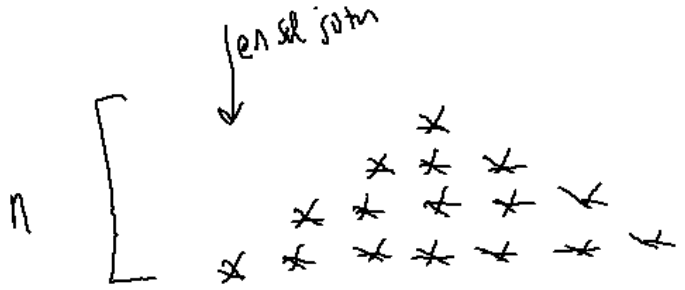
Çözüm:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int width, height;

            System.Console.Write("Genişlik giriniz: ");
            width = int.Parse(System.Console.ReadLine());

            System.Console.Write("Yükseklik giriniz: ");
            height = int.Parse(System.Console.ReadLine());

            for (int i = 1; i <= height; ++i)
            {
                for (int k = 1; k <= width; ++k)
                    if (k == 1 || k == width)
                        System.Console.Write("*");
                    else if (i == 1 || i == height)
                        System.Console.Write("*");
                    else
                        System.Console.Write(" ");
                System.Console.WriteLine();
            }
        }
    }
}
```

Aşağıdaki gibi bir çözüm de denenebilir:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int width, height;

            System.Console.Write("Genişlik giriniz: ");
            width = int.Parse(System.Console.ReadLine());

            System.Console.Write("Yükseklik giriniz: ");
            height = int.Parse(System.Console.ReadLine());

            for (int c = 1; c <= width; ++c)
                System.Console.Write("*");
            System.Console.WriteLine();

            for (int r = 2; r <= height - 1; ++r)
            {
                for (int c = 1; c <= width ; ++c)
                    if (c == 1 || c == width)
                        System.Console.Write("*");
                    else
                        System.Console.Write(" ");
                System.Console.WriteLine();
            }

            for (int c = 1; c <= width; ++c)
                System.Console.Write("*");
            System.Console.WriteLine();
        }
    }
 }
```

Sınıf Çalışması: Klavyedenn bir n sayısı isteyiniz ve aşağıdaki deseni çıkartınız:



Açıklama: Yazılımda karmaşık gibi görülen problemleri daha yalın parçalara ayırarak çözebiliriz. Yani problemi tek parça halinde çözmek yerine parçalara ayırp çözmek daha uygundur. Böylece karmaşıklık duygusu azaltılmış olur. Problemi daha yalın parçalara ayırmak o parçaları metotlar ya da sınıflar biçiminde oluşturup onları kullanarak problemi çözmek tasarımı kolaylaştırır. Bu tekniğe İngilizce "Böl ve ele geçir (divide and conquer)" denilmektedir.

Çözüm:

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

            System.Console.Write("Genişliği giriniz:");
            n = int.Parse(System.Console.ReadLine());
```

```
            Pattern.DispTriangle(n);
        }
    }

    class Pattern
    {
        public static void DispTriangle(int n)
        {
            int left, right;

            left = right = n - 1;
            for (int i = 0; i < n; ++i)
            {
                DispStar(left, right);
                --left;
                ++right;
            }
        }

        public static void DispStar(int left, int right)
        {
            for (int i = 0; i <= right; ++i)
                if (i < left)
                    System.Console.Write(" ");
                else
                    System.Console.Write("*");
            System.Console.WriteLine();
        }    }
}
```

break Deyimi

break deyimi döngüler içerisinde ya da switch deyimi içerisinde kullanılabilir. Kullanım biçimi şöyledir:

```
break;
```

Programın akışı break anahtar sözcüğünü gördüğünde döngü kırılır ve akış döngüden sonraki deyimle devam eder. Yani break döngü deyiminin kendisini sonlandırmaktadır. Örneğin:

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

            for (; ; )
            {
                System.Console.Write("Bir sayı giriniz:");
                val = int.Parse(System.Console.ReadLine());
                if (val == 0)
                    break;
                System.Console.WriteLine(val * val);
            }
        }
    }
}
```

Anahtar Notlar: Klavyeden tuşa basar basmaz (yani ENTER tuşuna gereksinim duymadan) okuma yapmak için şu kalıp kullanılır:

```
System.Console.ReadKey(true).KeyChar
```

Eğer ReadKey metoduna argüman olarak true girilirse basılan tuş ekranda gözükmez, false girilirse gözükür. Örneğin:

```
namespace CSD
```

```
{
    class App
    {
        public static void Main()
        {
            char ch;

            System.Console.WriteLine("Press any key to continue....");
            ch = System.Console.ReadKey(true).KeyChar;
            System.Console.WriteLine("Ok: {0}", ch);
        }
    }
}
```

İçi içe döngülerde break deyimi yalnızca kendi döngüsünü sonlandırır. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            for (int i = 0; i < 10; ++i)
                for (int k = 0; k < 10; ++k)
                {
                    System.Console.WriteLine("({0}, {1})", i, k);
                    if (System.Console.ReadKey(true).KeyChar == 'q')
                        break;
                }
        }
    }
}
```

Anahtar Notlar: Console sınıfının Clear metodu ekranı silmek için kullanılabilir.

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            CommandLine.Run();
        }
    }

    class CommandLine
    {
        public static void Run()
        {
            char ch;

            for (;;)
            {
                System.Console.Write("CSD>");
                ch = System.Console.ReadKey(true).KeyChar;
                System.Console.WriteLine(ch);
                if (ch == 'q')
                    break;
                if (ch == 'd')
                    System.Console.WriteLine("dir command");
                else if (ch == 'r')
                    System.Console.WriteLine("remove command");
                else if (ch == 'c')
                    System.Console.Clear();
                else
                    System.Console.WriteLine("{0}: command not found!", ch);
```

```
            }
        }
    }
}
```

İç içe döngülerde break deyimi yalnızca kendi döngüsünü kırar. Örneğin:

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

            for (int i = 0; i < 10; ++i)
                for (int k = 0; k < 10; ++k)
                {
                    System.Console.WriteLine("({0}, {1})", i, k);
                    ch = System.Console.ReadKey(true).KeyChar;
                    if (ch == 'q')
                        break;
                }
        }
    }
}
```

Yukarıdaki örnekte 'q' tuşuna basıldığında iki döngüden de çıkmak istiyorsanız iki ayrı break deyimi kullanmalısınız. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            char ch = ' ';

            for (int i = 0; i < 10; ++i)
            {
                for (int k = 0; k < 10; ++k)
                {
                    System.Console.WriteLine("({0}, {1})", i, k);
                    ch = System.Console.ReadKey(true).KeyChar;
                    if (ch == 'q')
                        break;
                }
                if (ch == 'q')
                    break;
            }
        }
    }
}
```

continue Deyimi

continue deyimi break deyimine göre seyrek kullanılmaktadır. Kullanım biçimi şöyledir:

```
continue;
```

continue deyimi yalnızca döngülerin içinde kullanılabilir. Programın akışı continue deyimini gördüğünde o anda çalıştırılmakta olan döngü deyimi (döngü deyiminin kendisi değil, onun çalıştırdığı deyim) sonlandırılır. Böylece yeni bir yinelemeye geçilmiş olur. Örneğin:

```
namespace CSD
{
```

```
    class App
    {
        public static void Main()
        {
            for (int i = 0; i < 10; ++i)
            {
                if (i % 2 == 0)
                    continue;
                System.Console.WriteLine(i);
            }
        }
    }
}
```

Programın akışı çift sayılarda continue deyhimini görür. Böylece ekrana tek sayılar çıkar. (Tabii bu örnek continue deyiminin çalışma biçimini anlatmak için verilmiştir. Yoksa tek sayıları bu biçimde yazdırmaya çalışmak iyi bir teknik değildir.)

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            CommandPrompt.Run();
        }
    }

    class CommandPrompt
    {
        public static void Run()
        {
            char ch;

            for (;;)
            {
                System.Console.Write("CSD>");
                ch = System.Console.ReadKey(true).KeyChar;
                System.Console.WriteLine(ch);

                if (ch == '\r' || ch == ' ')
                    continue;
                if (ch == 'q')
                    break;
                if (ch == 'r')
                    System.Console.WriteLine("Remove command");
                else if (ch == 'c')
                    System.Console.Clear();
                else if (ch == 'd')
                    System.Console.WriteLine("Dır command");
                else
                    System.Console.WriteLine("'{0}' is not recognized as an internal or external
command,\noperable program or batch file.", ch);
            }
        }
    }
}
```

İç içe döngülerde continue deyimi yalnızca kendi döngü deyimini (döngüde yinelenen deyimi) sonlandırır.

goto Deyimi

goto deyimi akışı belli bir noktaya koşulsuz olarak aktarmak için kullanılır. goto deyiminin genel biçimi

şöyledir:

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

Programın akışı goto deyimini gördüğünde akış etiket ile belirtilen noktaya aktarılır. goto deyiminin döngü oluşturmak gibi gerekçelerle kullanılması kötü bir tekniktir. Gereksiz goto kullanmak programın takip edilebilirliğini zorlaştırır ve kötü teknik olarak değerlendirilmektedir. Aşağıdaki örnek goto'nun kullanılması gereken yer için değil goto mekanizmasının anlaşılabilmesi için verilmiştir:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int i = 0;

        REPEAT:
            System.Console.WriteLine(i);
            ++i;
            if (i < 10)
                goto REPEAT;
        }
    }
}
```

goto deyimi ile başka bir metoda atlanılamaz. Ancak aynı metotta başka bir yere atlanılabilir. İç yerel bloktan dış yerel bloğa goto yapılabilir fakat dış yerel bloktan iç yerel bloğa goto yapılamaz.

goto etiketini bir deyim izlemek zorundadır. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            //...
            goto EXIT;
            //...

        EXIT:       // error!
        }
    }
 }
```

Bunun için boş deyim de kullanılabilir. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            //...
            goto EXIT;
            //...

        EXIT:       // geçerli
            ;
        }
```

```
    }
}
```

Farklı yerlerden aynı etikete goto yapılabilir. Bir metot içerisinde aynı isimli etiketten birden fazla bulunamaz.

goto ile atlanan yerden önce bir değişken bildirilmiş olabilir. Atlanan yerde bu değişkenin kullanılabilmesi için onun atlanılan yer itibari ile değer olmış olması gerekir. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            //...
            goto EXIT;
            //...
            int a = 10;
        EXIT:
            System.Console.WriteLine(a);        // error!
        }
    }
}
```

Fakat örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            //...
            goto EXIT;
            //...
            int a;
        EXIT:
            a = 10;
            System.Console.WriteLine(a);        // geçerli
        }
    }
}
```

goto deyiminin iyi teknik olarak kullanılabileceği üç yer vardır:

1) İç içe döngülerden ya da döngü içerisindeki switch deyiminden tek hamlede çıkmak için. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            for (int i = 0; i < 10; ++i)
            {
                for (int k = 0; i < 10; ++k)
                {
                    System.Console.WriteLine("({0}, {1})", i, k);
                    if (System.Console.ReadKey(true).KeyChar == 'q')
                        goto EXIT;
                }
            }
        EXIT:
            ;
        }
```

```
    }
}
```

2) goto deyimi ters sırada boşaltım yapmak için kullanılabilir. (Burada örneğini vermeyeceğiz.)

3) Bazı özel durumlarda goto tam tersine programın daha iyi anlaşılmasına yol açabilir. Bu tür durumlarda goto kullanabiliriz.

Yukarıdaki durumların dışında goto deyiminin kullanılması iyi teknik kabul edilmemektedir. Programın akışının başka bir yere devredilmesi kodun anlamlandırılmasını zorlaştırmaktadır.

Sabit İfadeleri (Constant Expressions)

Yalnızca sabitlerden ve operatörlerden oluşan ifadelere sabit ifadeleri denir. Örneğin:

```
1
10 * 2 - 3
5 * 4
```

birer sabit ifadesidir. Halbuki:

```
x
x + 8
x * 2
```

birer sabit ifadesi değildir. Sabit ifadelerinin net sayısal değerleri derleme aşamasında tespit edilebilir. C#'ta bazı durumlarda sabit ifadelerinin kullanılması zorunlu tutulmuştur.

switch Deyimi

switch deyimi bir ifadenin çeşitli sayısal derğerleri için çeşitli farklı işlemlerin yapılması amacıyla kullanılır. Genel biçimi şöyledir:

```
switch (<ifade>)
{
      case <s.i>:
            //...
            [break;]
      case <s.i>:
            //...
            [break;]
      case <s.i>:
            //...
            [break;]
      [
      default:
            //...
            [break;]
      ]
}
```

switch anahtar sözcüğünden sonra parantezler içerisinde bir ifade bulunmak zorundadır. switch deyimi case bölümlerinden oluşur. case anahtar sözcüğünün yanında bir sabit ifadesi bulunmak zorundadır. switch deyimi isteğe bağlı olarak default bölüm içerebilir. case bölümleri ve default bölüm genellikle (fakat mutlak değil) break deyimi ile sonlandırılır.

switch deyimi şöyle çalışır: Önce derleyici switch parantezi içerisindeki ifadenin sayısal değerini hesaplar. Sonra bu değere tam eşit olan bir case bölümü araştır. Eğer böyle bir case bölümü varsa akış o case bölüme, yoksa fakat switch deyiminin default bölümü varsa akış default bölüme yönlendirilir. break anahtar sözcüğü

switch deyimini de sonlandırmaktadır. Eğer switch parantezi içerisindeki ifadeye tam eşit olan bir case bölümü yoksa ve default bölüm de yoksa akış hemen switch deyiminde çıkar. Örneğin:

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

            System.Console.Write("Lütfen bir sayı giriniz:");
            val = int.Parse(System.Console.ReadLine());

            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 bir değer");
                    break;
            }

        }
    }
}
```

Sınıf Çalışması: Klavyeden gün, ay, yıl için değerler isteyiniz. DispDate isimli bir metot bu değerleri parametre olarak alsın ve tarihi aşağıdaki biçimde ekrana yazdırsın:

gg-ay yazısı-yyyy

Örneğin:

13-Haziran-2013

Çözüm:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int day, month, year;

            System.Console.Write("Gün:");
            day = int.Parse(System.Console.ReadLine());

            System.Console.Write("Ay:");
            month = int.Parse(System.Console.ReadLine());

            System.Console.Write("Yıl:");
            year = int.Parse(System.Console.ReadLine());

            Date.DispDate(day, month, year);
        }
    }
```

```
    class Date
    {
        public static void DispDate(int day, int month, int year)
        {
            System.Console.Write("{0}-", day);

            switch (month)
            {
                case 1:
                    System.Console.Write("Ocak");
                    break;
                case 2:
                    System.Console.Write("Şubat");
                    break;
                case 3:
                    System.Console.Write("Mart");
                    break;
                case 4:
                    System.Console.Write("Nisan");
                    break;
                case 5:
                    System.Console.Write("Mayıs");
                    break;
                case 6:
                    System.Console.Write("Haziran");
                    break;
                case 7:
                    System.Console.Write("Temmuz");
                    break;
                case 8:
                    System.Console.Write("Ağustos");
                    break;
                case 9:
                    System.Console.Write("Eylül");
                    break;
                case 10:
                    System.Console.Write("Ekim");
                    break;
                case 11:
                    System.Console.Write("Kasım");
                    break;
                case 12:
                    System.Console.Write("Aralık");
                    break;

            }
            System.Console.WriteLine("-{0}", year);
        }
    }
}
```

Anahtar Notlar: Ödev için ipucu niteliğinde örnek

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            char ch;
            int col = 20, row = 5;

            System.Console.CursorVisible = false;
            System.Console.SetCursorPosition(col, row);
            System.Console.Write("*");

            for (; ;)
            {
```

```
                ch = System.Console.ReadKey(true).KeyChar;

                if (ch == 'q')
                    break;

                System.Console.SetCursorPosition(col, row);
                System.Console.Write(" ");
                ++col;
                System.Console.SetCursorPosition(col, row);
                System.Console.Write("*");
            }
        }
    }
}
```

Sınıf Çalışması: Bir yıldızı 0, 0 noktasında gösteriniz. Sonra onu soldan sağa ve hareket ettiriniz. Bir satır bittiğinde alt satırın başından devam ediniz. * sağ alt köşeye geldiğinde program sonlansın. Her 100 milisaniyede bir yıldız konum değiştirmelidir.

Açıklama: Ekranın genişliği System:Console.Width ile yüksekliği ise System.Windows.Height ile elde edilebilir. Bekleme için System.Threading.Thread.Sleep(100) çağrısı yapılabilir.

Çözüm:
```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int col = 0, row = 0;

            System.Console.CursorVisible = false;

            System.Console.SetCursorPosition(col, row);

            for (;;)
            {
                System.Console.SetCursorPosition(col, row);
                System.Console.Write("*");
                System.Threading.Thread.Sleep(100);
                System.Console.SetCursorPosition(col, row);
                System.Console.Write(" ");
                ++col;
                if (col == System.Console.WindowWidth - 1)
                {
                    if (row == System.Console.WindowHeight - 1)
                        break;
                    col = 0;
                    ++row;
                }
            }
            System.Console.SetCursorPosition(0, 0);
            System.Console.Clear();
        }
    }
}
```

Bir switch içerisinde aynı değere sahip iki case bölümü bulunamaz. Örneğin:

```
switch (val)
{
      case 1:
            //...
            break;
      case 2:
            //...
```

```
            break;
      case 1:           // error!
            //...
            break;
}
```

switch deyiminin case bölümlerinin sıralı olması ya da default bölümün sonda olması zorunlu değildir. Örneğin:

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

            System.Console.Write("Lütfen bir sayı giriniz:");
            val = int.Parse(System.Console.ReadLine());

            switch (val)
            {
                default:
                    System.Console.WriteLine("Diğer bir değer");
                    break;
                case 3:
                    System.Console.WriteLine("Üç");
                    break;
                case 1:
                    System.Console.WriteLine("Bir");
                    break;
                case 2:
                    System.Console.WriteLine("İki");
                    break;
            }

        }
    }
}
```

switch deyiminin case bölümlerinde sabit ifadesi bulunmak zorundadır. Örneğin:

```
case 10 + 20:       // geçerli
case 10 + x:        // error
case 'a':           // geçerli
```

switch deyiminde akışın bir case bölümünden diğerine geçmesi yasaklanmıştır. Yani C#'ta aşağıya düşme (fall through) yoktur. Halbuki C, C++ ve Java'da akış case bölümünün sonunda break kullanılmamışsa aşağıya doğru düşer. İlk break görüldüğünde switch deyiminden çıkılmaktadır.

Örneğin:

```
case 1:
      ifade1;
      ifade2;           // error! akış geçiyor
case 2:
      ifade3;
      break;
```

Akışın bir case bölümünden diğerine geçmesini engellemek için C#'ta tipik olarak her case bölümünün sonuna break deyimi yerleştirilir. Ancak break yerine goto gibi, return gibi, continue gibi deyimler de akışın aşağıya geçmesini engellediği için kullanılabilir. Hatta sonsuz döngü bile akışın aşağıya geçmemesini sağlamak için için kullanılabilmektedir. Örneğin:

```
case 1:
     for (;;)         // geçerli! akış aşağıya geçmiyor!
          ;
case 2:
     ifade2;
     break;
```

Bazen birden fazla değer için aynı işlemlerin yapılması gerekebilir. Bunun tek yolu case ifadelerini peş peşe dizmektir. Örneğin:

```
case 1:
case 2:
     ifade1;
     ifade2;
     break;
```

Burada 1 ve 2 için aynı işlemler yapılmaktadır. Görüldüğü gibi istisna olarak bu durum fall through ihlali sayılmaz. Eğer bir case bölümüne hiçbir deyim yerleştirilmemişse akış aşağıya doğru düşebilir.

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            CommandPrompt.Run();
        }
    }

    class CommandPrompt
    {
        public static void Run()
        {
            char ch;

            for (; ; )
            {
                System.Console.Write("CSD>");
                ch = System.Console.ReadKey(true).KeyChar;
                System.Console.WriteLine(ch);

                if (ch == 'q')
                    break;

                switch(ch)
                {
                    case '\r':
                    case ' ':
                        continue;
                    case 'r':
                        System.Console.WriteLine("Remove command");
                        break;
                    case 'c':
                        System.Console.Clear();
                        break;
                    case 'd':
                        System.Console.WriteLine("Dir command");
                        break;
                    default:
```

```
                    System.Console.WriteLine("'{0}' is not recognized as an internal or
external command,\noperable program or batch file.", ch);
                    break;
                }
            }
        }
    }
}
```

C#'ta aşağıya doğru düşme işlemini yapay olarak gerçekleştirmek için goto case ve goto default deyimleri bulunmaktadır. Örneğin:

```
switch (a)
{
      case 1:
            ifade1;
            goto case 2;
      case 2:
            ifade2;
            goto default;
      default:
            ifade3;
            break;
}
```

switch deyiminde case ifadeleri gerçek sayı türlerine ilişkin olamaz. Tamsayı türlerine ilişkin olmak zorundadır. Örneğin:

```
case 1:               // geçerli
      //...
      break;
case 1.2:             // error!
      //...
      break;
```

Benzer biçimde switch parantezi içerisindeki ifade de gerçek sayı türlerine ilişkin olamaz. Tamsayı türlerine ilişikin olmak zorundadır. Örneğin:

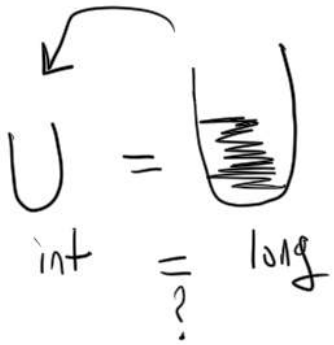
```
double d;
//...
switch (d)       // error!
{
      //...
}
```

Farklı Türlerin Birbirlerine Atanması ve Otomatik Tür Dönüştürmeleri

C#'ta her tür her türe atanamaz. Farklı türlerin birbirlerine atanmasının bazı kuralları vardır. Bir atama işleminde atanacak değerin türüne kaynak tür, atınalacak değişkenin türüne hedef tür denir. T1 türünün T2 türüne atanması sırasında T1 türü otomatik olarak (implicitly) T2 türüne dönüştürülür ondan sonra atama yapılır. Bir türün bir türe atanması için o türden o türe otomatik tür dönüştürmesinin (implicit type conversion) var olması gerekir.

C#'ta özet olarak büyük türden küçük türe atamalar yasaklanmıştır, fakat küçük türden büyük türe atamalar serbest bırakılmıştır. Başka bir deyişle küçük türlerden büyük türlere otomatik (implicit) tür dönüştürmesi vardır fakat büyük türlerden küçük türlere yoktur. Örneğin int türü long türüne atanır fakat long türü int türüne atanmaz. Bu kuralın nedeni bilgi kaybının oluşmasını engellemektir. Örneğin biz kaynak türdeki değerle hedef

türdeki değerleri bardaklara benzetebiliriz. Küçük bardaki suyu büyük bardağa dökerken su kaybı söz konus olmaz. Ancak büyük bardaktaki suyu küçük bardağa dökerken su kaybetme (taşma) potansiyeli vardır.



Büyük türün içerisindeki değer küçük türün sınırları içerisinde kalıyor olsa bile atama geçersizdir. Çünkü derleyici atanılacak değere bakmamakta, yalnızca kaynak ve hedef türlere bakmaktadır. Örneğin:

```
long a = 10;
int b;

b = a;       // error!
```

Atama denildiğinde yalnızca = operatörüyle yapılan atamalar anlaşılmamalıdır. return işlemi de, metot çağırma da bir çeşit atama işlemidir.

Otomatik tür dönüştürmelerinin bazı ayrıntıları vardır:

1) Gerçeksayı türlerinden tamsayı türlerine otomatik dönüştürme yani atama yoktur, tamsayı türlerinden gerçeksayı türlerine vardır. Örneğin float türü long türüne atanamaz ancak long türü float türüne atanabilir.

2) Küçük işaretli tamsayı türünden büyük işaretsiz tamsayı türüne otomatik dönüştürme (yani atama) yoktur. Fakat küçük işaretsiz tamsayı türünden büyük işaretli tamsayı türüne vardır. Örneğin int türü ulong türüne atanamaz fakat uint türü long türüne atanabilir. Küçük işaretsiz tamsayı türünün bütün değerlerinin büyük işaretli tamsayı türü tarafından içerildiğine dikkat ediniz.

3) Gerçek sayı türlerinden decimal türüne, decimal türünden de gerçek sayı türlerine otomatik dönüştürme (yani atama) yoktur. Örneğin ne double türü decimal türüne ne de decimal türü double türüne atanabilir.

4) char türünden ushort türüne ve daha büyük türlere otomatik dönüştürme (yani atama) vardır diğer türlerden char türüne otomatik dönüştürme (yani atama) yoktur. Ancak char türüne yalnızca char türü atanabilir.

5) bool türünden hiçbir türe, hiçbir türden de bool türüne otomatik dönüştürme (yani atama) yoktur.

6) Aynı tamsayı türünün işaretli, ve işaretsiz biçimleri arasında otomatik dönüştürme (yani atama) yoktur. Örneğin ne uint türü int türüne ne de int türü uint türüne atanabilir.

7) C#'ta int'ten küçük türlere atama yapılabilmesi için iki kural daha bulunmaktadır. int türünden bir sabit ifadesi, belirttiği değer hedef türün sınırları içerisinde kalıyorsa byte, sbyte, short, ushort ve ulong türlerine otomatik dönüştürülür (yani atanır). Ayrıca long türden bir sabit ifadesi belirttiği değer hedef türün sınırları içerisinde kalıyorsa ulong türüne otomatik dönüştürülür (yani atanır).
Örneğin:

```
short a;
int b = 10;
sbyte c;
ushort d;

a = 100;     // geçerli, 100 int türden sabit ifadesidir ve short sınırları içerisindedir
```

```
a = b;       // error! atanan değer sabit ifadesi değil
c = 200;     // error! atanan değer int türden sabit ifadesi fakat hedef türün sınırları
             // içerisinde değil
d = -1;      // error! ata!nan değer int türden sabit ifadesi fakat hedef türün
             // sınırları içerisinde değil
```

8) Bunların dışında kalan durumlar için küçük türden büyük türe otomatik dönüştürme (yani atama) vardır fakat büyük türden küçük türe yoktur.

Nihayet biz hangi türden hangi türe otomatik dönüştürmenin (yani atamanın) olduğunu hangi türden hangi türe de olmadığını tek tek yazabiliriz:

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

İşlem Öncesi Otomatik Tür Dönüştürmeleri

C#'ta yalnızca değişkenlerin ve sabitlerin değil her ifadenin de bir türü vardır. Derleyici bir operatörle karşılaştığında önce operand'ların türlerine bakar. Eğer iki operand da aynı türdense işlemi yapar ve sonuç aynı türden çıkar. Eğer operand'lar farklı türlerdense önce onları aynı türe dönüştürür ve işlemi ondan sonra yapar. Elde edilen değer de bu ortak tür türünden olur. Özet kural küçük türün büyük türe dönüştürülmesi biçimindedir. Örneğin int ile long işleme sokulduğunda int türü long türüne dönüştürülür sonuç long türünden çıkar. Örneğin:

```
int a = 10;
double b = 5.6;
int c;

c = a + b;          // error! double'dan int türüne otomatik dönüştürme (yani atama) yok
```

Tür dönüştürmesi geçici değişken yoluyla yapılmaktadır. Yani derleyici önce büyük tür türünden geçici bir değişken yaratır. Küçük türdeki değeri ona atar. Onu işleme sokar ve sonra geçici değişkeni yok eder. Örneğin:

```
int a = 10;
double b = 2.3;
double c;

c = a + b;
```

işlemi aslında şöyle yapılmaktadır:

```
<double türünden temp yaratılıyor>
temp = a;
c = temp + b;
<temp yok ediliyor>
```

İşlem öncesi otomatik tür dönüştürmelerinin bazı ayrıntıları vardır.

1) Bölme işleminde her iki operand da tamsayı türlerine ilişkinse sonuç tamsayı türünden çıkar. Elde edilen bölünde sayının noktadan sonraki kısmı atılır. Örneğin:

```
namespace CSD
{
    class App
```

```
    {
        public static void Main()
        {
            double result;

            result = 10 / 4;            // atanan değerin double olmasının bir önemi yok
            System.Console.WriteLine(result);      // 2
        }
    }
}
```

Burada 10 ve 4 int türdendir. Bölme işleminin sonucu da int türden elde edilecektir. Sonucun double türüne atanmasının bir önemi yoktur. Sonucun double çıkması için operandlardan en az birinin double olması gerekir:

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

            result = 10D / 4;
            System.Console.WriteLine(result);       // 25
        }
    }
}
```

2) Gerçeksayı türleriyle tamsayı türleri işleme sokulduğunda dönüştürme her zaman gerçeksayı türüne doğru yapılır. Örneğin float ile long işleme sokulsa long float'a dönüştürülür.

3) Küçük işaretli tamsayı türü ile büyük işaretsiz tamsayı türü işleme sokulamaz. Çünkü küçük işaretli tamsayı türünden büyük işaretsiz tamsayı türüne otomatik dönüştürme yoktur. Örneğin:

```
short a = 10;
ulong b = 20;
ulong c;

c = a + b;          // error
System.Console.WriteLine(c);
```

4) float ve double türü decimal türü ile işleme sokulamaz. Çünkü float ve double türünden decimal türüne otomatik dönüştürme yoktur.

5) int türünden küçük olan türler (yani sbyte, byte, short, ushort ya da char türleri) kendi aralarında işleme sokulduğunda önce her iki operand da bağımsız olarak int türüne dönüştürülür. Sonra işlem yapılır. İşlemin sonucu int türünden olur. Buna int türüne yükseltme (integer promotion) denilmektedir. Yani özetle C'de işlemler en az int duyarlılığında yapılmaktadır. Örneğin:

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

c = a + b;          // error!
```

Benzer biçimde örneğin sbyte ile ushort işleme sokulabilir. Bu durumda önce her iki operand da int türüne dönüştürülecek sonuç int türünden çıkacaktır. Örneğin:

```
short a = 1;

++a;            // geçerli
a = a + 1;      // error!
```

6) bool türü hiçbir türle işleme sokulamaz.

Taşma (overflow/underflow) Durumları

Bir işlemin sonucu o türün sınırları dışında kalıyorsa burada taşma söz konusudur. Örneğin:

```
int a = 2000000000, b = 2000000000, c;

c = a + b;      // dikkat taşma var!
System.Console.WriteLine(c);
```

Burada a + b işleminin sonucu int türündendir fakat elde edilen sonuç int türünün sınırları içerisinde kalmamaktadır. Taşma oluştuğunda ne olacağı içinde bulunulan bağlama (context) bağlıdır. Kontrollü bağlamda (checked context) taşma oluştuğunda exception fırlatılır ve bu exception ele alınmazsa program çöker. Kontrolsüz bağlamda (unchecked context) sayının yüksek anlamlı byte değerleri atılır, düşük anlamlı byte değerleri elde edilir. Default durumun kontrollü bağlam mı yoksa kontrolsüz bağlam mı olduğu derleyiciye bağlı olarak değişebilir. Microsoft'un csc.exe derleyicisinde default durum kontrolsüz bağlamdır. Visual Studio IDE'sinde default bağlam proje seçeneklerinden Build/Advanced/Check for arithmetic overflow/underflow seçeneği ile değiştirilebilir. Fakat bağlam istenirse kod içerisinde checked ve unchecked bloklarıyla değiştirilebilir. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int a = 2000000000, b = 2000000000, c;

            checked
            {
                c = a + b;      // exception oluşacak!
            }
            System.Console.WriteLine(c);
        }
    }
}
```

Fakat örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int a = 2000000000, b = 2000000000, c;

            unchecked     // yazmasaydık da olurdu default durum
            {
                c = a + b;      // exception oluşmaz, yüksek anlamlı byte'lar atılır
            }
            System.Console.WriteLine(c);
        }
    }
}
```

Tür Dönüştürme Operatörü

Bazen bir ifadenin türünü bilinçli olarak işleme sokarken başka bir türe dönüştürmek isteyebiliriz. Örneğin:

```
int a = 10, b = 4;
double c;
```

```
c = a / b;
```

Biz burada a ve b'nin int olmasını ve sonucun da kırpılmamasını istiyor olabiliriz. Tür dönüştürme operatörünün kullanım biçimi şöyledir:

```
(<tür>) operand
```

Tür dönüştürme operatörü tek operandlı önek bir operatördür ve öncelik tablosunun ikinci düzeyinde Sağdan-Sola grupta bulunur.

( )	Soldan-Sağa
+ - ++ -- ! (tür)	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
> < >= <=	Soldan-Sağa
== !=	Soldan-Sağa
&&	Soldan-Sağa
||	Soldan-Sağa
=, +=, -=, /=, *=, %=, ...	Sağdan-Sola

Örneğin:

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

i1: (double) a
İ2: İ1 / b
İ3: c = İ2
```

Tür dönüştürmesi bir işlemlik etki göstermektedir. Derleyici tür dönüştürmesi için önce dönüştürülecek tür türünden geçici bir değişken yaratır. Sonra dönüştürülecek ifadeyi ona atar işleme de o değişkeni sokar. Yani asıl değişkenin türü değişmemektedir. Örneğin:

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

işlemi aslında şöyle yapılmaktadır:

```
<double türünden temp yaratılır>
temp = a;
c = temp / b;
<temp yok edilir>
```

Aşağıdaki ifadede a * b işleminin sonucunun türü dönüştürülmektedir:

```
c = (double)(a * b);
```

Tür dönüştürme operatörü Sağdan-Sola önceliklidir. Örneğin:

```
b = (double)(long)a;

İ1: (long)a
İ2: (double)İ1
İ3: b = İ2
```

Tür dönüştürme operatörü ile yapılan dönüştürmelere “explicit conversion” denilmektedir. bool türü dışındaki

tüm temel türler tür dönüştürme operatörüyle birbirlerine dönüştürülebilir. Örneğin:

```
long a = 10;
int b;

b = a;          // error! long'tan int'e otomatik dönüştürme yok
b = (int)a;     // geçerli long'tan int'e operatörle dönüştürme yapılabilir
System.Console.WriteLine(b);
```

Pekiyi temel türler arasında tür dönüştürme operatörü ile dönüştürme yapıldığında ne olur? Kurallar şöyledir (else-if biçiminde):

1) Eğer dönüştürülecek kaynak türdeki değer dönüştürülecek hedef türün sınırları içerisinde kalıyorsa bilgi kaybı söz konusu olmaz. Örneğin:

```
long a = 10;
int b;

b = (int)a;
```

2) Büyük tamsayı türünden küçük tamsayı türüne dönüştürme yapıldığında sayının yüksek anlamlı byte değerleri kaybedilir. Düşük anlamlı byte değerleri atanır. Böylece sayı ilkiyle ilgisiz bir hale gelebilir. Örneğin:

```
int a = 70000;
short b;

b = (short)a;
System.Console.WriteLine(b);           // 4464
```

3) Aynı tamsayı türünün işaretli ve işaretsiz biçimleri arasında dönüştürme yapılırsa sayının bit kalıbı değişmez. Yalnızca işaret bitinin anlamı değişir. Örneğin:

```
int a = -1;
uint b;

b = (uint)a;
System.Console.WriteLine(b);    // 4294967295
```

4) Küçük işaretli tamsayı türünden büyük işaretsiz tamsayı türüne dönüştürme iki aşamada yapılır. Önce değer büyük türün işaretli biçimine dönüştürülür sonra büyük türün işaretli biçiminden büyük türün işaretsiz biçimine dönüştürme yapılır. Örneğin:

```
sbyte a = -1;
uint b;

b = (uint)a;
System.Console.WriteLine(b);    // 4294967295
```

5) Gerçek sayı türlerinden tamsayı türlerine dönüştürme yapıldığında sayının noktalı kısmı tamamen atılır, tam kısmı elde edilir. Örneğin:

```
double a = -3.99;
int b;

b = (int)a;
System.Console.WriteLine(b);    // -3
```

Eğer sayının noktalı kısmı atıldığı halde tam kısmı hala hedef türün sınırları içerisinde kalmıyorsa bu durumda kontrolsüz bağlamda herhangi bir değer elde edilir, kontrollü bağlamda exception oluşur.

6) double türü float türüne dönüştürüldüğünde eğer basamaksal bir kayıp varsa en büyük ya da en küçük float

değer elde edilir. Fakat basamaksal kayıp yoksa mantissel bir kayıp varsa float türüyle ifade edilebilen en yakın sayı elde edilir.

7) float ya da double türü decimal türe dönüştürülmek istendiğinde eğer basamaksal bir kayıp varsa exception oluşur. Eğer basamaksal bir kayıp yoksa decimal türüyle ifade edilebilen en yakın sayı elde edilir.

8) bool türü tür dönüştürme operatörüyle bile başka bir türe dönüştürülemez.

Koşul Operatörü (Conditional Operator)

Koşul operatörü ? : ile temsil edilir ve C#'ın üç operandlı tek operatörüdür. Kullanımı şöyledir:

```
ifade1 ? ifade2 : ifade3
```

Soru işaretinin solundaki ifade bool türden olmak zorundadır. Koşul operatörü şöyle çalışır: Önce soru işaretinin solundaki ifade yapılır. Bu ifade true ise yalnızca soru işareti ile iki nokta üst üste arasındaki ifade yapılır ve koşul operatörü bu değeri üretir. Eğer soru işaretinin solundaki ifade false ise bu durumda da yalnızca iki nokat üst üstenin sağındaki ifade yapılır ve koşul operatöründen bu ifadenin değeri elde edilir. Örneğin:

```
result = a > 0 ? Foo() : Bar();
```

Burada a > 0 ise Foo çağrılır ve onun geri dönüş değeri result'a atanır. a > 0 değilse bu kez Bar çağrılır ve onun geri dönüş değeri result'a atanaır. Şüphesiz koşul operatörüyle yapılabilen her şey if deyimiyle de yapılabilir. Örneğin yukarıdaki ifadenin if eşdeğeri şöyledir:

```
if (a > 0)
      result = Foo();
else
      result = Bar();
```

Örneğin:

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

            System.Console.Write("Lütfen bir sayı giriniz:");
            a = int.Parse(System.Console.ReadLine());

            result = a > 0 ? 100 : 200;

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

Koşul operatörü üç durumda güzel bir biçimde kullanılabilmektedir:

1) Bir karşılaştırmanın sonucunda elde edilen değerin bir değişkene atandığı durum. Örneğin:

```
result = a > 0 ? 100 : 200;
```

Böyle bir durumda if yerine koşul operatörü kullanmak ifadeyi daha kompakt göstermektedir. Örneğin aynı işlemi if deyimi kullanarak şöyle yapabilirdik:

```
if (a > 0)
```

```
      result = 100;
else
      result = 200;
```

2) Metot çağrılırken argüman ifadesinde. Örneğin:

```
Foo(a > 0 ? 100 : 200);
```

Eşdeğe rif karşılığı:

```
if (a > 0)
      Foo(100);
else
      Foo(200);
```

Örneğin:

```
System.Console.WriteLine(a % 2 == 0 ? "Çift" : "Tek");
```

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            for (int i = 0; i < 100; ++i)
                System.Console.Write("{0}{1}", i, i % 5 == 4 ? "\n" : " ");
        }
    }
}
```

3) Koşul operatörü return ifadelerinde de kullanılabilmektedir. Örneğin:

```
return a > 0 ? 100 : 200;
```

Eşdeğer if karşılığı şöyle yazılabilir:

```
if (a > 0)
      return 100;
else
      return 200;
```

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int x, y, result;

            System.Console.Write("Bir sayı giriniz:");
            x = int.Parse(System.Console.ReadLine());

            System.Console.Write("Bir sayı daha giriniz:");
            y = int.Parse(System.Console.ReadLine());

            result = Max(x, y);
            System.Console.WriteLine(result);
        }
```

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

Koşul operatörü öncelik tablosunda atama operatörünün hemen yukarısında Sağdan-Sola grupta bulunur.

( )	Soldan-Sağa
+ - ++ -- ! (tür)	Sağdan-Sola
* / %	Soldan-Sağa
+ -	Soldan-Sağa
> < >= <=	Soldan-Sağa
== !=	Soldan-Sağa
&&	Soldan-Sağa
||	Soldan-Sağa
? :	Sağdan-Sola
=, +=, -=, /=, *=, %=, ...	Sağdan-Sola

Koşul operatörü parantezlere alınırsa diğer operatörlerden ayrıştırılabilir. Örneğin:

```
result = (a > 0 ? 100 + 200 : 300 ) + 400;
```

Burada artık + operatörü koşul operatörünün operandı olmaktan çıkmıştır. Parantezler olmasaydı + 400 koşul operatörünün yanlışsa kısmındaki ifadeye dahil olacaktı.

Koşul operatörü iç içe kullanılabilir. Aslında böyle bir kullanımda hiç parantez gerekmez:

```
max = a > b ? a > c ? a : c : b > c ? b : c;
```

Ancak parantezler okunabilirliğe katkıda bulunabilir:

```
max = a > b ? (a > c ? a : c) : (b > c ? b : c);
```

Aynı Sınıfta Aynı İsimli Birden Fazla Metodun Bulunması Durumu (Method Overloading)

Farklı sınıflarda aynı isimli metotlar her koşul altında bulunabilir. Bunlar zaten farklı sınıflarda olduğu için bir soruna yol açmaz. Örneğin:

```
class A
{
    public static void Foo(int a)
    {
        //...
    }
    //...
}


class B
{
    public static void Foo(int a)
    {
        //...
    }
```

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

Fakat aynı sınıf içerisinde de aynı isimli birden fazla metot bulunabilmektedir. Bu duruma İngilizce "method overloading" denilmektedir. Ancak aynı sınıf içerisinde aynı isimli metotların bulunabilmesi için bunların parametrik yapılarının birbirinden farklı olması gerekir. Aynı parametrik yapıya sahip aynı isimli birden fazla metot aynı sınıf içerisinde bulunamaz. Parametrik yapıların farklı olması demek parametre değişkenlerinin sayıca ya da türce farklı olması demektir. Parametre değişkenlerinin isimlerinin farklı olmasının bu anlamda bir etkisi yoktur. Önemli olan onların türleridir. Örneğin:

```
class Sample
{
    public static void Foo(int a)
    {
        //...
    }

    public static void Foo(long a)
    {
        //...
    }

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

Buradaki Foo metotlarının parametrik yapıları farklıdır. Fakat aşağıdaki örnekte error oluşur:

```
class Sample
{
    public static void Foo(int a)
    {
        //...
    }

    public static void Foo(long a)
    {
        //...
    }

    public static void Foo(int c)        // Dikkat parametrik yapı aynı!
    {
        //...
    }
    //...
}
```

Aynı isimli metotların geri dönüş değerleri bu anlamda bir farklılık oluşturmamaktadır. Yani örneğin parametrik yapısı aynı geri dönüş değerlerinin türleri farklı aynı isimli iki metot aynı sınıfta bulunamaz:

```
class Sample
{
    public static void Foo(int a)
    {
        //...
    }

    public static int Foo(int a)        // error!
    {
        //...
        return 0;
    }
    //...
```

```
}
```

Benzer biçimde erişim belirleyicilerinin farklı olması metotların static olup olmaması da bu anlamda metotları ayrıştırmamaktadır. Örneğin:

```
class Sample
{
    public static void Foo(int a)
    {
        //...
    }

    private int Foo(int a)        // error!
    {
        //...
        return 0;
    }
    //...
}
```

Metotların ya parametre sayıları farklı olmalıdır ya da eğer parametre sayıları aynıysa onların türleri farklı olmalıdır.

Bir metodun ismi ve sırasıyla parametre türlerinin oluşturduğu dizilime metodun imzası (signature) denilmektedir. Örneğin:

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

Bu metodunun imzası [Foo, int, long, double] biçimindedir. Örneğin:

```
private int Foo(int a, long b)
{
      //...
}
```

metodunun imzası [Foo, int, long] biçimindedir.

Görüldüğü gibi metodun erişim belirleyicisinin, geri dönüş değerinin türünün, parametre değişkenlerinin isimlerinin imza üzerinde bir etkisi yoktur. Pekiyi aşağıdaki iki metodun imzası aynı mıdır?

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

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

Yanıt: Hayır. Çünkü imzada dizilim önemlidir. Birinci metodun imzası [Foo, int, long] ikinci metodun imzası [Foo, long, int] biçimindedir. Görüldüğü gibi bu imzalar farklıdır.

Aynı sınıf içerisinde aynı isimli metotların bulunma kuralı için daha kısa biçimde şunlar söylenebilir: “Aynı sınıfta aynı imzaya sahip birden fazla metot bulunamaz”.

Aynı isimli bir metot çağrıldığında acaba overload edilmiş olan bu metotlardan hangisi çağrılacaktır? İşte bunun derleyici tarafından tespit edilmesi sürecine “overload resolution” denilmektedir. “Overload resolution” sürecinin özet kuralı şöyledir: Derleyici çağrılma ifadesindeki argümanların türlerine bakar, o türlerle aynı

parametre türlerine sahip aynı isimli bir metot varsa onun çağrılmış olduğunu kabul eder. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            int a = 100;
            double b = 12.3;

            Sample.Foo(a);      // int parametreli Foo çağrılacak
            Sample.Foo(b);      // double parametreli Foo çağrılacak
        }
    }

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

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

        public static void Foo(double a)
        {
            System.Console.WriteLine("Foo, double");
        }

        public static void Foo(float a)
        {
            System.Console.WriteLine("Foo, float");
        }

        public static void Foo(char a)
        {
            System.Console.WriteLine("Foo, char");
        }
        //...
    }
}
```

.NET'in sınıf kütüphanesindeki sınıflarda da aynı isimli pek çok metot vardır. (Örneğin Console sınıfında aslında tek bir Write ya da WriteLine metodu yok. 19 ayrı Write ve WriteLine metotları var.) Biz bu metodu çağırdığımızda derleyici argüman türlerine bakarak uygun olanı çağırmaktadır.

Pekiyi ya argüman yapısıyla tamamen aynı parametrik yapıya sahip bir metot yoksa ne olacaktır? İşte "overload resolution" sürecinin bazı ayrıntıları vardır.

Overload resolution işlemi üç aşamada yürütülmektedir. Birinci aşamada aday metotlar (candidate methods) belirlenir. İkinci aşamada aday metotlar arasında uygun olanlar (applicable methods) alınır, diğerleri atılır. Üçüncü aşamada ise uygun olanlar yarışa sokularak en uygun metot (the most applicable method) seçilmektedir. En uygun metot bulunamazsa ya da birden fazla olarak bulunursa error oluşur. Bu süreci iş başvuru sürecine benzetebiliriz. Bir iş ilanına tüm CV gönderenler "aday" kişilerdir. Bunların arasından seçme yapılarak "uygun" olanlar belirlenir ve mülakata çağrılır. Mülakat sonucunda "en uygun" olan aday seçilir.

Sınıfın çağrılma ifadesindeki isimle aynı isimli olan tüm metotları aday metotlardır. Örneğin:

```
class Sample
{
    public static void Foo(int a, int b)        // #1
    {
```

```
        System.Console.WriteLine("Foo, int, int");
    }

    public static void Foo(int a, long b)       // #2
    {
        System.Console.WriteLine("Foo, int, long");
    }

    public static void Foo(int a, double b)     // #3
    {
        System.Console.WriteLine("Foo, int, double");
    }

    public static void Foo(long a, int b)       // #4
    {
        System.Console.WriteLine("Foo, long, int");
    }

    public static void Foo(double a, double b)  // #5
    {
        System.Console.WriteLine("Foo, double, double");
    }

    public static void Foo(int a)               // #6
    {
        System.Console.WriteLine("Foo, int");
    }

    public static void Bar(int a)               // #7
    {
        System.Console.WriteLine("Bar, int");
    }
    //...
}
```

Biz şöyle bir çağırma yapmışolalım:

```
Sample.Foo(123, 10.2);
```

Burada 1, 2, 3, 4, 5 ve 6 numaralı metotlar aday metotlardır.

Uygun metotlar (applicable methods) çağırma ifadesindeki argüman sayısıyla aynı parametre sayısına sahip olan ve her argümanla parametre arasında otomatik (implicit) dönüştürmenin mümkün olduğu metotlardır. Örneğin şöyle bir çağırma yapmış olalım:

```
short a = 10;
float b = 20;

Sample.Foo(a, b);
```

Burada 1, 2, 3, 4, 5 ve 6 numaralı metotlar aday metotlardır. Ancak yalnızca 3 ve 5 numaralı metotlar uygun metotlardır.

En uygun metodun uygun metotlar arasından seçilmesi argüman parametre dönüştürmelerinin kalitesine bağlıdır. En uygun metot öyle bir metottur ki "her argüman parametre dönüştürmesi diğer uygun metotlarla yarışa sokulduğunda ya onlardan daha iyi (daha kaliteli) dönüştürme sunar ya da daha kötü olmayan dönüştürme sunar".

Otomatik dönüştürmeler arasında kalite şöyle tespit edilmektedir (else if biçiminde düşünülmelidir):

1) T1 -> T2 ve T1 -> T3 otomatik dönüştürmelerinde T2 ile T3'ün hangisi T1 ile aynıysa o dönüştürme daha kalitelidir. Yani özdeş durum her zaman daha kalitelidir. Örneğin:

int -> int
int -> long

Burada int -> int dönüştürmesi daha iyidir.

2) T1->T2 ve T1->T3 otomatik dönüştürmelerinde T2'den T3'e otomatik dönüştürme var, T3'ten T2'ye yoksa T1->T2 dönüştürmesi daha kalitelidir. Örneğin:

int -> double
int -> long

Burada int -> long dönüştürmesi daha iyidir. Örneğin:

```
short -> int
short -> long
```

dönüştürmelerinde short->int dönüştürmesi daha iyidir.

3) T1->T2 ve T1->T3 otomatik dönüştürmelerinde ne T2'den T3'e ne de T3'den T2'ye dönüştürme yoksa (doğrusu varsa diye söylenir) bu durumda işaretli türe yapılan dönüştürme daha kalitelidir. Örneğin:

uint -> long
uint -> ulong

Burada uint -> long daha kalitelidir. Örneğin:

ushort -> uint
ushort -> int

Burada ushort -> int dönüştürmesi gaha kalitelidir.

En uygun metot tüm argüman parametre dönüştürmeleri yarışa sokulduğunda diğerlerine göre her argüman parametre dönüştürmesi ya daha kaliteli ya da daha kalitesiz olmayan dönüştürme sunun metottur. Örneğin yukarıdaki Sample sınıfı için şu çağrısı yapılmış olsun:

```
short a = 10;
short b = 20;

Sample.Foo(a, b);       // 1 numaralı metot seçilir
```

Burada birinci metodun tüm argüman parametre dönüştürmesi diğerlerinden ya daha kötü değil ya da daha iyidir. Örneğin çağrı yapılmış olsun:

```
int a = 10;
float b = 20;

Sample.Foo(a, b);
```

Burada yalızca 3 ve 5 numaralı metotlar uygundur. 3 numaralı metot en uygun metot olarak seçilir. Çünkü birinci argüman parametre dönüştürmesi 5'e göre daha kötü değildir. Ancak ikinci argüman parameytre dönüştürmesi 5'e göre daha iyidir. Örneğin:

```
class Sample
{
    public static void Foo(int a, long b)        // #1
    {
        System.Console.WriteLine("Foo, int, long");
    }
```

```
    public static void Foo(long a, int b)       // #2
    {
        System.Console.WriteLine("Foo, long, int");
    }
    //...
}
```

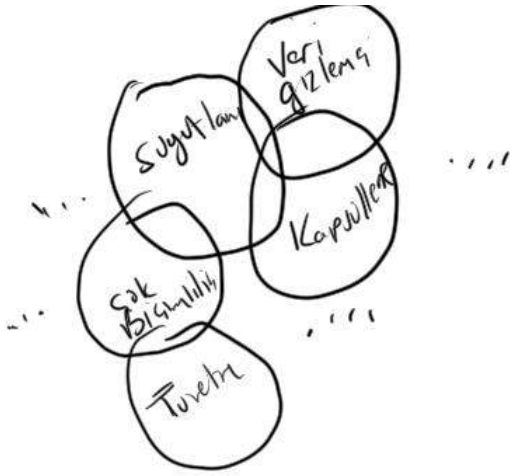
Burada şöyle bir çağrı yapılmış olsun:

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

Burada 1 ve 2 numaralı metotlar aday ve uygun metotlardır. Birinci metodun birinci argüman parametre dönüştürmesi, ikinci metodun da ikinci argüman parametre dönüştürmesi daha kalitelidir. Yani burada "tüm argüman parametre dönüştürmeleri diğer uygun metotlarla yarışa sokulduğunda onlardan adaha iyi olan ya da daha kötü olmayan bir metot yoktur".

Nesne Yönelimli Programlama Tekniği Nedir?

NYPT'nin tek bir cümleyle tatminkar bir tanımını yapmak mümkün değildir. Fakat yine de "sınıflar kullanılarak program yazmak tekniğidir" denebilir. Aslında NYPT birtakım anahtar kavramların birleşiminden oluşmaktadır. Bu anahtar kavramlar bütünüyle birbirinden ayrık değildir, iç içe geçmiştir. Bu anahtar kavramları birbirleriyle kesişen daireler biçiminde düşünebiliriz. Tüm bu anahtar kavramların temelinde kodun daha iyi algılanması ve yönetilmesi vardır.



Bilgisayarın Kısa Tarihi

Elektronik düzeyde bugün kullandığımız bilgisayarlara benzer ilk aygıtlar 1940'lı yıllarda geliştirilmeye başlanmıştır. Ondan önce hesaplama işlemlerini yapmak için pek çok mekanik aygıt üzerinde çalışılmıştır. Bunların bazıları kısmen başarılı olmuştur ve belli bir süre kullanılmıştır. Mekanik bilgisayarlarlardaki en önemli girişim Charles Babbage tarafından yapılan "Analytical Engine" ve "Diffrenece Engine" aygıtlarıdır. Analitical Engine tam olarak bitirilememiştir. Fakat bunlar pek çok çalışmaya ilham kaynağı olmuştur. Hatta bir dönem Babbage'in asistanlığını yapan Ada Lovelace bu Analytical Engine üzerindeki çalışmalarından dolayı dünyanın ilk programcısı kabul edilmektedir. Şöyle ki: Rivayete göre Babbage Ada'dan Bernolli sayılarının bulunmasını sağlayan bir yönerge yazmasını istemiştir. Ada'nın yazdığı bu yönergeler dünyanın ilk programı kabul edilmektedir. (Gerçi bu yönergelerin bizzat Babbage'in kendisi tarafından yazılmış olduğu neredeyse ispatlanımış olsa bile böyle atıf vardır.) Daha sonra 1800'lü yılların son çeyreğindne itibaren elektronikte hızlı bir ilerleme yaşanmıştır. Bool cebri ortaya atılmış, çeşitli devre elemanları kullanılmaya başlanmış ve mantık devreleri üzerinde çalışmalar başlatılmıştır. 1900'lü yılların başlarında artık yavaş yavaş elktomekanik bilgisayar fikri belirmeye başlamıştır. 1930'lu yıllarda Alan Turing konuya matematiksel açıdan yaklaşmış ve böyle bir bilgisayarın hangi matematik problemleri çözebileceği üzerine kafa yormuştur. Turing bir şerit üzerinde ilerleyen bir kafadan oluşan ve ismine "Turing Makinası" denilen soyut makina tanımlamıştır ve bu makinanın neler yapabileceği üzerinde kafa yormuştur. ACM Turing'in anısına bilgisayarın Nobel ödülü gibi kabul edilen Turing ödelleri vermektedir.

Dünyanın ilk elektronik bilgisayarının ne olduğu konusunda bir fikir birliği yktur. Bazıları Konrad Zuse'nin 1941'de yaptığı Z3 bilgisayarını ilk bilgisayar olarak kabul ederken bazıları Harward Mark 1, bazıları da ENIAC'ı kabul etmektedir.

İlk bilgisayarlarda transistör yerine vakum tüpler kullanılıyordu. (Vakum tüpler transistör görevi yapan büyük, ısınma problemi olan lambaye benzer devre elemanlarıdır). Modern bilgisayar tarihi 3 döneme ayrılarak incelenebilir:

1) Transistör öncesi dönem (1940-1950'lerin ortalarına kadar)
2) Transistör dönemi (1950'lerin ortalarından 1970'lerin ortalarına kadar)
3) Entegre devre dönemi (1970'lerin ortalarından günmüze kadarki dönem)

Transistör icad edilince bilgisayarlar transistörlerle yapılmaya başlandı ve önemli aşamalar bu sayede kaydedildi. Bilgisayar devreleri küçüldü ve kuvvetlendi. O zamanların en önemli firmaları IBM, Honeywell, DEC gibi firmalardı.

Transistörü bulan ekipten Shockley bir şirket kurarak yanına genç mühendisler aldı. Bu ekipteki Noyce ve arkadaşları ilk entegre devreleri yaptılar ve Intel firmasını kurdular. Böylece Entegre devre devrine geçilmil oldu.

Dünyanın enregre olarak üretilen ilk mikroişlemcisi Intel'in 8080'i kabul edilmektedir. Intel daha önce 4004, 8008 gibi entegreller yaptıysa da bunlar tam bir mikroişlemci olarak kabul edilmemektedir. O yıllara kadar dünyadaki bilgisayarlar sayılabilecek kadar azdı. Bunlar yüzbinlerce dolar fiyatı olan dev makinalardı ve IBM gibi şirketler çoğu kez bunları kiraya verirdi. Kiilerin evine bilgisayar alması uçuk bir fikirdi.

Intel 8080'i yaptığında bundan bir kişisel bilgisayar yapılabileceği onların aklına gelmemiştir. Kişisel bilgisayar fikri Ed Roberts isimli bir girişimci tarafından ortaya atılmıştır. Ed Roberts 8080'i kullanarak Altair isimli ilk kişsel bilgisayarı yaptı ve "Popular Electronics" isimli dergiye kapak oldu. Altair makina dilinde kodlanıyordu. Roberts buna Basiz derleyicisi yazacak kişi aradı ve Popular Electronics dergisine ilan verdi. İlana o zaman Harward'ta öğrenci olan Bill Gates ve Paul Allen başvurdular. Böylece Altair daha sonra Basic ile piyasaya sürüldü. Gates ve Allen okuldan ayrıldılar Microsoft firmasını kurdular. (O zamanlar bu yeni kişisel bilgisayarlara mikrobilgisayarlar denilmekteydi). Amerika'da bu süreç içerisinde bilgisayar kulüpleri kuruldu ve pek çok kişi kendi kişisel bilgisayarlarını yapmaya çalıştı. Steve Jobs ve Steve Wozniak Apple'ı böyle bir süreçte kurmuştur.

IBM kişsel bilgisayar konusunu hafife aldı. Fakat yine de bir ekip kurarak bugün kullandığımız PC'lerin donanımını IBM tasarlamıştır. Ancak IBM küçük iş olduğu gerekçesiyle bunlara işletim sistemini kendisi yazmadı taşeron bir firmaya yazdırmak istedi. Microsoft IBM ile anlaşarak DOS işletim sistemini geliştirdi. İlk PC'lerin donanımı IBM tarafından, yazılımı Microsoft tarafından yapılmıştır. Microsoft IBM'le iyi bir anlaşma yaptı. IBM uzağı göremedi. Anlaşmaya göre başkalarına DOS'un satışını tamamaen Microsoft yapacaktı. IBM PC için patentleri ihmal etti. Pek çok firma IBM uyumlu daha ucuz PC yaptılar. Fakat bunların hepsi işletim sistemini Microsoft'tan satın aldı. Böylece Microsoft 80'li yıllarda çok büyüdü.

İlk devirlerde bilgisayar programları ancak birkaç sayfa uzunluğunda oluyordu. Sonra transistör devrinde onbin satırdan oluşan projeler yazılmaya başlandı. Sonra yüzbin satırlara çıkıldı. PC'lerin başlarında donanım yetersizdi. PC projeleri genellikle onbinlerle ölçülen satırlarda kalıyordu. Ancak donanımlar iyileştikçe yazılımlarda kod büyümesi yaşanmaya başladı. O zamanlar kullanılan prosedürel tekniğin artık yetersiz kaldığı görülmüştür. İşte NYPT donanımların gelişmesiyle yazılımlarda ortaya çıkan kod büyümesi ile algısal olarak mücadele etmek için geliştirilmiştir. NYPT'nde artık fonksiyonlarla değil sınıflarla konuşulur. Böylece "çok şey var" duygusundan uzaklaşılarak "az şey var" duygusuna kapılınır.

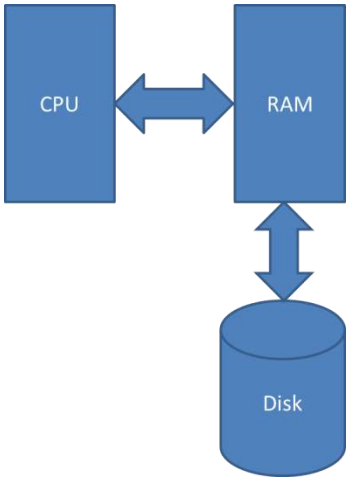
NYPT yazılım endüstrisine 90'lı yıllarda girmiştir. Fakat ilk denemeler 80'lerin başlarında yapılmıştır. Bugün yazılım endüstrisinde ağırlıklı olarak NYPT kullanılmaktadır.

Aynı İsimli Metotların Nesne Yönelimli Programlama Tekniği Bakımından Anlamı

NYPT'deki tüm anahtar kavramlar kodun daha iyi algılanmasına hizmet etmektedir. Benzer işlmleri yapan metotlara aynı ismin verilmesi de aslında algılamayı kolaylaştırır. Böylece "sanki çok şley varmış" duygusundan uzaklaşılıp "az şey var" biçiminde bir algı oluşturulur. Aslında insanın doğayı algılaması da benzer biçimdedir. Örneğin biz sandalyeler farklı olsa da konuşurken hepsine sandalye deriz. Ancak gerekirse onun başka özelliklerini söyleriz. O halde biz de benzer işlemleri yapan metotlara hep aynı isimleri vermeliyiz.

Adres Kavramı

Bir bilgisayar sistemindeki CPU (Central Processing Unit), RAM (Random Access Memory) ve Disk en önemli üç birimdir. CPU entegre devre biçiminde üretilmiş olan mikroişlemcidir. Bütün hesaplamalar ve karşılaştırmalar CPU tarafından yapılır. (Örneğin Intel Core-I5 bir CPU'dur.) CPU RAM ile elektriksel olarak bağlantılıdır. RAM'ler de bir kart üzerine monte edilmiş entegre devre modüllerinden oluşur. Disk bilgisayarı kapattığımızda bilgilerin saklandığı birimdir. Disk ile RAM arasında bir bağlantı vardır.



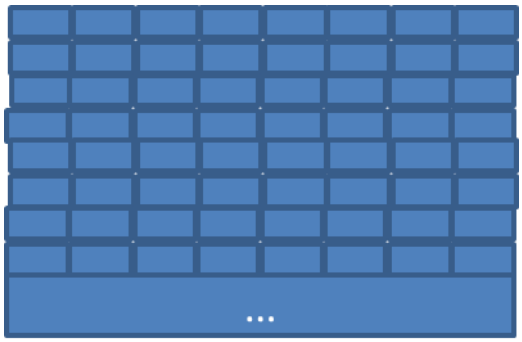
Programalama dillerindeki değişkenler RAM'de tutulurlar. İşlemler ise CPU tarafından yapımaktadır. Örneğin:

```
a = b + c;
```

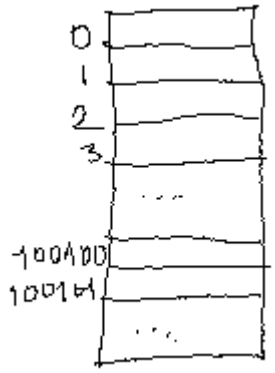
ifadesinde a, b ve c RAM'dedir. Fakat toplama ve atama işlemleri CPU tarafından yapılır. CPU RAM'e erişir oradan b'yi ve c'yi alır. Bunu elektrik devreleriyle toplar, sonucu da a'ya atar. Dosyalar ise disktedir. Bilgisayarı kapattığımızda onlar kaybolmazlar.

Örneğin bir ".exe" dosya disktedir. Biz bunu çalıştırmak istediğimizde önce işletim sistemi dosyayı diskten RAM'ye yükler. Çalışma RAM'de gerçekleşir.

Bellek (yani RAM) byte'lardan byte'lar da bitlerden oluşur. 1 byte 8 bittir. Bit (binary digit) 0 ya da 1 değerini tutan bellek hücreleridir.



Bellketeki her bir byte'a ilki 0 olmak üzere artan sırada bir sayı karşılık düşürülmüştür.

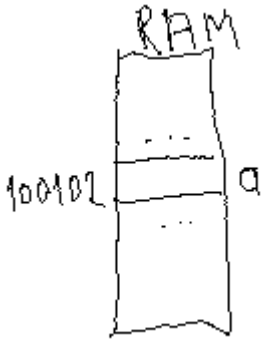


Bir byte'ın kaç numaralı byte olduğunu belirten bu sayıya o byte'ın adresi denilmektedir.

Değişkenler bellekte olduğuna göre onların da birer adresi vardır. Örneğin:

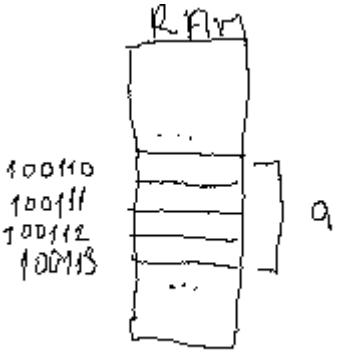
```
byte a;
```

Burada a 1 byte yer kaplayan bir değişkendir. a'nın da bir adresi vardır.



Burada a'nın adresinin 100102 olduğu görülmektedir. Bir byte'tan daha uzun olan değişkenlerin adresleri onların yalnızca en düşük adresleriyle ifade edilir. Örneğin:

```
int a;
```



Burada a 4 byte uzunluğundadır. a'nın adresi 100110'dur.

Değişkenlerin Ömürleri

Çalıştırılabilen (executable) bir dosya diskte bulunur. Çalıştırılmak istendiğinde işletim sistemi tarafından diskten alarak RAM'e yüklenir. Programın çalışması bittiğinde program RAM'den boşaltılır. Bir değişkenin bellekte yer kapladığı zaman aralığına ömür denilmektedir. Dolayısıyla bir değişkenin olabilecek maksimum ömrü çalışma zamanı (run time) kadardır.

Yerel Değişkenlerin Ömürleri

Yerel değişkenler sürekli bellekte tutulmazlar. Akış yerel değişkenin bildirildiği noktaya geldiğinde yerel değişken yaratılır, akış yerel değişkenin bildirildiği bloktan çıktığında o blokta bildirilmiş tüm yerel değişkenler yok edilir. Yani programımızda binlerce yerel değişken olsa da belli bir anda onların hepsi bellekte yer kaplıyor değildir. Bu tıpkı Sarıyer'den Beşitaş'a giden bir dolmuşta müşterilerin belli bir yerde binip belli bir yerde inmelerine benzer. Toplamda dolmuş kendi kapasitesinin ötesinde çok daha fazla kişiyi taşımış olur.

Yerel değişkenlerin yaratıldığı alana "stack" denilmektedir. Stack işletim sistemi tarafından RAM'de organize edilen bir bölümdür. Stack'in yeri ve uzunluğu sistemden sisteme değişebilmektedir. Stack'te yaratım ve yokedim tek bir makina komutuyla yani çok hızlı bir biçimde yapılmaktadır.

Yerel bir değişkene onun bildirildiği bloğun dışından erişilememesinin asıl nedeni o değişkenin blok dışında yaşamıyor olmasındandır. Değişken bildirilmeden önce de yaşamıyor durumdadır:

```
{
      //...
      a = 10;          // error! a daha yaratılmamış
      {
            int a;
            //...
      }
      a = 20;          // error! a diye bir değişken artık RAM'de yok!
      //...
}
```

Parametre Değişkenlerinin Ömürleri

Metodun parametre değişkenleri metot çağrıldığında yaratılır, metot bittiğinde yok edilir. Parametre değişkenleri de tıpkı yerel değişkenler gibi stack'te yaratılmaktadır. Parametreli bir metot çağrıldığında önce argümanların değerleri hesaplanır. Sonra parametre değişkenleri yaratılır. Argümanlardan parametre değişkenlerine bir atama yapılır. Akış metoda aktarılır. Sonra parametre değişkenleri metot bittiğinde yok edilir.

Sınıfların Veri Elemanları (Fields)

Bildirimi sınıf bildiriminin içinde yapılan değişkenlere sınıfın veri elemanları denir. Metotlar ve veri elemanları sınıfın elemanlarıdır. Sınıfın elemanları erişim belirleyicisi alabilir (yazılmazsa private anlamına gelir), static olabilir ya da olmayabilir. Örneğin:

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

    public static void Foo()
    {
        //...
    }

    public void Bar()
    {
        //...
    }
}
```

Burada a ve b sınıfın static olmayan veri elemanlarıdır (instance fields). c de sınıfın static veri elemanıdır (static/class field). Aynı biçimde Foo sınıfın static metodu, Bar da static olmayan metodudur. Fakat C#'ta yerel ve parametre değişkenleri static yapılamazlar ve erişim belirleyicisine sahip olamazlar.

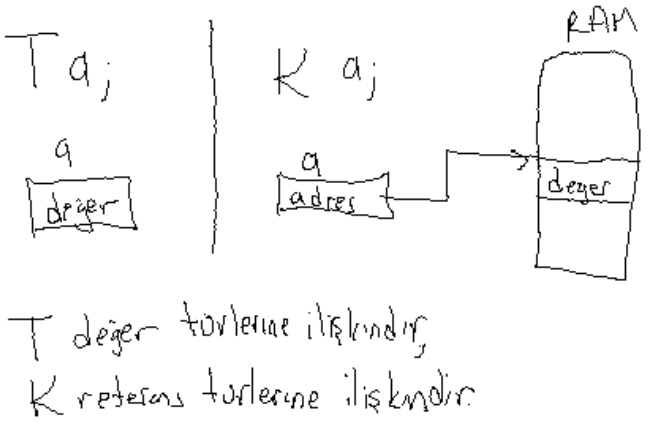
Bir sınıf bildirildiğinde aynı zamanda bir tür de oluşturulmuş olur. Sınıf türünden değişkenler de bildirilebilir. Örneğin:

```
Sample a;
```

Burada a Sample türündendir.

Değer Türleri (Value Types) ve Referans Türleri (Reference Types)

Bir tür kategori olarak ya değer türlerine ilişkindir ya da referans türlerine ilişkindir. T bir tür olmak üzere T türünden bir değişken eğer doğrudan değerin kendisini tutuyorsa T türü kategori olarak değer türlerine ilişkindir. Eğer T türünden değişken değerin kendisini değil de bir adres tutuyorsa ve asıl değer o adreste bulunuyorsa T türü kategori olarak referans türlerine ilişkindir.



Bugüne kadar gördüğümüz int, long, double gibi temel türler kategori olarak değer türlerine ilişkindir.

C#'ta bütün sınıf türleri kategori olarak referans türlerine ilişkindir. Yani bir sınıf türünden değişken bir adres tutar, değerin kendisini tutmaz.

Sınıf Türünden Nesnelerin Yaratılması

Bir sınıf türünden değişken bildirildiğinde yalnızca potansiyel olarak adres tutacak bir referans elde edilmiş olur. Ayrıca sınıf nesnesinin kendisini yaratıp onun adresini referansa yerleştirmek gerekir. Sınıf nesnelerini yaratmak için new operatörü kullanılmaktadır. new operatörünün genel biçimi şöyledir:

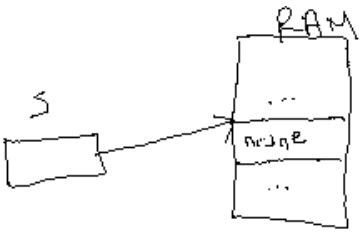
```
new <sınıf ismi>([argüman listesi])
```

new operatörü nesneyi tahsis eder ve onun adresini bize verir. new operatörüyle verilen adres aynı sınıf türünden bir referansa atanmalıdır. Örneğin:

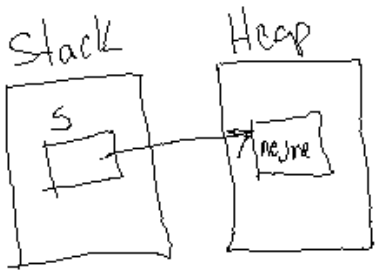
```
Sample s;
s = new Sample();
```

ya da:

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



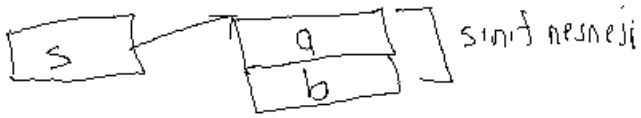
new operatörüyle yaratılan nesne belleğin "heap" denilen bölümünde yer alır. Halbuki bütün yerel değişkenler stack'tedir. Yukarıdaki örnekte s yerel ise stack'te bulunacaktır. Ancak s'in gösterdiği yerdeki nesne heap'tedir.



new operatörüyle tahsis edilen alana sınıf nesnesi (class object) denilmektedir. Sınıf türünden değişkenlere de kısaca "referans" denir.

Bir sınıf nesnesi için o sınıfın static olmayan veri elemanlarının toplam uzunluğu kadar yer ayrılmaktadır. Sınıfın static veri elemanları ve metotları new ile tahsis edilen alanda yer kaplamaz. Sınıfın static olmayan veri elemanları ardışıl bir blok oluşturur. new operatörü bize bu bloğun başlangıç adresini verir. Örneğin:

```
Sample s;

s = new Sample();
```



Sınıf nesneleri birden fazla parçadan oluşan bileşik nesnelerdir.

Sınıfın metotları belleğin code bölümündedir. Sınıfın static veri elemanları ileride ele alınacaktır.

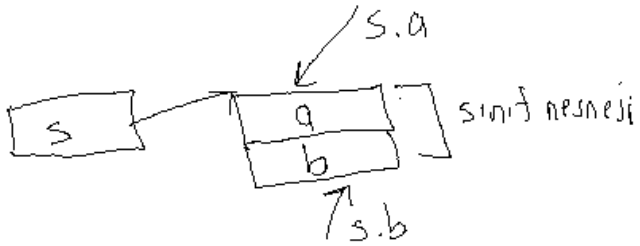
Sınıfın Veri Elemanlarına Erişim ve Nokta Operatörü

Bir bir sınıf türünden referans a da bu sınıfın static olmayan bir veri elemanı olmak üzere r.a ifadesi ile r referansının gösterdiği yerdeki nesnenin a parçasına erişilir. Nokta operatörü iki operandlı araek bir operatördür. Örneğin:

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

            s = new Sample();

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

```
            System.Console.WriteLine("{0}, {1}", s.a, s.b);
        }
    }

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

        public static void Foo()
        {
            //...
        }

        public void Bar()
        {
            //...
        }
    }
}
```

Bu örnekte s Sample türündendir. s.a ve s.b int türdendir.

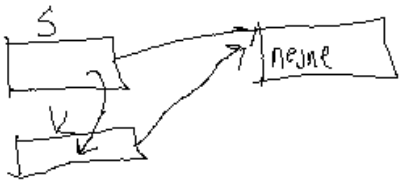


Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Date d = new Date();

            d.day = 5;
            d.month = 12;
            d.year = 2015;

            System.Console.WriteLine("{0}/{1}/{2}", d.day, d.month, d.year);
        }
    }

    class Date
    {
        public int day;
        public int month;
        public int year;
        //...
    }
}
```

Aynı Türden Referansların Birbirlerine Atanması

Aynı türden iki referans birbirlerine atanabilir. Bu durumda onların içindeki adresler atanmış olur. Böylece bunlar aynı nesneyi gösterir hale gelir. Örneğin:




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

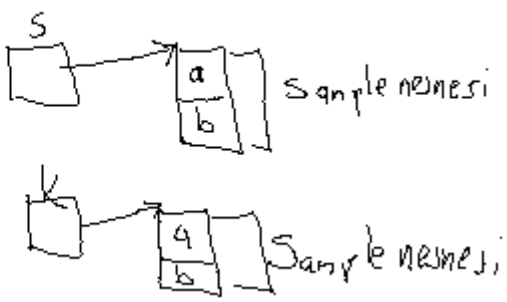
Burada artık s.a ifadesi ile k.a ifadesi aynı değişkeni belirtir. Bu örnekte bir tane nesneyi iki ayrı referans göstermektedir. Örneğin:

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

            s = new Sample();

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

            k = s;

            System.Console.WriteLine("{0}, {1}", k.a, k.b);
        }
    }

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

        public static void Foo()
        {
            //...
        }

        public void Bar()
        {
            //...
        }
    }
}
```

Birden Fazla Nesnenin Yaratılması Durumu

Her new işlemi ile farklı bir nesne yaratılır. Örneğin:

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



Atrık s.a ile k.a aynı değişkenler değildir. Örneğin:

```
namespace CSD
```

```
{
    class App
    {
        public static void Main()
        {
            Sample s, k;

            s = new Sample();
            k = new Sample();

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

            k.a = 30;
            k.b = 40;

            System.Console.WriteLine("{0}, {1}", s.a, s.b);
            System.Console.WriteLine("{0}, {1}", k.a, k.b);
        }
    }

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

        public static void Foo()
        {
            //...
        }

        public void Bar()
        {
            //...
        }
    }
}
```

Örneğin:

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

            x.real = 10;
            x.imag = 5;

            y.real = 30;
            y.imag = 2;

            System.Console.WriteLine("{0}+{1}i", x.real, x.imag);
            System.Console.WriteLine("{0}+{1}i", y.real, y.imag);
        }
    }

    class Complex
    {
        public double real;
        public int imag;
        //...
    }
}
```

Sınıfların static Veri Elemanları

Sınıfın static veri elemanlarının toplamda tek bir kopyası vardır. Bunlar için new işlemi sırasında yer ayrılmaz. Program çalıştırılğında yer ayrılır, programın sonuna kadar bunlar bellekte kalır. Sınıfın static veri elemanlarına sınıf ismi belirtilerek nokta operatörüyle erişilir. Örneğin:

```
Sample.c = 10;
```

Peki neden sınıfın static veri elemanlarına referansla değil de sınıf ismiyle erişilmektedir? Eğer nunlara referansla erişilseydi sanki static veri elemanları o referansın gösterdiği nesnenin içerisindeymiş gibi bir yanlış anlaşılma oluşurdu. Halbuki onlardan toplamda tek bir kopya vardır. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample.c = 10;
            System.Console.WriteLine(Sample.c);
        }
    }

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

        public static void Foo()
        {
            //...
        }

        public void Bar()
        {
            //...
        }
    }
}
```

Kavramlar, Nesneler ve Sınıflar

Kavramlar bizim kafamızdadır ve gerçek dünyada yer kaplamazlar. Örneğin ağaç, doktor, kedi gerçek var olan şeyler değildir. Biz belirli özellikteki nesneleri birbirine benzeterek onlara isim veririz. Aslında öyle birşey yoktur. Kavramlar bizim karmaşık dünyayı algılamamız için uydurduğumuz soyut şeylerdir. İşte NYPT'de kavramlar sınıflara karşılık gelir. Bir Doktor sınıfı bildirdiğimizde aslında bir doktor oluşturmuş olmayız. Doktorların ortak özelliklerini belirten bir kavram oluşturmuş oluruz. Doktor sınıfı kendi başına bellekte yer kaplamaz (tıpkı Doktor kavramının aslında fiziksel dünyada yer kaplamadığı gibi). Bir sınıf türünden new işlemi yaptığımızda artık gerçek bir nesne yaratılmış olur. O artık yer kaplar. İngilizce sınıf türünden new ile yaratılan nesnelere "instance (örnek)" de denilmektedir. Buradan anlatılmak istenen şey "bizim new yaptığımızda o kavram türünden bir örnek nesne oluşturduğumuzdur".

İşte bir proje NYPT'ne göre modellenecekse önce proje içerisinde kavramları sınıflarla temsil ederiz. Sonra gerçek nesneleri sınıf nesneleri biçiminde new operatörüyle yaratırız. Örneğin, bir hastane otomasyonunda Hastane, Doktor, Hemşire, Hasta, İlaç, Ameliyatnahe, Hastalık gibi kavramlar birer sınıfla temsil edilir. Hastanede 10 doktor 15 hemişre çalışıyorsa Doktor sınıfı türünden new operatörüyle 10 nesne Hemşire sınıfı türünden new operatörüyle 15 nesne yaratırız. Böyle işlemlere devam ederiz.

Örneğin tarih kavramı Date isimli bir sınıfla temsil edilebilir:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Date d = new Date();

            d.day = 23;
            d.month = 4;
            d.year = 1920;

            Date k = new Date();
            k.day = 25;
            k.month = 4;
            k.year = 2015;

            System.Console.WriteLine("{0}/{1}/{2}", d.day, d.month, d.year);
            System.Console.WriteLine("{0}/{1}/{2}", k.day, k.month, k.year);
        }
    }

    class Date
    {
        public int day;
        public int month;
        public int year;
        //...
    }
}
```

Bir sınıf bildiriminin kendisi henüz hiç new yapılamadan bellekte yer kaplamaz. Sınıf bildirimi derleyiciye new yapıldığı taktirde o nesnenin hangi parçaları olacağını vs. anlatır.

Sınıfların static Olmayan Metotları

Sınıfların static olmayan metotları referans ve nokta operatörüyle çağrılır. Halbuki static metotlar sınıf ismi ve nokta operatörüyle çağrılmaktadır. Örneğin:

```
Sample.Foo();       // Foo static metot

Sample s = new Sample();
s.Bar();            // Bar static olmayan metot
```

Aslında aynı durum veri elemanları için de söz konusudur. Anımsanacağı gibi sınıfın static olmayan veri elemanları referansla, static veri lemenları sınıf ismiyle kullanılıyordu.

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample.Foo();   // static metot
            Sample.c = 100; // static veri elemanı

            Sample s = new Sample();

            s.a = 10;       // static olmayan veri elemanı
            s.b = 20;       // static olmayan veri elemanı
            s.Bar();        // static olmayan metot
        }
    }
```

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

        public static void Foo()
        {
            System.Console.WriteLine("Foo");
        }

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

Sınıfın static olmayan veri elemanları sınıfın static olmayan metotları tarafından doğrudan kullanılabilirler. Ancak sınıfın static olmayan veri elemanları sınıfın static metotları tarafından doğrudan kullanılamazlar. Örneğin:

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

            s = new Sample();
            s.Set(10, 20);
            s.Disp();
        }
    }

    class Sample
    {
        public int a;
        public int b;

        public void Set(int x, int y)
        {
            a = x;
            b = y;
        }

        public void Disp()
        {
            System.Console.WriteLine("{0}, {1}", a, b);
        }
    }
}
```

Sınıfn static olmayan metotları içerisinde kullanılan static olmayan veri elemanları metot hangi referansla çağrılmışsa, o referansın gösterdiği yerdeki nesnenin veri elemanlarıdır. Yani örneğin:

```
s.Foo();
```

böyle bir çağırmada Foo'nun içerisinde kullandığımız static olmayan veri elemanları aslında s referansının gösterdiği yerdeki nesnesnin veri elemanlarıdır. Örneğin:

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

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

            s.Set(10, 20);
            k.Set(30, 40);

            s.Disp();
            k.Disp();
        }
    }

    class Sample
    {
        public int a;
        public int b;

        public void Set(int x, int y)
        {
            a = x;
            b = y;
        }

        public void Disp()
        {
            System.Console.WriteLine("{0}, {1}", a, b);
        }
    }
}
```

Örneğin:

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

            s.Set(10, 20);

            k = s;
            k.Disp();       // 10, 20
        }
    }

    class Sample
    {
        public int a;
        public int b;

        public void Set(int x, int y)
        {
            a = x;
            b = y;
        }

        public void Disp()
        {
            System.Console.WriteLine("{0}, {1}", a, b);
        }
    }
}
```

Sınıfın stataic olmayan bir metodu sınıfın başka bir static olmayan metodunu doğrudan çağırabilir. Fakat static bir metodu static olmayan metodunu doğrudan çağıramaz. (Yani static metotlar sınıfın static olmayan veri

elemanlarını kullanamadıkları gibi, sınıfın static olmayan metotlarını da çağıramazlar.)

Sınıfın static olmayan bir metodu sınıfın başka bir static olmayan metodunu çağırdığında çağrılan metot, çağıran metot hangi referansla çağrılmışsa aynı referabla çağrılmış gibi etki gösterir. Örneğin:

```
Sample s = new Sample();

s.Foo();
//...

class Sample
{
	public int a;
	public int b;

	public void Foo()
	{
		//...
		Bar(); // Burada sanki Bar da s ile çağrılmış gibi işlem görür
		//...
	}

	public void Bar()
	{
		//...
	}
	//...
}
```

Sınıfın static metotları referansla çağrılmadığı için sınıfın static olmayan metotlarını çağıramaz (eğer çağırabilseydi bu metot içerisindeki veri elemanların hangi nesnenin elemanları olduğu anlaşılamazdı).

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Date d = new Date();

            d.Set(10, 9, 2014);
        }
    }

    class Date
    {
        public int day, month, year;

        public void Set(int d, int m, int y)
        {
            day = d;
            month = m;
            year = y;

            Disp();
        }

        public void Disp()
        {
            System.Console.WriteLine("{0}/{1}/{2}", day, month, year);
        }
    }
}
```

Sınıfın static olmayan metotları sınıfın static veri elemenlarını doğrudan kullanabilir ve static metotlarını doğrudan çağırabilir. (Zaten bunlar için bir referansa gereksinim yoktur.) Örneğin:

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

            s.Set(10, 20);

            k = s;
            k.Disp();
        }
    }

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

        public void Set(int x, int y)
        {
            a = x;
            b = y;

            c = 100;        // geçerli
        }

        public void Disp()
        {
            System.Console.WriteLine("{0}, {1}, {2}", a, b, c);
            Foo();      // geçerli
        }

        public static void Foo()
        {
            //...
        }
    }
}
```

Sınıfın static bir metodu sınıfın static veri elemanlarını doğrudan kullanabilir ve static metotlarını doğrudan çağırabilir.

Biz bir sınıf incelerken bir metodun static olmayan metot olduğunu gördüğümüzde ne düşünmeliyiz? Düşüncemiz şöyle olmalıdır: “Bizim bu metodu çağırabilmemiz için elimizde o sınıf türünden bir referansın bulunuyor olması gerekir. Elimizde bir referans yoksa önce bir referans bildirip, o referans için new işlemi yaptıktan sonra o metodu çağırabiliriz. Halbuki bir sınıfın static bir metodunu gördüğümüzde “hiç referans olmadan sınıf ismiyle onu çağırabileceğimizi” anlamalıyız.

Kim Kimi Kullanır, Kim Kimi Çağırır?

Sınıfn elemanı denildiğinde bildirimi sınıf içerisinde yapılan metotlar ve veri elemanları anlaşılır. (Sınıfın başka başka elemanları da olabilir. İleride görülecektir.) Yani metotlar da veri elemanları da sınıfın elemanlarıdır. Sınıfın static elemanları denildiğinde sınıfın static metotları ve static veri elemanları, sınıfın static olmayan elemanları denildiğinde ise static olmayan veri elemanları ve static olmayan metotlar anlaşılmalıdır. Buna göre yukarıda anlatılan kurallar basit bir biçimde şöyle özetlenebilir:

1) Sınıfın static olmayan metotları sınıfın hem static elemanlarını hem de static olmayan elemanlarını doğrudan kullanabilir.

2) Sınıfın static metotları yalnızca sınıfın static elemanlarını doğrudan kullanabilir.

Rastgele Sayı (Rassal) Üretilmesi ve Random Sınıfı

Rassal sayılar rassal süreçlerin yardımıyla üretilebilirler. (Örneğin bir torbadan pul çekmek, bir zarı atmak gibi). Ancak bilgisayar rassal sayıları böyle rassal olaylar sonucunda değil aritmetik hesaplarla elde eder. Bu nedenle bilgisayarın elde ettiği bu rassal sayılara "Sahte Rassal Sayılar (Pseudo Random Numbers)" denilmektedir.

Olasılık bir teorik limit değeridir. Bir parayı attığımızda yazı gelme olasılığı 0.5'tir. Ancak parayı 10 kere attığımızda 5 kere yazı geleceği garanti değildir. Fakat parayı sonsuz kere atarsak oran 0.5 olacaktır.

.NET'te rassal sayı üretmek için System isim alanı içerisindeki Random sınıfı kullanılmaktadır. Random sınıfının static olmayan Next metotları her çağrıldığında bize rastgele bir sayı verir. Sınıfın overload edilmiş üç Next metodu vardır:

```
public int Next()
public int Next(int maxValue)
public int Next(int minValue, int maxValue)
```

Parametresiz Next metodu [0, +2147483647] arasında int parametreli Next metodu [0, maxValue) arasında, ve iki int parametreli Next metodu ise [minValue, maxVale) arasında rastgele bir değer geri döndürür. Örneğin:

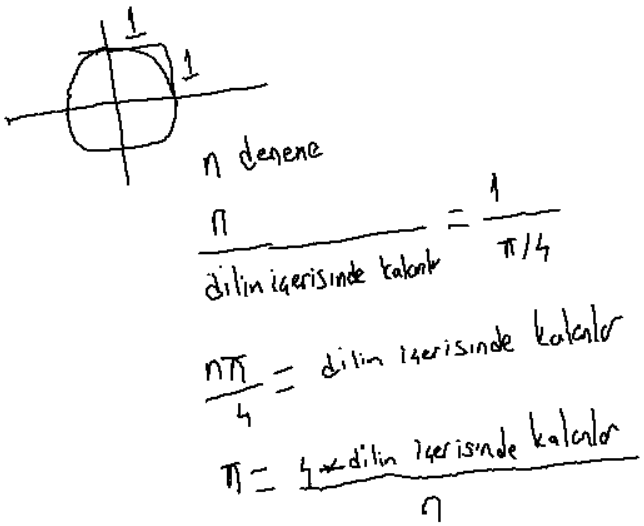
```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            System.Random r = new System.Random();
            int val;

            for (int i = 0; i < 10; ++i)
            {
                val = r.Next(100);
                System.Console.Write("{0} ", val);
            }
            System.Console.WriteLine();
        }
    }
}
```

Yazı tura atma oranına ilişkin bir örnek:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            long n;
            long head, tail;
            System.Random r;
            int val;

            System.Console.Write("Deney sayısını giriniz:");
            n = long.Parse(System.Console.ReadLine());

            head = tail = 0;
            r = new System.Random();
            for (long i = 0; i < n; ++i)
            {
                val = r.Next(2);
                if (val == 0)
```

```
                ++tail;
            else
                ++head;
        }

        double headRatio, tailRatio;

        headRatio = (double)head / n;
        tailRatio = (double)tail / n;

        System.Console.WriteLine("Tura oranı = {0}", headRatio);
        System.Console.WriteLine("Yazı oranı = {0}", tailRatio);
    }
  }
}
```

Sınıf Çalışması: [0, 1] arasında rastgele gerçek sayı üreterek dairenin dörtte birinin alanını karenin alanına bölme yoluyla pi sayısını elde ediniz:



Çözüm:

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

            System.Console.Write("Deneme sayınızı giriniz:");
            long n = long.Parse(System.Console.ReadLine());
            int quadrentCount = 0;
            double x, y, pi;

            for (int i = 0; i < n; ++i)
            {
                x = (double)r.Next() / System.Int32.MaxValue;
                y = (double)r.Next() / System.Int32.MaxValue;
                if (System.Math.Sqrt(x * x + y * y) < 1)
                    ++quadrentCount;
            }

            pi = 4D * quadrentCount / n;
            System.Console.WriteLine(pi);
        }
    }
}
```

Sınıf Çalışması: System.Console.ReadKey(true).KeyChar ifadesini kullanarak tuşa her basıldığında ekrana "Ali", "Veli", "Selam", "Ayşe", "Fatma" isimlerindne rastgele birini yazdıran 'q' tuşuna basıldığında çıkan programı yazınız.

Çözüm:

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

            while (System.Console.ReadKey(true).KeyChar != 'q')
            {
                val = r.Next(5);
                switch (val)
                {
                    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;
                }
            }
        }
    }
}
```

Metotlar Static mi Olsun Static Olmasın mı?

Çoğu zaman C#'ı yeni öğrenen programcılar bir metodu static yapıp yapmama konusunda tereddüt etmektedir. Bu konuda aşağıdaki reçeteyi verebiliriz:

1) Metot eğer sınıfın static olmayan elemanlarını doğrudan kullanıyorsa zaten onu "static" yapamayız. Mecburen onu "static olmayan metot" yapmak zorundayız.

2) Metot sınıfın hiçbir static olmayan elemanını doğrudan kullanmıyorsa teorik olarak o metot static metot da yapılabilir, static olmayan metot da yapılabilir. Bu durumda metodun static yapılması doğru ve iyi tekniktir. Çünkü metot static olmayan yapılırsa boşuna o metodu çağırabilmek için bir nesne yaratmak zorunda kalırız. Halbuki o metot o nesnenin elemanları zaten kullanmıyor durumdadır.

3) Biz bir sınıfın bir metodunun static olmayan bir metot olduğunu gördüğümüzde kesinlikle o metodun sınıfın static olmayan bir elemanını kullandığını düşünmeliyiz. (Eğer kullanmıyor olsaydı sınıfı tasarlayan kişi onu static yapardı.)

4) Static olmayan metotlar çağrıldığında doğrudan ya da dolaylı olarak bu çağrı static olmayan bir veri elemanının kullanılmasına yol açacaktır. (Örneğin static olmayan Foo sınıfın hiçbir static olmayan veri elemanını kullanmıyor olsun fakat Bar isimli static olmayan bir metodunu çağırıyor olsun. İşte demek ki dolaylı olarak Bar static olmayan veri elemanlarını kullanıyor durumdadır. Zaten o da kullanmasaydı o static olurdu bu durumda Foo da static olurdu).

Metotların Sınıf Türünden Referans Parametrelerine Sahip Olması Durumu (Sınıf Nesnelerinin Referans Yoluyla Metotlara Aktarılması)

Bir metodun parametre değişkeni bir sınıf türünden referans olabilir. Bu durumda bu metot aynı sınıf türünden bir referansla çağrılmalıdır. Böylece nesnenin adrese metoda geçirilmiş olur. Metot içerisinde biz o nesneye erişebiliriz. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Date d = new Date();

            d.day = 10;
            d.month = 3;
            d.year = 2010;

            Foo(d);

            System.Console.WriteLine("{0}/{1}/{2}", d.day, d.month, d.year);
        }

        public static void Foo(Date k)
        {
            System.Console.WriteLine("{0}/{1}/{2}", k.day, k.month, k.year);

            k.day = 11;
            k.month = 4;
            k.year = 2011;
        }
    }

    class Date
    {
        public int day;
        public int month;
        public int year;
        //...
    }
}
```

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Date d = new Date();

            d.Set(10, 12, 2007);
            Foo(d);
            d.Disp();
        }

        public static void Foo(Date k)
        {
            k.Disp();
            k.Set(1, 1, 1900);
        }
    }


    class Date
    {
        public int day;
```

```
        public int month;
        public int year;

        public void Set(int d, int m, int y)
        {
            day = d;
            month = m;
            year = y;
        }

        public void Disp()
        {
            System.Console.WriteLine("{0}/{1}/{2}", day, month, year);
        }
    }
}
```

Örneğin:

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

            x.real = 3;
            x.imag = 2;

            Complex y = new Complex();

            y.real = 7;
            y.imag = 3;

            Disp(x);
            Disp(y);
        }

        public static void Disp(Complex z)
        {
            System.Console.WriteLine("{0}+{1}i", z.real, z.imag);
        }
    }

    class Complex
    {
        public int real;
        public int imag;
    }
}
```

Metotların Geri Dönüş Değerlerinin Bir Sınıf Türünden Olması Durumu

Bir metodun geri dönüş değeri bir sınıf türünden olabilir. Örneğin:

```
public static Sample Foo()
{
      //...
}
```

Bu durumda metot çağrıldığında bize o sınıf türünden bir nesnenin adresini (yani referansını) verir. Örneğin:

```
namespace CSD
{
    class App
    {
```

```
        public static void Main()
        {
            Date date;

            date = GetDate();
            date.Disp();
        }

        public static Date GetDate()
        {
            Date date = new Date();

            System.Console.Write("Gun:");
            date.day = int.Parse(System.Console.ReadLine());

            System.Console.Write("Ay:");
            date.month = int.Parse(System.Console.ReadLine());

            System.Console.Write("Yıl:");
            date.year = int.Parse(System.Console.ReadLine());

            return date;
        }
    }

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

        public void Set(int d, int m, int y)
        {
            day = d;
            month = m;
            year = y;
        }

        public void Disp()
        {
            System.Console.WriteLine("{0}/{1}/{2}", day, month, year);
        }
    }
}
```

Sınıfların Başlangıç Metotları (Constructors)

Bir sınıf nesnesi yaratılırken new operatörü tarafından otomatik olarak çağrılan sınıfın static olmayan metotlarına başlangıç metotları (constructors) denilmektedir. (construct sözcüğü İngilizce “yapmak, inşa etmek” anlamına geliyor. Bu nedenle “başlangıç metotları” için Türkçe bazı kaynaklarda "yapıcı metotlar" ismi de kullanılmaktadır.) new operatörü önce sınıfın static olmayan veri elemanlarını içeren sınıf nesnesini heap’te yaratır. Sonra bu nesnenin elemanlarını sıfırlar. Sonra da yaratılmış olan nesne için sınıfın başlangıç metodunu çağırır. new operatörü başlangıç metodu çağrıldıktan sonra işini bitirip geri dönmektedir.

Başlangıç metotları sınıfın sınıf ismiyle aynı isimli metotlarıdır. Başlangıç metotlarının geri dönüş değerleri diye bir kavramları yoktur. Bu nedenle bildirim sırasında başlangıç metotlarında geri dönüş değeri türü yerine birşey yazılmaz. Eğer yazılarsa (void da dahil olmak üzere) error oluşur. Örneğin:

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

```
            s = new Sample();
            s.Foo();
            s.Bar();
        }
    }

    class Sample
    {
        private int a;
        private int b;

        public Sample()
        {
            System.Console.WriteLine("I am a constructor");
        }

        public void Foo()
        {
            System.Console.WriteLine("I am Foo");
        }

        public void Bar()
        {
            System.Console.WriteLine("I am Bar");
        }
    }
}
```

Sınıfın başlangıç metotları overload edilebilir. Yani sınıfın parametrik yapıları farklı olmak üzere birden fazla başlangıç metodu bulunabilir. Sınıfın parametresiz başlangıç metoduna "default başlangıç metodu (default constructor)" denilmektedir.

new operatörü ile nesne yaratılırken new operatörüne verilen argüman listesi hangi başlangıç metodunun çağrılacağını belirtir. Örneğin:

```
s = new Sample();              // default başlangıç metodu
k = new Sample(10, 20);        // int, int parametreli başlangıç metodu
```

En uygun başlangıç metodu yine "overload resolution" kurallarıyla tespit edilmektedir. Sınıfın tüm başlangıç metotları aday metotlar olarak seçilir. Bunlar arasından uygunlar ve en uygun metot belirlenmeye çalışılır. Örneğin:

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

            s = new Sample();
            k = new Sample(10, 20);
        }
    }

    class Sample
    {
        public int a;
        public int b;

        public Sample()
        {
            System.Console.WriteLine("Default conctructor");
        }

        public Sample(int a, int b)
        {
```

```
            System.Console.WriteLine("int, int constructor");
        }

        public void Foo()
        {
            System.Console.WriteLine("I am Foo");
        }

        public void Bar()
        {
            System.Console.WriteLine("I am Bar");
        }
    }
}
```

Örneğin:

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

            d = new Date();
            d.Disp();

            Date k;

            k = new Date(12, 12, 2015);
            k.Disp();
        }
    }

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

        public Date()
        {
            day = 1;
            month = 1;
            year = 1900;
        }

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

        public void Disp()
        {
            System.Console.WriteLine("{0}/{1}/{2}", day, month, year);
        }
    }
}
```

Başlangıç metotlarında return kullanılabilir fakat yanına bir ifade yazılamaz.

new operatörü önce heap'te tahsisatı yapar. Sonra o alanı sıfırlar sonra da başlangıç metodunu çağırır. Yani biz başlangıç metodu içerisinde sınıfın veri elemanına bir değer atamamışsak o elemanda sıfır değeri gözükecektir. Fakat atamışsak o değer gözükecektir. Örneğin:

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

            s = new Sample();
            s.Disp();       // a = 100, b = 0
        }
    }

    class Sample
    {
        private int a;
        private int b;

        public Sample()
        {
            a = 100;
        }

        public void Disp()
        {
            System.Console.WriteLine("a = {0}, b = {1}", a, b);
        }
    }
}
```

Örneğin:

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

            s = new Sample(100, 200);
            s.Disp();       // a = 100, b = 200
        }
    }

    class Sample
    {
        public int a;
        public int b;

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

        public void Disp()
        {
            System.Console.WriteLine("a = {0}, b = {1}", a, b);
        }
    }
}
```

Örneğin:

```
namespace CSD
{
    class App
```

```
    {
        public static void Main()
        {
            Complex x = new Complex();
            Complex y = new Complex(10, 20);

            x.Disp();
            y.Disp();
        }
    }

    class Complex
    {
        public double real;
        public double imag;

        public Complex()
        {}

        public Complex(double r, double i)
        {
            real = r;
            imag = i;
        }

        public void Disp()
        {
            System.Console.WriteLine("{0}+{1}i", real, imag);
        }
    }
}
```

Başlangıç metotlarının içerisinde kullandığımız sınıfın veri elemanları o anda yaratılmış olan nesnenin veri elemanlarıdır. Yani yaratılmış bir nesnenin veri elemanlarına erişim en erken sınıfın başlangıç metotlarında yapılabilir. Örneğin:

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

Burada new önce heap'te tahsisatı yapacak, sonra sınıfın başlangıç metodunu çağıracak sonra nesnenin adresiyle geri dönüp bu s'e atanacaktır.

Bir sınıf için programcı hiçbir başlangıç metodu yazmamışsa derleyici default başlangıç metodunu (yani parametresiz başlangıç metodunu) içi boş olarak yazar. Örneğin aşağıdaki sınıf için new Sample() biçiminde nesne yaratmak soruna yol açmaz:

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

            s = new Sample();      // geçerli
            s.Disp();
        }
    }

    class Sample
    {
        private int a;
        private int b;

        public void Disp()
        {
            System.Console.WriteLine("a = {0}, b = {1}", a, b);
        }
```

```
    }
}
```

Başlangıç Metotlarına Neden Gereksinim Duyulmaktadır?

Sınıfların başlangıç metotları temel olarak iki amaçla kullanılmaktadır:

1) Nesne yaratıldığında veri elemanlarına birtakım ilkdeğerleri vermek.

2) Nesne yaratıldığında birtakım ilk işleri arka planda yapmak

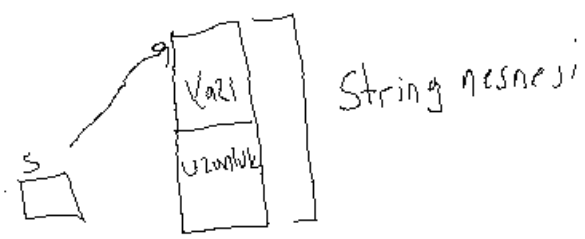
Şüphesiz başlangıç metotları yerine tüm bu işler başka bir metoda da yaptırılabilirdi. Ancak başlangıç metotları bu işlemleri arka planda kodda yer kaplamayacak biçimde yapmaktadır. Böylece kodun daha sade gözüknkmesini sağlamaktadır.

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            Complex x = new Complex();
            Complex y = new Complex(10, 20);

            x.Disp();
            y.Disp();
        }
    }

    class Complex
    {
        public double real;
        public double imag;

        public Complex()
        { }

        public Complex(double r, double i)
        {
            real = r;
            imag = i;
        }

        public void Disp()
        {
            System.Console.WriteLine("{0}+{1}i", real, imag);
        }
    }
}
```

String Sınıfı

System isim alanı içerisindeki String sınıfı yazıları tutan ve onlar üzerinde işlem yapan önemli sınıftır. Bir String nesnesi yaratıldığında nesnenin içerisinde yazı ve onun uzunluğu tutulur.

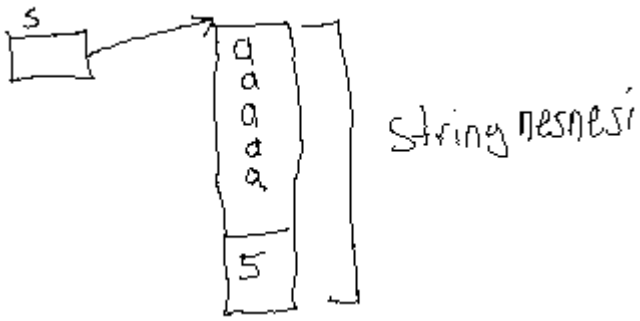


String sınıfının default başlangıç metodu yoktur. Bir String nesnesi String sınıfının çeşitli başlangıç metotlarıyla yaratılabilir. Örneğin aşağıdaki başlangıç metoduyla belirli bir karakterden n tane olacak biçimde String nesnesi oluşturulabilir:

```
public String(char c, int count)
```

Örneğin:

```
System.String s = new System.String('a', 5);
```



Console sınıfının String parametreli Write ve WriteLine metotları verilen referansın gösterdiği yerdeki String nesnesinin içerisindeki yazıyı yazdırır. Örneğin:

System.String sınıfı çok kullanıldığı için string anahtar sözcüğüyle de temsil edilmiştir. Yani System.String demekle string demek tamamen aynı anlamdadır. Örneğin:

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

            s = new string('a', 5);
            System.Console.WriteLine(s);
        }
    }
}
```

Aslında uygulamada string nesneleri new operatörü ile değil daha çok otomatik olarak yaratılmaktadır. Şöyle ki: C#'ta ne zaman biz iki tırnak içerisinde bir yazı yazsak derleyici new operatörü ile bir string nesnesi yaratır, iki tırnak içerisindeki yazıyı nesnenin içerisine yerleştirir ve nesnenin referansını bize verir. Yani örneğin C#'ta "Ankara" gibi bir ifade "Bir string nesnesi yarat, içerisine Ankara yazısını yerleştir ve onun referansını ver" anlamına gelmektedir. Dolayısıyla iki tırnak içerisindeki yazıları biz doğrudan string referanslarına atayabiliriz. Örneğin:

```
string s;

s = "Ankara";
System.Console.WriteLine(s);
```

string sınıfının int türden Length isimli read-only property elemanı yazınının bize uzunluğunu verir.

Anahtar Notlar: Property veri elemanı gibi kullanılan metotlardır. Property kavramı ileride ele alınacaktır.

Örneğin:

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

            System.Console.WriteLine(s.Length);     // 6
        }
    }
}
```

Console sınıfının parametresiz bir ReadLine metodu vardır:

```
public static string ReadLine()
```

Bu metot klavyeden bir yazı girilip ENTER tuşuna basılana kadar bekler. Girilen yazıyı bir string nesnesinin içerisine yerleştirir ve o nesnenin referansıyla geri döner. Örneğin:

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

            System.Console.Write("Bir yazı giriniz:");
            s = System.Console.ReadLine();
            System.Console.WriteLine("Yazı: {0}, Uzunluğu: {1}", s, s.Length);
        }
    }
}
```

string sınıfının int türden read-only indeskleyicisi string nesnesi içerisindeki yazının istediğimiz bir karakterini bize verir.

Anahtar Notlar: Bir sınıf türünden referansı köşeli parantez operatörü ile kullanabilmek için o sınıfın indeksleyici (indexer) denilen bir elemanının olması gerekir. String sınıfında da int parametreli bir indeksleyici vardır. İndeksleyici konusu kursumuzun son bölümlerinde ele alınmaktadır.

string sınıfının indeksleyicisi köşeli parantezler içerisinde bizden bir indeks numarası alır. Yazının o indeksteki karakterini bize verir. Yazının ilk karakteri sıfırıncı indekstedir. Örneğin:

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

            System.Console.Write("Bir yazı giriniz:");
            s = System.Console.ReadLine();
            for (int i = 0; i < s.Length; ++i)
                System.Console.WriteLine(s[i]);
        }
    }
}
```

Eğer köşeli parantez içerisindeki indeks negatifse ya da büyükse bu durumda excedption oluşur ve program çöker.

Sınıf Çalışması: Klavyeden bir yazı alınız ve bunu tersten yazdırınız.

Çözüm:

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

            System.Console.Write("Bir yazı giriniz:");
            s = System.Console.ReadLine();
            for (int i = s.Length - 1; i >= 0; --i)
                System.Console.Write(s[i]);
            System.Console.WriteLine();
        }
    }
}
```

Bir string nesnesinin karakterleri nesne yaratıldıktan sonra artık hiç değiştirilemez. Bu string sınıfının çok önemli bir özelliğidir. Örneğin:

```
string s = "Ankara";
s[0] = 'a';           // error!
```

Tabii biz bir string referansını başka bir yazıyı gösterecek hale getirebiliriz. Bu işlem yazı üzerinde bir değişiklik yapmamaktadır. Örneğin:

```
string s;

s = “Ankara”;
s = “İstanbul”;
```

Biğz burada yazının karakterlerini değiştirmiş olmuyoruz, yeni bir string nesnesi yaratmış ve s referansının artık o yeni string nesnesini göstermesini sağlamış oluyoruz. Bu nedenle string nesnesi içerisindeki yazı üzerinde işlem yapan metotlar asıl yazıyı değiştiremediklerinden değiştirilmiş yeni bir yazıyı bize verirler.

string sınıfının ToLower ve ToUpper static olmayan metotları bize küçük harfe ve büyük harfe dönüştürülmüş yeni bir string nesnesi verirler. Metotların parametrik yapıları şöyledi:

```
public string ToLower()
public string ToUpper()
```

Örneğin:

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

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

            k = s.ToUpper();
            System.Console.WriteLine(k);

            k = s.ToLower();
            System.Console.WriteLine(k);
```

```
        }
    }
}
```

String sınıfının overload edilmiş iki Substring metodu vardır.

```
public string Substring(int startIndex)
public string Substring(int startIndex, int length)
```

Birinci Substring metodu belli bir indeksten başlayarak sonuna kadar olan yazıyı alır ve bize bir string nesnesi olarak verir. İkinci Substring metodu belli bir indeksten itibaren belli sayıda karakteri alarak bize bir string nesnesi biçiminde verir. Eğer indeks ya da uzunluk bakımından sınır dışına çıkılırsa exception oluşur (yani program çöker). Örneğin:

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

            s = "istanbul";

            k = s.Substring(2);
            System.Console.WriteLine(k);        // tanbul

            k = s.Substring(2, 3);
            System.Console.WriteLine(k);        // tan
        }
    }
}
```

Sınıf Çalışması: Klavyeden gg/aa/yyyy formatında bir yazı okuyunuz. Bu yazıdan Substring metotlarını kullanarak gg, aa ve yyyy kısımlarını çekiniz. Sonra bu kısımları gg-aa-yyyy biçiminde yeniden yazdırınız.

Çözüm:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string date;
            string day, month, year;

            System.Console.Write("gg/aa/yyyy formatında bir tarih giriniz:");
            date = System.Console.ReadLine();

            day = date.Substring(0, 2);
            month = date.Substring(3, 2);
            year = date.Substring(6, 4);

            System.Console.WriteLine("{0}-{1}-{2}", day, month, year);
        }
    }
}
```

string sınıfının IndexOf isimli static olmayan metotları yazı içerisindeki bir karakteri ya da bir yazıyı arar. Bulursa bulduğu yerin indeks numarasıyla geri döner. Bulamazsa -1 değeri ile geri döner. Overload edilmiş pek çok IndexOf metodu vardır. En önemlileri şunlardır:

```
public int IndexOf(char value)
```

```
public int IndexOf(string value)
public int IndexOf(char value, int startIndex)
public int IndexOf(string value, int startIndex)
```

Örneğin:

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

            s = "eskişehir";
            index = s.IndexOf('i');
            if (index == -1)
                System.Console.WriteLine("Cannot find!..");
            else
                System.Console.WriteLine("Found at the {0} index", index);
        }
    }
}
```

Sınıf Çalışması: Klavyeden bir yazı isteyiniz. Yazının içerisinde küme parantezleri arasında bir kısım olsun. Küme parantezlerinin yerini IndexOf metotlarıyla bulunuz. Onun arasındaki yazıyı Substring metoduyla elde edip yazdırınız. Örneğin:

Bir yazı giriniz: bugün {hava} çok güzel
hava

Çözüm:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string text;
            int indexBeg, indexEnd;

            System.Console.Write("Bir yazı giriniz:");
            text = System.Console.ReadLine();

            if ((indexBeg = text.IndexOf('{')) == -1)
            {
                System.Console.WriteLine("Yanlış giriş!");
                return;
            }

            if ((indexEnd = text.IndexOf('}')) == -1)
            {
                System.Console.WriteLine("Yanlış giriş!");
                return;
            }

            if (indexBeg >= indexEnd)
            {
                System.Console.WriteLine("Yanlış giriş!");
                return;
            }

            string result = text.Substring(indexBeg + 1, indexEnd - indexBeg - 1);
            System.Console.WriteLine(result);
```

```
        }
    }
}
```

Diğer bir seçenek:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string s, k;
            int beg = 0, end = 0;

            System.Console.Write("Bir yazı giriniz:");
            s = System.Console.ReadLine();
            if ((beg = s.IndexOf('{')) == -1)
                System.Console.WriteLine("yazının formatı bozuk!");
            else if ((end = s.IndexOf('}')) == -1)
                System.Console.WriteLine("yazının formatı bozuk!");
            else if (beg > end)
                System.Console.WriteLine("yazının formatı bozuk!");
            else
            {
                k = s.Substring(beg + 1, end - beg - 1);
                System.Console.WriteLine(k);
            }
        }
    }
}
```

Diğer bir seçenek:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string s, k;
            int beg = 0, end = 0;

            System.Console.Write("Bir yazı giriniz:");
            s = System.Console.ReadLine();
            beg = s.IndexOf('{');
            end = s.IndexOf('}');

            if (beg == -1 || end == -1 || beg > end)
                System.Console.WriteLine("yazının format bozuk!");
            else {
                k = s.Substring(beg + 1, end - beg - 1);
                System.Console.WriteLine(k);
            }
        }
    }
}
```

string sınıfının LastIndexOf metotları tamamen IndexOf metotları gibi kullanılır. Ancak bu metotlar ilk bulunan yerin indeksiyle değil son bulunan yerin indeksiyle geri dönerler.

Sınıf Çalışması: Klavyeden mutlak bir bir yol ifadesi (absolute path) isteyiniz. Yol ifadesindeki dosya ismini (uzantı dahil değil yazdırınız. Örneğin “c:\windows\temp\a.dat” gibi bir girişte a'nın yazdırılması gerekir. Yol ifadesindeki dosyanın uzantısı olmak zorunda değildir.

Çözüm:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string path, fileName;
            int index;

            System.Console.Write("Bir yol ifadesi (path) giriniz:");
            path = System.Console.ReadLine();
            if ((index = path.LastIndexOf('\\')) == -1)
            {
                System.Console.WriteLine("Girilen yol ifadesi geçerli değil!");
                return;
            }
            fileName = path.Substring(index + 1);
            if ((index = fileName.IndexOf('.')) != -1)
                fileName = fileName.Substring(0, index);

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

string sınıfının Remove isimli metotları yazının belli bir kısmını atmak (silmek) için kullanılır. Tabi bu metotlar asıl yazı üzerinde değişiklik yapmazlar. Silinmiş yeni bir yazıyı bize verirler.

```
public string Remove(int startIndex)
public string Remove(int startIndex, int count)
```

Metotların birinci parametreleri silme işleminin başlatılacağı indeksi, ikinci parametreleri silinecek karakter sayısını belirtir. Metotlar silinmiş yazıyla geri dönerler. Örneğin:

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

            s = text.Remove(2);
            System.Console.WriteLine(s);        // is

            s = text.Remove(2, 3);
            System.Console.WriteLine(s);        // isbul
        }
    }
}
```

Sınıf Çalışması: IndexOf metotlarının iki parametreli versaiyonunu kullanarak bir yazı içerisindeki belli bir karakterden kaç tane olduğunu bulunuz. Örneğin:

Bir ayzı giriniz: ankara bugün soğuk bir havaya sahip
aranacak karakter: a
7

Çözüm:

```
namespace CSD
{
    class App
    {
```

```
        public static void Main()
        {
            string text;
            int index, count;
            char ch;

            System.Console.Write("Bir yazı giriniz:");
            text = System.Console.ReadLine();
            System.Console.Write("Bir karakter giriniz:");
            ch = char.Parse(System.Console.ReadLine());

            count = index = 0;
            while ((index = text.IndexOf(ch, index)) != -1)
            {
                ++count;
                ++index;
            }
            System.Console.WriteLine(count);
        }
    }
}
```

Alternatif bir çözüm şöyle olabilir:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string text;
            int index, count;
            char ch;

            System.Console.Write("Bir yazı giriniz:");
            text = System.Console.ReadLine();
            System.Console.Write("Bir karakter giriniz:");
            ch = char.Parse(System.Console.ReadLine());

            for (count = 0, index = 0; (index = text.IndexOf(ch, index)) != -1; ++count, ++index)
                ;
            System.Console.WriteLine(count);
        }
    }
}
```

string sınıfının Insert isimli metodu bir yazıya başka bir yazıyı insert etmek için kullanılır. Örneğin:

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

Metodun birinci parametresi insert edilecek indeksi, ikinci parametresi insert edilecek yazıyı belirtir. Metot insert işlemi işlemi yapılmış yeni yazıyla geri döner. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string text = "isbul";
            string result;

            result = text.Insert(2, "tan");
            System.Console.WriteLine(result);       // istanbul
        }
    }
}
```

string sınıfının static olmayan Trim isimli metodu yazının başındaki ve sonundaki boşluk karakterlerini atmakta kullanılır. Metot boşluk karakterleri atılmış yeni bir yazıyı bize verir. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string str = "    bugün hava çok güzel     ";
            string result;

            result = str.Trim();
            System.Console.WriteLine(":{0}:", result);
        }
    }
}
```

string sınıfının Replace metotları tüm yazı içerisindeki belli bir karakteri başka bir karakterle ya da yazı içerisindeki belli bir yazıyı başka bir yazı ile yer değiştirmek için kullanılır.

```
public string Replace(char oldChar, char newChar)
public string Replace(string oldValue, string newValue)
```

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string str = "ankara";
            string result;

            result = str.Replace('a', 'x');
            System.Console.WriteLine(result);       // xnkxrx
        }
    }
}
```

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string str = "ali top at, ali ip atla";
            string result;

            result = str.Replace("ali", "veli");
            System.Console.WriteLine(result);       // veli top at, veli ip atla
        }
    }
}
```

string sınıfının == ve != operatör metotları vardır. Bu metotlar sayesinde biz iki string referansını == ve != operatörleriyle karşılaştırma işlemine sokabiliriz. Bu durumda karşılaştırılan şey iki referansın içerisindeki adresler değildir. Bu referansların gösterdiği yerdeki string nesnelerinin içerisindeki yazılardır. Bunun dışında string'ler >, <, >= ve <= operatörleriyle karşılaştırma işlemine sokulamazlar. Örneğin:

```
namespace CSD
{
```

```
    class App
    {
        public static void Main()
        {
            string passwd = "maviay";
            string str;

            System.Console.Write("Enter password:");
            str = System.Console.ReadLine();

            if (passwd == str)
                System.Console.WriteLine("Ok");
            else
                System.Console.WriteLine("Invalid password");
        }
    }
}
```

Sınıf çalışması: Yukarıdaki password sorma işleminde kullanıcıya üç hak verilsin. Yani password yanlış girilmişse yeniden password sorulsun. En fazla 3 kere sorulacaktır.

Çözüm:

```
class App
{
    public static void Main()
    {
        string passwd = "maviay";
        string str;

        for (int i = 0; i < 3; ++i)
        {
            System.Console.Write("Enter password:");
            str = System.Console.ReadLine();

            if (passwd == str)
            {
                System.Console.WriteLine("Ok");
                break;
            }
            else
                System.Console.WriteLine("Invalid password");
        }
    }
}
```

C#'ta switch parantezinin içerisindeki ifade string türünden olabilir ve case ifadeleri de iki tırnaklı string içerebilir. Örneğin:

```
class App
{
    public static void Main()
    {
        string name;

        System.Console.WriteLine("Bir isim giriniz:");
        name = System.Console.ReadLine();

        switch (name)
        {
            case "Ali":
                System.Console.WriteLine("Ali girildi");
                break;
            case "Veli":
                System.Console.WriteLine("Veli girildi");
                break;
            case "Selami":
                System.Console.WriteLine("Selami girildi");
```

```
                break;
            case "Ayşe":
                System.Console.WriteLine("Ayşe girildi");
                break;
            case "Fatma":
                System.Console.WriteLine("Fatma girildi");
                break;
            default:
                System.Console.WriteLine("Diğer bir isim girildi");
                break;
        }
    }
}
```

Burada bir noktaya dikkatini çekmek istiyoruz: case ifadelerinin yanında string türünden herhangi bir ifade bulunamaz, yalnızca iki tırnak içerisinde yazılardan oluşan ifadeler bulunabilir.

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            CommandPrompt cp = new CommandPrompt("CSD");
            cp.Run();
        }
    }

    class CommandPrompt
    {
        public string prompt;

        public CommandPrompt(string pr)
        {
            prompt = pr;
        }

        public void Run()
        {
            string cmd;

            for (; ; )
            {
                System.Console.Write("{0}>", prompt);
                cmd = System.Console.ReadLine();
                cmd = cmd.Trim();
                if (cmd == "")
                    continue;
                switch (cmd)
                {
                    case "dir":
                        System.Console.WriteLine("dir command");
                        break;
                    case "del":
                        System.Console.WriteLine("del command");
                        break;
                    case "clear":
                        System.Console.Clear();
                        break;
                    case "quit":
                        goto EXIT;
                    default:
                        System.Console.WriteLine("invalid command: '{0}'", cmd);
                        break;
                }
            }
        EXIT:
```

```
            ;
        }
    }
}
```

Sıfır karakterden oluşan bir string de söz konusu olabilir. Böyle string'ler "" biçiminde belirtilir.

İki string referansı + operatörüyle toplama işlemine sokulabilir. (Ancak çıkartma, çarpma ve bölme gibi başka işlemlere sokulamaz.) Bu durumda yeni bir string nesnesi yaratılır. Bu yeni string nesnesi iki string nesnesinin içerisindeki yazıların birleşiminden oluşur. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string x = "ankara", y = "izmir", z;

            z = x + y;
            System.Console.WriteLine(z);
        }
    }
}
```

Daha fazla string referansı toplama işlemine sokulabilir. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string x = "ankara", y = "izmir", z;

            z = x + " " + y;
            System.Console.WriteLine(z);
        }
    }
}
```

Burada önce “ankara” yazısına boşluk eklenmiştir sonra bu elde edilen yeni yazıya “izmir” eklenmiştir.

Örneğin:

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

            str = "Ali" + "Veli" + "Selami";
            System.Console.WriteLine(str);
        }
    }
}
```

a = a + b işleminin kısa yazımı a += b olduğunu anımsayınız. += operatörünü string'lerle de kullanabiliriz. Örneğin:

Sınıf Çalışması: Bir döngü içerisinde klavyeden Console sınıfının ReadLine metoduyla bir isim isteyiniz. Sonra girilen isimleri bir string’te birleştiriniz. “exit” yazıldığında döngüden çıkıp birleştirilmiş stringi yazdırınız.

Çözüm:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string name, names = "";

           for (;;)
            {
                System.Console.Write("Bir isim giriniz:");
                name = System.Console.ReadLine();
                if (name == "exit")
                    break;
                names += name;
            }
            System.Console.WriteLine(names);
        }
    }
}
```

Sınıf Çalışması: Yukarıdaki programı öyle bir hale getiriniz ki isimler arasında “, “ karakterleri olsun. Örneğin:

```
Ali, Veli, Selami
```

Çözüm:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string name, names = "";

           for (;;)
            {
                System.Console.Write("Bir isim giriniz:");
                name = System.Console.ReadLine();
                if (name == "exit")
                    break;
                if (names != "")
                    names += ", ";
                names += name;
            }
            System.Console.WriteLine(names);
        }
    }
}
```

İki tırnak içerisindeki yazılar iki ayrı satıra yayılamaz. Tek satırda bulunmak zorundadır. Örneğin:

```
str = "Ali, Veli,
      Selami, Ayşe";        // error!
```

Bu tür durumlarda + operatörü ile yazıları satırlara bölebiliriz. Örneğin:

```
str = "Ali, Veli, " +
      "Selami, Ayşe";       // geçerli
```

İki tırnak ifadeleri string referansı belirttiğine göre doğrudan işleme sokulabilir

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

            n = "ankara".Length;

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

Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            string s = "  ankara   ".ToUpper().Trim();
            System.Console.WriteLine(s);
         }
    }
}
```

Tam String'ler (Verbatim Strings)

İki tırnak ifadelerinin başına onunla bitişik olarak '@' karakteri getirilirse böyle iki tırnak ifadelerine "tam stringler (verbatim strings) denilmektedir. Bunlar yine string türündendir (tam string diye bir tür yoktur). Tam string'lerin normal string'lerden iki farkı vardır:

1) Bunlar içerisindeki ters bölü karakterleri ters bölü karakter sabiti anlamına gelmez. Gerçekten ters bölü karakteri anlamına gelir.

2) Tam string'ler birden fazla satıra bölünebilirler.

Örneğin:

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

            path = "c:\temp\a.dat";
            System.Console.WriteLine(path);

            path = @"c:\temp\a.dat";
            System.Console.WriteLine(path);

            path = @"Bugün hava
     çok güzel";
            System.Console.WriteLine(path);
        }
    }
}
```

İsim Alanları (Namerspaces)

İsim alanları isim çakışmasını en aza indirmek için dile sokulmuştur. .NET bileşen tabanlı (component based) bir ortamdır. Başkaları tarafından yazılmış olan sınıflar kolaylıkla başka birileri tarafından bu ortamda kullanılabilmektedir. Bazı projelerde onlarca farklı kaynaktan alınmış kodlar ve sınıflardan programcılar faydalanabilirler. Örneğin biz bir projede A ve B şirketlerinin ya da kurumlarının iki sınıf kütüphanesini birarada kulanacak olalım. Bu iki şirketin birbirleriyle bir ilgileri olmadığı için bunlar sınıflarına tesadüfen aynı isimleri vermiş olabilirler. Örneğin her iki şirket de bir sınıfına Timer ismini vermiş olabilir. Bu durumda isim çakışması oluşur. İşte eğer isim alanları olmasaydı bu çakışma kdoun derlenmesi sırasında soruna yol açardı. Halbuki isim alanları sayesinde her şirket sınıflarını kendi isimlerine ilişkin isim alanlarında oluşturacağı için bu çalışma olasılığı azaltılmaktadır. Örneğin:

```
namespace A
{
	class Timer
	{
		//...
	}
}

namespace B
{
	class Timer
	{
		//...
	}
}
//...
A.Timer aTimer = new A.Timer();
B.Timer bTimer = new B.Timer();
```

C#'ta aynı isim alanı içerisinde aynı isimli birden fazla sınıf bildirilemez. Fakat farklı isim alanları içerisinde aynı isimli sınıflar bildirilebilir. Her şirketin ya da kurumun sınıflarını kendisine ilişkin bir isim alanında bildirmesi tavsiye edilmektedir.

İsim alanı bildiriminin genel biçimi şöyledir:

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

Bir isim alanı içerisindeki isimler isim alanı ismiyle niteliklendirilerek kullanılırlar. Örneğin A.Timer, System.Random gibi...

Bir isim alanının ikinci kez bildirilmesi error oluşturmaz. Bu durum birleştirme anlamına gelir. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            A.Sample s = new A.Sample();
            A.Test k = new A.Test();
            //...
        }
    }
}

namespace A
{
    class Sample
    {
```

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

Burada Sample ve Test A isim alanın içerisindedir.

İç içe isim alanları bildirilebilir. İç isim alanındaki isimler dış isim alanları ile niteliklendirilerek ifade edilirler. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            A.B.Test.Foo();
            A.Sample.Bar();
        }
    }
}

namespace A
{
    namespace B
    {
        class Test
        {
            public static void Foo()
            {
                System.Console.WriteLine("A.B.Test.Foo");
            }
        }
    }
    class Sample
    {
        public static void Bar()
        {
            System.Console.WriteLine("A.Sample.Bar");
        }
    }
}
```

Aynı isim alanı içerisindeki aynı isimli isim alanları birleştirilmektedir. Farklı isim alanlarının içerisindeki aynı isimli isim alanları birleştirilmez. Örneğin aşağıdaki kod parçasında B isim alanları birleştirilmeyecektir:

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

namespace A
{
    namespace B
    {
        class X
```

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

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

İç bir isim alanını tek hamlede noktalı biçimde bildirebiliriz. Örneğin:

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

Burada Sample sınıfı A isim alanı içerisinde değildir. A isim alanı içerisindeki B isim alanı içerisindedir. Yani yukarıdaki işlemin eşdeğeri şöyledir:

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

Örneğin aşağıdaki iki isim alanı birleştirilir:

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

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

Hiçbir isim alanının içerisinde bulunmayan en dış bölge de bir isim alanı belirtir. Buna "global isim alanı (global namespace)" denilmektedir. Global isim alanında da sınıflar bildirilebilir. Böylece aslında bizim isim alanlarımız global isim alanı içerisindedir. Yani örneğin “Merhaba Dünya” programı şöyle de yazılabilirdi:

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

```
    {
        System.Console.WriteLine("Merhaba Dünya");
    }
}
```

System isim alanı .NET'in sınıf kütüphanesine ayrılmıştır. .NET'in sınıfları ya System isim alanının içerisindedir ya da System isim alanının içerisindeki bir isim alanının içerisindedir.

Dinamik Kütüphanelerin Oluşturulması (DLL'lerin Oluşturulması)

İçerisinde derlenmiş biçimde kodların bulunduğu özel dosyalara kütüphane (library) denir. Kütüphaneler statik ve dinamik olmak üzere ikiye ayrılırlar. Windows'ta statik kütüphane dosyalarının uzantısı .LIB, dinamik kütüphane dosyalarının uzantıları ise .DLL (Daynamic Link Library) biçimindedir. .NET'te statik kütüphane kullanımı yoktur. Yalnızca dinamik kütüphaneler kullanılabilmektedir.

Aslında teknik olarak .EXE dosyaları ile .DLL dosyaları arasında önemli bir farklılık yoktur. Her iki dosya da "PE (Portable Executable)" dosya formatına sahiptir. Bunların aralarındaki tek fark .EXE dosyanın çalıştırılabilmesi, .DLL dosyasının çalıştırılamamasıdır. .EXE dosyada bir Main metodu (entry point) vardır. .DLL dosyasında yoktur. (Tabi biz bir DLL dosyasına Main metodu eklersek sorun çıkmaz. Fakat bu Main metodu sıradan bir metot olarak değerlendirilir.)

Komut satırında DLL oluşturmak için tek yapılacak şey csc.exe derleyicisinde /target:library seçeneğini kullanmaktır (kısaca /t:library biçiminde de bu seçenek belirtilebilir.) Örneğin:

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

Özetle komut satırından bir DLL oluşturmak için şunlar yapılır:

1) Kaynak dosya bir editör kullanılarak oluşturulur ve save edilir. (Örneğin ismi Sample.cs olsun)

2) Komut satırından /target:library seçeneği ile derleme yapılır:

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

Artık ürün olarak .exe dosya değil, .dll dosyası oluşacaktır.

Aslında /target seçeneğinde aşağıdakilerin herhangi biri kullanılabilmektedir:

```
/target:exe
/target:winexe
/target:library
/target:module
```

/target:exe seçeneği console uygulaması oluşturmak için kullanılır. Bu default durumdur. Yani /target seçeneği hiç belirtilmemişse sanki /target:exe yapılmış gibi işlem görür. /target:winexe GUI uygulamaları oluşturmak için kullanılmaktadır. GUI programlar bizim Windows sistemlerinde gördüğümüz klasik pencereli programlardır. /target:library DLL dosyası oluşturmak için kullanılır. /target:module ise .NET modül dosyası oluşturmak için kullanılmaktadır. Biz modül dosyalarını bu kursta ele almayacağız.

Dinamik kütüphaneler Visual Studio IDE'si kullanılarak da kolay bir biçimde oluşturulabilirler. Bunun için "Empty Project" seçeneği ile boş bir proje oluşturulur. İçerisine kaynak dosya eklenir. Sonra proje seçeneklerine gelinerek (project properties) burada "Output Type" "Class Library" olarak seçilir. Artık build işlemi yapıldığında .exe yerine .dll dosyasıoluşturulacaktır. Tabii böyle bir projeyi Ctrl+F5 tuşlarıyla çalıştırmaya çalışmak anlamsızdır. Çünkü DLL'ler çalışmazlar. Ancak başka uygulamalardan kullanılabilirler. Bu nedenle böyle bir proje "build" edilerek bırakılmalıdır. (Aslında Visual Studio "Output Type" seçeneğini "Class Library" olarak değiştirdiğimizde csc.exe derleyicisini "/target:library" seçeneğiyle çalıştırmaktadır.)

Ayrıca Visual Studio IDE'sinde File/New/Project seçilerek açılan “New Project” diyalog penceresindeki “Class Library” seçeneği bize içi boş bir sınıf eşliğinde DLL projesi oluşturur. Visual Studio bizim için projeye içi boş bir sınıf da ekler.

DLL'lerin Kullanılması

.NET'te kabaca .dll ve .exe dosyalarına assembly denilmektedir. (Aslında assembly kavramının bazı detayları vardır. Fakat bu kursta ele alınmayacaktır.) Assembly terimini “sembolik Makine Dili” derken kullanılan “Assembly Language”'teki assembly ile karıştırmayınız.

Bir DLL'in içerisindeki sınıfları kullanabilmek için o dll'e referans etmek gerekir. Referans etme işlemi komut satırında /reference:<dll yol ifadesi> ya da kısaca /r:<dll yol ifadesi> seçeneğiyle yapılır. Örneğin:

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

Burada Sample.cs dosyası derlenerek .exe dosyası elde edilmek istenmiştir. Ancak Sample.cs içerisinde Test.dll isimli dll’in içerisindeki sınıflar kullanılmakistenmiştir.

Birden fazla dll'e referans edilirken yeniden /reference ya da /r seçeneğini belirtmek gerekir. Örneğin:

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

Referans edilecek dll başka bir dizindeyse referans etme sırasında yol ifadesi belirtilmelidir. Örneğin:

```
csc /r:C:\Temp\TestDll.dll Sample.cs
```

Ancak dll kullanan bir exe çalıştırılırken exe ile dll dosyalarının aynı dizinde bulunması gerekmektedir.

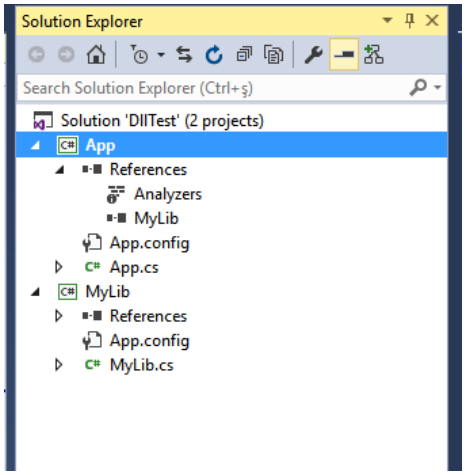
Ayrıca dll içerisindeki sınıfları dışarıdan (yani başka bir exe ya da dll içerisinden) kullanabilmek için sınıf bildirimlerinin başına public erişim belirleyicisinin getirilmesi gerekir. Örneğin:

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

İsim alanları içerisindeki sınıf bildirimlerinin önüne erişim belirleyici olarak yalnızca public ve internal anahtar sözcükleri getirilebilir. public anahtar sözcüğü ilgili sınıfın referans edilmek koşuluyla dışarıdan başka bir assembly'den kullanılabileceğini belirtir. internal sınıflar DLL içerisindeki başka sınıflar tarafından kullanılabilirler fakat dışarıdan başka bir assembly'den kullanılamazlar. Hiçbir erişim belirleyicisinin kullanılmaması ile internal kullanılması aynı anlamdadır.

Visual Studio IDE'sinde DLL'e referans etmek için "Solution Explorer"da "References" kısmına gelinerek "Add Reference" yapılır. Açılan diyalog penceresinde "Browse" tabı seçilir. Ve ilgili DLL bulunarak eklenir. Artık program çalıştırıldığında IDE derleme için csc'yi kullanırken /reference seçeneğini de ekleyecektir. IDE bir dll'e referans edildiğinde o dll'i aynı zamanda exe'nin bulunduğu dizine de kopyalamaktadır.

Visual Studio IDE'sinde bir solution içerisine istersek biz birden fazla proje ekleyebiliriz. Örneğin projelerden biri dll projesi olabilir, diğeri exe projesi olabilir. Exe projesinde dll projesindeki dll'e referans edilebilir.



Burada bir solution içerisinde iki proje bulunmaktadır. App projesi “exe”, MyLib projesi ise “dll” projesidir. App projesi MyLib projesinden elde edilen dll’e referans etmektedir.

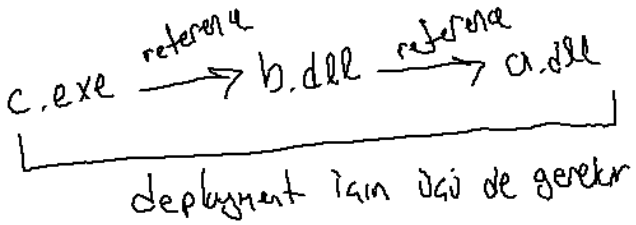
Bir dll de başka bir dll'i kullanıyor olabilir. Örneğin:

```
csc /target:library a.cs
csc /target:library /reference:a.dll b.cs
```

Burada önce a.dll dosyası oluşturulmuştur. Sonra b.dll dosyası oluşturulurken a.dll'e referans edilmiştir. Şimdi b.dll içerisindeki sınıfları kullanan bir c.exe oluşturmak isteyelim:

```
csc /target:exe /reference: b.dll c.cs
```

Görüldüğü gibi bu derlemede a.dll'e referans edilmemiştir. Hangi dll'ler doğrudan kullanılıyorsa yalnızca onlara referans edilmesi yeterlidir. Tabi program çalışırken hem a.dll, hem b.dll hem de c.exe'nin aynı dizinde bulunuyor olması gerekir.



Özel (Private) ve Paylaşımlı (Shared) Dll'ler

.NET dünyasında DLL'ler özel ve paylaşımlı olmak üzere ikiye ayrılmaktadır. Şimdiye kadar bizim oluşturduğumuz ve kullandığımız DLL'ler özel DLL'lerdi. Özel DLL'ler program çalıştırılırken .exe dosya ile aynı dizin içerisinde bulunmak zorundadır. Özel DLL'ler daha esnek bir kullanıma olanak sağlarlar. Ancak bunların bir dezavantajı da vardır. Aynı DLL'li kullanan birden fazla program aynı makineye taşınmak istendiğinde (setup yapılmak istendiğinde) bu DLL'in kopyaları o dizinlere yeniden çekilmek zorunda kalır. Halbuki paylaşılan DLL'lerde yalnızca bir kopya özel bir yere (buraya "GAC (Global Assembly Cache)" denilmektedir) çekilir. Uygulamaların hepsi GAC’daki aynı DLL'i kullanır. Böylece biz onları tekrar tekrar aynı makinaya kopyalamak zorunda kalmayız.

Paylaşılan DLL'lerin oluşturuşması ve kullanılması "Uygulama Geliştirme-II" kursunda ele alınmaktadır.

.NET'in Kendi Sınıfları ve DLL'ler

.NET'in kendi sınıfları da birtakım DLL'lere yerleştirilmiştir. Bunlar paylaşılan DLL'lerdir ve GAC (Global Assembly Cache) denilen özel bir yerde bulunmaktadır. (MSDN yardım sisteminde hangi sınıfların hangi DLL'lerde olduğu belirtiliyor.) İter özel olsun isterse paylaşımlı olsun bir dll’i kullanan program o dill’e

referans etmek zorundadır. Aynı durum .NET'in sınıflarının bulunduğu dll'ler için de söz konusudur. Bu bakımdan .NET'in DLL'lerinin diğer DLL'lerden hiçbir farkı yoktur. Ancak .NET'in en çok kullanılan sınıfları "mscorlib.dll" isimli bir DLL'e yerleştirilmiştir. csc.exe derleyicisi de bu DLL'e otomatik referans edecek biçimde yazılmıştır. Yani bir sınıf ya da tür eğer mscorlib.dll içerisindeyse ona referans etmemize gerek yoktur. Zaten kendiliğinden ona referans edilmektedir. Ancak bir sınıf ya da tür başka bir .NET DLL'inin içerisindeyse ona referans etmek gerekir. Örneğin Math sınıfı mscorlib.dll içerisindedir. Bizim Math sınıfını kullanabilmemiz için ona referans etmemiz gerekmez. Fakat System.Windows.Forms isim alanı içerisindeki MessageBox sınıfı System.Windows.Forms.dll içerisindedir. Onu kullanabilmek için bu DLL'e referans etmemiz gerekir.

Visual Studio IDE'sinde "Add Reference" dialog pencersindeki "Framework" sekmesi .NET'in kendi DLL'lerine ayrılmıştır. Pekiyi NET'in bu DLL'lerini GAC'a kim yerleştirmiştir? İşte .NET ortamı (.NET framework) kurulurken bu DLL'ler kurulum programı tarafından de GAC'a çekilmektedir.

İsim alanlarıyla DLL'leri karıştırmamak gerekir. İsim alanları mantıksal bir organizasyondur fakak DLL'ler fiziksel dosyalardır. Biz bir sınıfı kullanacaksak onun hangi DLL içerisinde olduğunu ve hangi isim alanı içerisinde olduğunu bilmemiz gerekir. Bir DLL'in içerisinde çeokm farklı isim alanları bulunabilir.

Birden Fazla Kaynak Dosya İle Derleme İşlemleri

Büyük projelerin tek bir kaynak dosya ile organize edilmesi iyi bir teknik değildir. Çünkü durumda kodların yönetilmesi ve edit edilmesi zor olur. Aslında bir proje birden fazla kaynak dosyayla derlenebilir. Birden fazla kaynak dosyadan oluşan bir programı komut satırından derlerken tüm bu kaynak dosyaların isimleri belirtilmelidir. Örneğin:

```
csc a.cs b.cs c.scs
```

Burada program üç kaynak dosyadan oluşmaktadır: a.cs, b.cs ve c.cs. Bu durumda exe derlemesi yapıldığı için çalıştırılabilen dosyanın ismi (yani exe'nin ismi) Main metodu hangi kaynak dosyadaysa o dosyanın ismiyle aynı olacaktır. (Örneğin Main metodu b.cs içerisindeyse biz bu işlemden b.exe elde ederiz.) Eğer dll derlemesi yapılmış olsaydı .DLL dosyasının ismi ilk kaynak dosyanın ismi olacaktı. Örneğin:

```
csc /target: library a.cs b.cs c.cs
```

Bu derleme işleminden ürün olarak a.dll elde dosyası edilecektir.

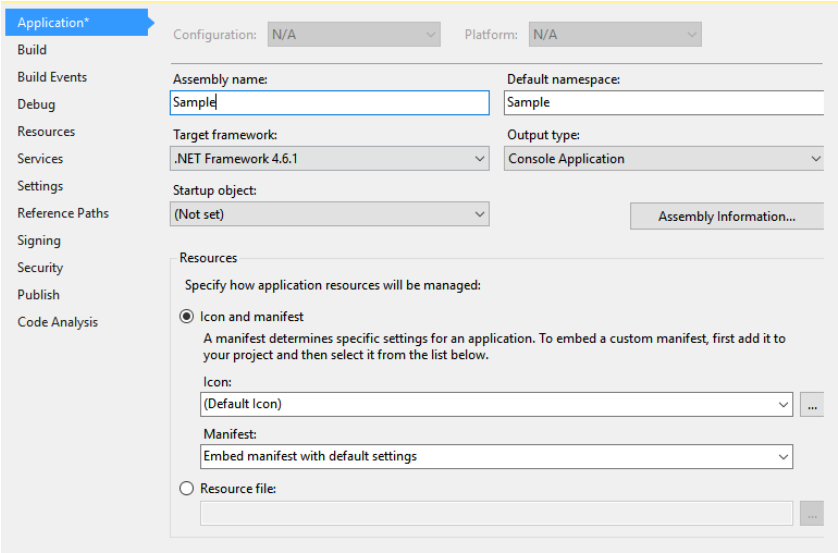
Fakat biz /out:<isim> seçeneği ile çıktı dosyasına istediğimiz ismi verebiliriz. Örneğin:

```
csc /out:project.exe a.cs b.cs c.scs
```

Burada artık Main hangi dosyada olursa olsun çalıştırılabilen dosyanın ismi "project.exe" olacaktır.

IDE'de birden fazla kaynak dosyayla derleme işlemi için tek yapacağımız şey projeye birden fazla kaynak dosyayı yerleştirmektir. Uygulamada büyük projeler onlarca kaynak dosyanın toplamından oluşmaktadır.

Visual Studio IDE'si ile çalışırken projeye birden fazla kaynak dosya eklediğimizde elde edilen "exe" ya da "dll" dosyasının ismi proje özelliklerinden ayarlanabilir. Default durumda bu isim proje ismiyle aynıdır.



İsim Araması (Name Lookup)

Her kullanılan ismin bir bildirimi bulunmak zorundadır. Derleyici bir isimle karşılaştığında o isme ilişkin bildirimi bulmak ister. Eğer derleyici isme ilişkin bir bildirim bulamazsa error oluşur. İsim araması niteliksiz (unqualified) ve nitelikli (qualified) olmak üzere ikiye ayrılmaktadır. İsim araması sırasında derleyici sırasıyla bazı faaliyet alanlarına bakar. Eğer ismi bu faaliyet alanlarında bulursa artık aramaya devam etmez. Eğer sonuna kadar devam edip ismi bulmazsa error oluşur.

Nokta operatörü olmadan doğrudan kullanılan isimler (a, b, System gibi) niteliksiz isim arama kurallarına göre aranır. Ancak nokta operatörünün sağındaki isimler nitelikli arama kuralına göre aranırlar. Eğer birden fazla nokta operatörü kombine edildiyse en soldaki isim niteliksiz arama kurallarına göre onun sağındakiler nitelikli arama kurallarına göre aranmaktadır. Örneğin A.B.C ifadesinde A ismi niteliksiz olarak, B ve C isimleri nitelikli olarak aranacaktır.

Bildirilen isim aranmaz, kullanılan isim aranır. Örneğin:

```
Sample s;
```

Burada s ismi aranmayacaktır. Çünkü bu isim zaten bildirilmektedir. Fakat Sample ismi kullanılmıştır, dolayısıyla aranacaktır.

İsim aramasına derleme işlemine katılan tüm kaynak dosyalar ve referans edilen tüm DLL'ler dahil edilmektedir. Örneğin biz bir DLL'e referans etmişsek sanki o DLL'in kodunu projeye dahil etmişiz gibi isim aramasına o DLL de dahil edilmektedir.

Niteliksiz İsim Arama Kuralları

Aşağıdaki isim arama maddeleri else-if biçiminde değerlendirilmelidir. İsim bulunursa sonraki maddeye geçilmez. Niteliksiz isim arama sırasında şu faaliyet alanlarına sırasıyla baklmaktadır.

1) İsim bir metot içerisinde kullanılmışsa kullanım yerinden yukarıya doğru isim metodun yerel bloklarında aranır.

2) İsim içinde bulunduğu sınıf bildiriminin her yerinde aranır. (Örneğin sınıfın bir veri elemanıyla aynı isimli bir yerel değişken olabilir. Metot içerisinde biz bu ismi kullandığımızda yerel değişkene erişmiş oluruz.)

3) İsim kullanıldığı sınıfın içinde bulunduğu isim alanının her yerinde aranır.

4) İsim kullanıldığı sınıfın içinde bulunduğu isim alanını kapsayan isim alanlarında içeriden dışarıya doğru sırasıyla aranır

5) İsim global isim alanında aranır.

Nitelikli İsim Arama Kuralları

Nokta operatörünün solunda üç şey bulunabilir: Bir sınıf ismi, isim alanı ismi ve referans ismi.

1) Nokta operatörünün solunda bir sınıf ismi varsa, sağındaki isim yalnızca o sınıf bildiriminin her yerinde aranır. Başka bir yere bakılmaz. (Örneğin Sample.Foo() gibi bir ifadede Sample niteliksiz olarak aranır. Bulunursa bu sefer Foo nitelikli olarak aranır. Foo yalnızca Sample'da aranacaktır.)

2) Nokta operatörünün solunda bir isim alanı ismi varsa sağındaki isim yalnızca o isim alanın her yerinde aranır (taban sınıflara da bakılır). Kapsayan isim alanlarına bakılmaz.

3) Nokta operatörünün solunda bir referans varsa sağındaki isim o referansın ilişkin olduğu sınıf bildiriminin her yerinde aranır (taban sınıflara da bakılır). Başka bir yerde aranmaz.

Örneğin:

```
A.B.C.Foo();
```

ifadesinde A niteliksiz olarak aranır. Bulunursa B A'nın içerisinde, C B'nin içerisinde, Foo da C'nin içerisinde aranacaktır.

using Direktifi

using direktifi niteliktendirmeyi azaltarak programcıya kolaylık sağlamak için düşünülmüştür. Direktifin genel biçimi şöyledir:

```
using <isim alanı ismi>;
```

Örneğin:

```
using System;
using A.B;
```

İsim alanı ismi noktalı biçimde de olabilir.

using direktifleri isim alanlarının başına yerleştirilmek zorudadır. Yani using direktifleri isim alanlarının ilk elemanları olmak zorundadır. using direktifleri global isim alanına da yerleştirilebilir. Global isim alanın başı aynı zamanda kaynak kodun da başıdır. Bu durumda eğer using direktifleri global isim alanına yerleştirilecekse kaynak kodun tepesine yerleştirilmek zorundadır.

using direktifinde iki isim alanı önemlidir: Direktifin yerleştirildiği isim alanı ve direktifte belirtilen isim alanı. using direktifi şöyle etki gösterir: Niteliksiz isim arama sırasında isim using direktifinin yerleştirildiği isim alanına kadar ve using direktifinin yerleştirildiği isim alanında bulunamazsa, direktifte belrtilen isim alanına da bakılır.

Örneğin:

```
namespace CSD
{
    using A;

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

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

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

Örneğin:

```
namespace CSD
{
    using System;

    class App
    {
        public static void Main()
        {
            Console.WriteLine("Test");  // geçerli
        }
    }
}
```

Eğer isim using direktifinin yerleştirildiği isim alanına kadar ya da o isim alanında bulunursa using direktifi ile belirtilen isim alanına bakılmaz. Örneğin:

```
namespace CSD
{
    using A;

    class App
    {
        public static void Main()
        {
            Sample s;       // Sample CSD içerisindeki Sample
            //...
        }
    }

    class Sample
    {
        //...
    }
}

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

using direktifleri herhangi bir isim alanına yerleştirilebilir. Ancak yaptığı etki aynı olmayabilir. Örneğin:

```
using A;

namespace CSD
{
    // using A;

    class App
    {
```

```
        public static void Main()
        {
            Sample s;     // Global Sample
            //...
        }
    }
}

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

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

Burada using direktifinin CSD isim alanına yerleştirilmesiyle global isim alanına yerleştirilmesi farklı etkiye yol açmaktadır.

İsim using direktifinin yerleştirildiği isim alanına kadar ve o isim alanında bulunamamışsa fakat birden fazla using direktifi ile belirtilen isim alanında bulunursa error oluşur. Aynı isim alanındaki using direktiflerinin sırasının bir önemi yoktur. Örneğin:

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

    class App
    {
        public static void Main()
        {
            Sample s;     // error!
            //...
        }
    }
}

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

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

C#'ta using direktiflerinde geçişme özelliği yoktur (halbuki örneğin C++'ta vardır). Yani using direktifi ile belirtilen isim alanında arama yapılırken o isim alanındaki using direktifleri artık etki göstermez. Örneğin:

```
namespace CSD
{
    using A;

    class App
```

```
    {
        public static void Main()
        {
            Sample s;     // error!
            //...
        }
    }
}

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

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

using direktifi nitelikli aramalarda etki göstermez. Yani örneğin A.Sample ifadesinde Sample A'da nitelikli aranır. A'da bulunamazsa A'daki using direktifleri etki göstermez:

```
namespace CSD
{
    using A;

    class App
    {
        public static void Main()
        {
            A.Sample s;     // error!
            //...
        }
    }
}

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

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

using direktifi ile belirtilen isim alanında arama yapılırken buradaki isim alanı isimleri görülmez. Örneğin:

```
namespace CSD
{
    using A;

    class App
    {
        public static void Main()
        {
            B.Sample s;     // error!
            //...
        }
    }
```

```
}

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

Fakat bu durumda iç isim alanına using uygulayabiliriz:

```
namespace CSD
{
    using A.B;

    class App
    {
        public static void Main()
        {
            Sample s;     // geçerli!
            //...
        }
    }
}

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

Ya da niteliklendirme yapabiliriz:

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

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

Bundan sonra kursumuzda niteliklendirmeyi en aza indirerek using direktiflerindne faydalanacağız. Madem ki .NET'in en önemli sınıfları doğrudan System isim alanı içerisindedir. O halde System isim alanına using uygulayabiliriz. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Console.WriteLine("Merhaba using direktifi!");
        }
    }
}
```

using Alias Direktifi

using alias direktifi de isim alanlarının başına yerleştirilmek zorundadır. using direktifileriyle karışık sırada yerleştirilebilir. Genel biçimi şöyledir:

```
using <isim> = <isim alanı ismi ya da sınıf ismi ya da tür ismi>;
```

using alias direktifi bir isim alanı ismine ya da sınıf ismine alternatif bir isim vermek için kullanılır. Böylece '=' atomunun solundaki isim tam olarak sağındaki ismin yerini tutar. Örneğin:

```
using System;

namespace CSD
{
    using Con = System.Console;

    class App
    {
        public static void Main()
        {
            Con.WriteLine("Test");
        }
    }
}
```

Burada CSD isim alanı içerisinde Con denildiğinde System.Console anlaşılır.

Bu direktif genellikle uzun nitelikli bir ismi kısaltmak için kullanılır. Örneğin:

```
using System;

namespace CSD
{
    using MB = System.Windows.Forms.MessageBox;

    class App
    {
        public static void Main()
        {
            MB.Show("Ok");
        }
    }
}
```

Fakat bu direktif bazen isim çakışmasında onları ayrıştırmak için de kullanılmaktadır. Örneğin:

```
using System;

namespace CSD
{
    using ASmp = A.Sample;
```

```
    using BSmp = B.Sample;

    class App
    {
        public static void Main()
        {
            ASmp s = new ASmp();
            BSmp k = new BSmp();
            //...
        }
    }
}

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

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

extern alias Bildirimi

extern alias bildirimi birden fazla DLL içerisinde aynı isimli isim alanları varsa buradaki isim çakışmasını engellemek amacıyla kullanılmaktadır. Örneğin birbirlerini tanımayan A ve B isimli iki şirket bulnuyor olsun. Biz projemizde onların sınıflarını kullanacak olalım. Fakat bu şirket A.DLL ve B.DLL kütüphanelerinde tesadüfen aynı isimli isim alanları kullanmış olsunlar. Ve üstelik buradaki bazı sınıf isimleri çakışıyor olsun. İşte böylesi nadir durumda extern alias bildirimi kullanılmaktadır. extern alias bildiriminin genel biçimi şöyledir:

```
extern alias <isim>;
```

Örneğin:

```
extern alias ACompany;
extern alias BCompany;
```

extern alias bildirimleri de isim alanlarının başına yerleştirilmek zorundadır. Eğer orada using direktifleri ve using alias direktifleri de varsa onlardan önce bulundurulmak zorundadır. Örneğin:

```
namespace CSD
{
      extern alias ACompany;
      extern alias BCompany;
      using Con = System.Console;
      using System.IO;
      //...
}
```

extern alias bildirimi şöyle kullanılır:

1) Öncelikle ilgili DLL'lere referans edilirken bir alias ismi verilmelidir. csc.exe komut satırında alias ismi /reference: ACompany=A.DLL biçiminde bir sentaksla verilmektedir. Visual Studio IDE'sinde aynı işlem "References" kısmında "Properties/Aliases" menüsüyle yapılır.

2) Program içerisinde referans edilen dll'lere verilen alias isimler extern alias direktifi ile bildirilir. Örneğin:

```
extern alias ACompany;
extern alias BCompany;
```

3) Program içerisinden bu alias ismi ve :: operatörü ile o DLL'in köküne erişilir. Örneğin:

```
ACompany::X.Sample s = new ACompany::X.Sample();
BCompany::X.Sample k = new BCompany::X.Sample();
```

Örneğin:

```
using System;

namespace CSD
{
    extern alias ACompany;
    extern alias BCompany;

    class App
    {
        public static void Main()
        {
            ACompany::X.Sample s = new ACompany::X.Sample();
            BCompany::X.Sample k = new BCompany::X.Sample();

            s.Foo();
            k.Bar();
        }

    }

}
```

Tazımı kısaltmak için using alias direktifi de kullanılabilirdi:

```
using System;

namespace CSD
{
    extern alias ACompany;
    extern alias BCompany;
    using AComp = ACompany::X.Sample;
    using BComp = BCompany::X.Sample;

    class App
    {
        public static void Main()
        {
            AComp s = new AComp();
            BComp k = new BComp();

            s.Foo();
            k.Bar();
        }

    }
}
```

Diziler (Arrays)

Elemanları aynı türden olan ve bellekte ardışıl bir biçimde tutulan veri yapılarına dizi denir. C#'ta her T türü için T[] biçiminde belirtilen T türünden dizi türü de vardır. T türü kategori olarak ister değer türlerine ilişkin olsun isterse referans türlerine ilişkin olsun T[] türü her zaman referans türlerine ilişkindir. Yani örneğin:

```
int a;
int[] b;
```

Burada a değerin kendisini tutar. Fakat b adres tutar.

Diziler C#'ta sınıfsal bir biçimde temsil edilmektedir. Yani bir dizi sanki bir sınıf nesnesiymiş gibi new operatörüyle yaratılır. Dizinin elemanları dizi nesnesinin içerisindedir. Dizi nesnesinin içerisinde ayrıca dizinin uzunluğu da tutulmaktadır. Dizi tahsisatının genel biçimi şöyledir:

```
new <tür><[<uzunluk ifadesi>]>;
```

Örneğin:

```
new int[10]
new string[5]
new double[2]
```

gibi.

new operatöründen ürün olarak dizi nesnesinin başlangıç adresi elde edilmektedir. Bu adres aynı türden bir dizi referansına atanabilir. Örneğin:

```
int[] a;

a = new int[5];
```

Dizi elemanlarına köşeli parantez operatörüyle erişilir. a bir dizi referansı olmak üzere a[i] ifadesi a referansının gösterdiği yerdeki dizi nesnesinin i'inci indeksteki elemanına erişmekte kullanılır. Dizinin ilk elemanı 0'ıncı indekstedir. Bu durumda n elemanlı bir dizinin n'inci elemanı n-1'inci indekste olacaktır. Dizi elemanına erişimin genel biçimi şöyledir:

```
referans<[ifade]>
```

Dizi uzunluğunu belirtirken ve dizi elemanına erişirken köşeli parantez içerisindeki ifade tamsayı türlerine ilişkin olmak zorundadır.

Örneğin:

```
using System;

namespace CSD
{

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

            a = new int[100];
```

```
            for (int i = 0; i < 100; ++i)
                a[i] = i * i;

            for (int i = 0; i < 100; ++i)
                Console.Write("{0} ", a[i]);
            Console.WriteLine();
        }
    }
}
```

Dizilere neden gereksinim duyulmaktadır? Diziler sayesinde çok sayıda değişkene bir döngü yoluyla erişebiliriz. Bu dizilerin en önemli kullanım gerekçesini oluşturmaktadır.

Örneğin:

```
using System;

namespace CSD
{

    class App
    {
        public static void Main()
        {
            int[] a;
            int n;

            Console.Write("Dizi uzunluğunu giriniz:");
            n = int.Parse(Console.ReadLine());
            a = new int[n];

            for (int i = 0; i < n; ++i)
            {
                Console.Write("{0}.indeksli elemanı giriniz:", i);
                a[i] = int.Parse(Console.ReadLine());
            }

            int total = 0;
            for (int i = 0; i < n; ++i)
                total += a[i];

            Console.WriteLine("Dizideki elemanların toplamı = {0}", total);
        }
    }
}
```

new İşlemi Sırasında Dizi Elemanlarına İlkdeğer Verilmesi

new işlemi ile dizi tahsis edilirken, tahsis edilen dizi elemanlarına hemen ilkdeğerleri verilebilir. Bunun için new operatöründe köşeli parantezlerden sonra küme parantezi içerisinde elemanlar virgüllerle ayrılarak belirtilirler. Örneğin:

```
int[] a;

a = new int[3] {10, 20, 30};
```

Eğer dizi elemanlarına bu biçimde ilkdeğer veriliyorsa uzunluk belirten ifadenin sabit ifadesi olması zounludur. Örneğin:

```
int[] a;
int n = 3;

a = new int[n] {10, 20, 30};        // error!
```

Ayrıca küme parantezlerinin içerisindeki elemanların tam olarak dizi uzunluğu kadar olması da zorunludur. Örneğin:

```
a = new int[3] = {1, 2};            // error!
```

Örneğin:

```
using System;

namespace CSD
{

    class App
    {
        public static void Main()
        {
            int[] a;
            int max;

            a = new int[10] { 10, 4, 7, 23, 34, 71, -13, 27, 45, 32 };

            max = a[0];
            for (int i = 1; i < 10; ++i)
                if (max < a[i])
                    max = a[i];

            Console.WriteLine("Max = {0}", max);
        }
    }
}
```

Aslında ilkdeğer verilirken köşeli parantezlerin içi tamamen boş da bırakılabilir. Bu durumda derleyici küme parantezlerinin içerisindeki ilkdeğerleri sayar ve dizinin o uzunlukta açılmış olduğunu kabul eder. Örneğin:

```
a = new int[] {1, 2, 3, 4};         // geçerli
b = new int[] {10, 20 };            // geçerli
```

Ancak ilkdeğer verme işlemi yapılmadan dizi tahsis edilirken köşeli parantezin içi boş bırakılamaz. Örneğin:

```
a = new int[];          // error!
```

Ancak ilkdeğer veriliyorsa boş bırakılabilir:

```
a = new int[] {10, 20, 30};         // geçerli
```

Fakat programcılar gerekmese bile dizi uzunluğunu belirtmek isteyebilirler. Bunun iki nedeni olabilir: Birincisi okunabilirliği artırmaktır. Yani koda bakan kişi dizinin eleman sayısını hemen anlayabilir. İkincisi de derleme zamanında derleyicinin yanlışlıkla eksik ya da fazla girişler için kontrol uygulamasıdır.

Küme parantezleri içerisinde verilen ilkdeğerlerin sabit ifadesi olması zorunlu değildir. Örneğin:

```
int[] a;
int x = 5;

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

Bir dizi referansına ilkdeğer verilirken hiç new operatörü kullanılmayabilir. Doğrudan ilkdeğerler küme parantezlerinin içerisinde belirtilebilir. Örneğin:

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

Aslında burada yine derleyici tarafından new işlemi yapılmaktadır. Yani bu işlem aslında aşağıdakiyle

eşdeğerdir:

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

Ancak new operatörünün ihmal edilmesi yalnızca ilkdeğer verme işlemiyle sınırlıdır. Daha sonra bu ihmal yapılamaz. Örneğin:

```
int[] a;

a = {1, 2, 3};          // error!
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int[] a = { 10, 12, 34, 23, 1, 4, 7, 9, 23, 33 };
            int odd = 0, even = 0;

            for (int i = 0; i < a.Length; ++i)
                if (a[i] % 2 == 0)
                    ++even;
                else
                    ++odd;

            Console.WriteLine("Tek = {0}, Çift = {1}", odd, even);
        }
    }
}
```

Yukarıda da belirtildiği gibi dizi uzunlukları dizi nesnelerinin içerisinde tutulmaktadır. İşte dizilerin Length isimli read-only int türden property'leri referansın gösterdiği yerdeki dizinin uzunluğunu bize verir.

Aynı türden iki dizi referansı birbirlerine atanabilir. Bu durumda bu referanslar aynı diziyi görür. Örneğin:

```
using System;

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

            b = a;
            for (int i = 0; i < b.Length; ++i)
                Console.Write("{0} ", b[i]);
            Console.WriteLine();
        }
    }
}
```

T1 türünden T2 türüne otomatik dönüştürmenin olması T1[] türünden T2[] türüne otomatik dönüştürmenin olacağı anlamına gelmez. Yani örneğin int türünde long türüne atama yapabiliriz. Fakat int[] türünden long[] türüne atama yapamayız. Yalnızca aynı dizi türleri birbirlerine atanabilir.

Bir dizi referansına biz aynı türden olmak koşuluyla herhangi uzulukta bir dizi referansını atayabiliriz. Çünkü

dizilerin uzunluğu referans ile belirli değildir. O referansların gösterdiği yerdeki nesnenin içerisinde uzunluk bilgisi bulunur. Örneğin:

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

b = a;  // Burada artık b 10 elemanlı bir diziyi gösteriyor
```

Dizilerin Metotlara Parametre Yoluyla Aktarılması

Bir metodun parametre değişkeni bir dizi türünden olabilir. Bu durumda biz bu metodu bir dizi referansıyla çağırmalıyız. Böylece metoda dizi nesnesinin adresi katarılmış olur. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int[] a = { 4, 7, 23, 45, 67, 89, 32 };
            int max;

            MyArray.Disp(a);
            max = MyArray.GetMax(a);
            Console.WriteLine("Max = {0}", max);
        }
    }

    class MyArray
    {
        public static int GetMax(int[] a)
        {
            int max = a[0];

            for (int i = 1; i < a.Length; ++i)
                if (a[i] > max)
                    max = a[i];

            return max;
        }

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

    }
}
```

Dizi yaratılır yaratılmaz argüman olarak da geçilebilir. Örneğin:

```
Sample.Foo(new int[] {1, 2, 3, 4, 5});
```

int türden bir dizinin ortalamasını veren metot şöyle yazılabilir:

```
using System;

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

```
        {
            double result;

            result = Sample.Avg(new int[] { 1, 2, 3, 4, 5 });
            Console.WriteLine(result);

            result = Sample.Avg(new int[] { 1, 1, 1, 1, 2 });
            Console.WriteLine(result);
        }
    }

    class Sample
    {
        public static double Avg(int[] a)
        {
            int total = 0;

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

            return (double)total / a.Length;
        }
    }
}
```

Örnek şöyle de olabilirdi:

```
using System;

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

            Console.Write("Dizi uzunluğunu giriniz:");
            n = int.Parse(Console.ReadLine());
            a = new int[n];

            for (int i = 0; i < n; ++i)
            {
                Console.Write("{0}'inci elemanı giriniz:", i + 1);
                a[i] = int.Parse(Console.ReadLine());
            }
            max = Sample.Avg(a);
            Console.WriteLine("Ortalama = {0}", max);
        }
    }

    class Sample
    {
        public static double Avg(int[] a)
        {
            int total = 0;

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

            return (double)total / a.Length;
        }
    }
}
```

C#'ta sıfır elemanlı diziler yaratılabilir. Örneğin:

```
int[] a = new int[0];
```

Dizilerin Sıraya Dizilmesi

Dizi elemanlarının sıraya dizilmesine İngilizce "sorting" denilmektedir. Dizi elemanlarını sıraya dizilmesi için pek çok algoritma kullanılabilmektedir. “Kabarcık sıralaması (bubble sort)” yönteminde dizinin yan yana iki elemanı karşılaştırılır duruma göre yer değiştirilir. Tabi bu bir kez yapılmaz. Her yinelemede en büyük eleman daraltılmış dizinin sonuna gider. Böylece her yinelemede eskisinden bir gereyiye kadar gitmek yeterli olur. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int[] a = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

            Sort.Bubble(a);
            Sort.Disp(a);
        }
    }

    class Sort
    {
        public static void Bubble(int[]a)
        {
            for (int i = 0; i < a.Length - 1; ++i)
                for (int k = 0; k < a.Length - 1 - i; ++k)
                    if (a[k] > a[k + 1])
                    {
                        int temp = a[k];
                        a[k] = a[k + 1];
                        a[k + 1] = temp;
                    }
        }

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

Seçerek sıralama yönteminde dizinin en küçük elemanı bulunur, ilk elemanla yer değiştirilir. Dizi bir daraltılır, aynı şey daraltılmış dizi için de yapılır. Böyle ilerlenir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int[] a = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

            Sort.Selection(a);
            Sort.Disp(a);
        }
    }

    class Sort
```

```
    {
        public static void Selection(int[]a)
        {
            int min, minIndex;

            for (int i = 0; i < a.Length - 1; ++i)
            {
                min = a[i];
                minIndex = i;

                for (int k = i + 1; k < a.Length; ++k)
                    if (a[k] < min)
                    {
                        min = a[k];
                        minIndex = k;
                    }

                a[minIndex] = a[i];
                a[i] = min;
            }
        }

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

Metotların Geri Dönüş Değerlerinin Dizi Türünden Olması Durumu

Bir metodun geri dönüş değeri bir dizi türünden olabilir. Bu durumda geri dönüş değerinin türü yerine T bir tür belirtmek üzere T[] yazılır. Örneğin:

```
public static int[] Foo()
{
      //...
}
```

Dizi metot içerisinde new operatörüyle yaratılıp onun referansı geri dönüş değeri olarak veriliyor olabilir. Örneğin:

```
using System;

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

            a = Sample.GetRandomNumbers(1, 100, 10);
            for (int i = 0; i < a.Length; ++i)
                Console.Write("{0} ", a[i]);
            Console.WriteLine();
        }
    }

    class Sample
    {
        public static int[] GetRandomNumbers(int min, int max, int n)
        {
            Random rand = new Random();
            int[] a = new int[n];
```

```
            for (int i = 0; i < n; ++i)
                a[i] = rand.Next(min, max + 1);

            return a;
        }
    }
}
```

Bir dizi referansına geri dönen metodun geri dönüş değeri aynı türden bir dizi referansına atanmalıdır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Lotto lotto = new Lotto();
            int[] column;

            column = lotto.GetRandomColumn();
            Lotto.Disp(column);
        }
    }

    class Lotto
    {
        public Random rand;

        public Lotto()
        {
            rand = new Random();
        }

        public int[] GetRandomColumn()
        {
            int[] column = new int[6];
            int val;
            bool repeat;

            for (int k = 0; k < 6; ++k)
            {
                do
                {
                    repeat = false;
                    val = rand.Next(1, 50);
                    for (int i = 0; i < k; ++i)
                        if (column[i] == val)
                        {
                            repeat = true;
                            break;
                        }
                } while (repeat);
                column[k] = val;
            }

            return column;
        }

        public static void Disp(int[] a)
        {
            for (int i = 0; i < a.Length; ++i)
                Console.Write("{0} ", a[i]);

            Console.WriteLine();
        }
    }
```

```
}
```

Sınıf Çalışması: Prime isminde bir sınıfın içerisine aşağıdaki gibi static GetPrimes isimli bir metot yerleştiriniz

```
class Prime
{
    public static int[] GetPrimes(int n)
    {
        //...
    }

    public static bool IsPrime(int val)
    {
        //...
    }
}
```

Bu metot ilk n tane asal sayıyı bulup onu bir diziye yerletirip o dizinin referansıyla geri dönüyor olsun. Kodu şöyle test edebilirsiniz:

```
class App
{
    public static void Main()
    {
        int[] primes;

        primes = Prime.GetPrimes(10);
        for (int i = 0; i < primes.Length; ++i)
            Console.Write("{0} ", primes[i]);
        Console.WriteLine();
    }
}
```

Çözüm:

```
using System;

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

            primes = Prime.GetPrimes(100);
            for (int i = 0; i < primes.Length; ++i)
                Console.Write("{0} ", primes[i]);
            Console.WriteLine();
        }
    }

    class Prime
    {
        public static int[] GetPrimes(int n)
        {
            int[] a = new int[n];
            int count = 0;

            for (int i = 2; count < n; ++i)
                if (IsPrime(i))
                    a[count++] = i;

            return a;
        }

        public static bool IsPrime(int val)
        {
            if (val % 2 == 0)
```

```
            return val == 2;
        for (int i = 3; i * i <= val; i += 2)
            if (val % i == 0)
                return false;
        return true;
    }
  }
}
```

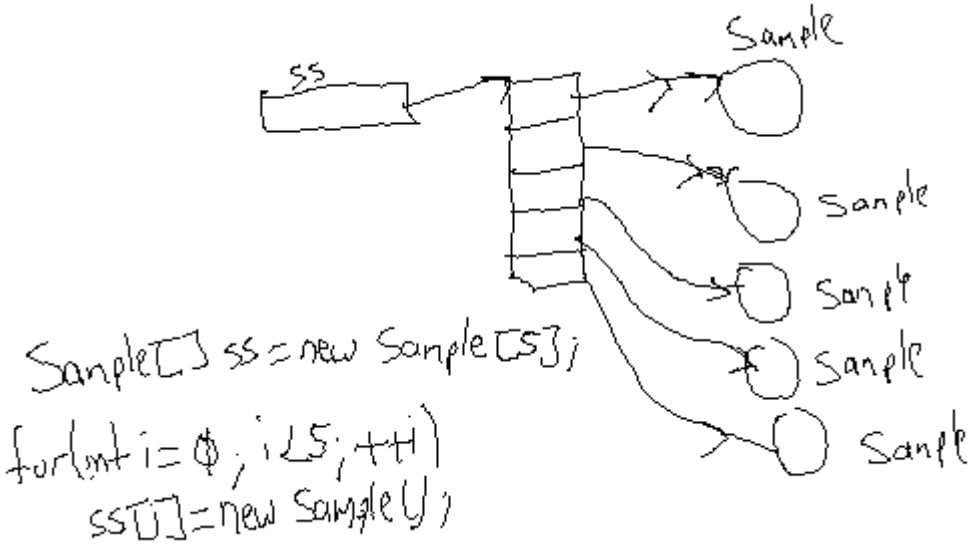
Sınıf Türünden Diziler

Sınıflar türünden diziler söz konusu olabilir. Bir sınıf türünden dizi aslında bir referans dizisidir. Yani bu tür dizilerin her elemanı bir sınıf nesnesinin adresini tutar. Örneğin:

```
Sample[] ss = new Sample[5];
```

Burada ss 5 elemanlı bir Sample dizisidir. Dizinin her elemanı bir Sample nesnesinin adresini tutabilecek bir referanstır. O halde bizim ayrıca 5 tane Sample nesnesi yaratıp onların adreslerini dizi elemanlarına yerleştirmemiz gerekir. Örneğin:

```
for (int i = 0; i < 5; ++i)
      ss[i] = new Sample();
```



Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample[] ss = new Sample[5];

            for (int i = 0; i < 5; ++i)
                ss[i] = new Sample(i);

            for (int i = 0; i < 5; ++i)
                ss[i].Disp();
        }
    }

    class Sample
    {
```

```
        public int val;

        public Sample(int v)
        {
            val = v;
        }

        public void Disp()
        {
            Console.WriteLine(val);
        }
    }
}
```

new operatöründe sınıf dizileri için de ilkdeğerler verebiliriz. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample[] ss;

            ss = new Sample[] { new Sample(10), new Sample(20), new Sample(30),
                                new Sample(40), new Sample(50) };
            for (int i = 0; i < ss.Length; ++i)
                ss[i].Disp();
        }
    }

    class Sample
    {
        public int val;

        public Sample(int v)
        {
            val = v;
        }

        public void Disp()
        {
            Console.WriteLine(val);
        }
    }
}
```

İki tırnak içerisindeki yazılar birer string referansı belirttiğine göre biz bunları bir string dizisinin elemanlarına atayabiliriz. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            string[] names = new string[5];

            names[0] = "ali";
            names[1] = "veli";
            names[2] = "selami";
            names[3] = "ayşe";
            names[4] = "fatma";
```

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

Aynı şey şöyle de yapılabilirdi:

```
using System;

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

            names = new string[5] { "ali", "veli", "selami", "ayşe", "fatma" };

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

Hatta şöyle de olabilirdi:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            string[] names = { "ali", "veli", "selami", "ayşe", "fatma" };

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

Sınıf Çalışması: Program önce bizden kaç isim girileceğini sorsun. Sonra bu isimleri tutabilecek bir string dizisi oluşturunuz. Bu dizinin her elemanı için Console.ReadLine metodu ile bir isim isteyiniz. Sonra bu diziyi dolaşarak isimleri aralarına virgül koyarak yazdırınız.

Çözüm:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int n;
            string[] names;

            Console.Write("Kaç isim girmek istiyorsunuz? ");
            n = int.Parse(Console.ReadLine());
            names = new string[n];

            for (int i = 0; i < n; ++i)
            {
```

```
            Console.Write("{0}. ismi giriniz:", i + 1);
            names[i] = Console.ReadLine();
        }

        for (int i = 0; i < n; ++i)
        {
            if (i != 0)
                Console.Write(", ");
            Console.Write(names[i]);
        }
        Console.WriteLine();
    }
  }
}
```

System.IO isim alanındaki Directory isimli sınıfın GetFiles isimli static metodu şöyledir:

```
public static string[] GetFiles(string path)
```

Metot parametresiyle aldığı dizindeki tüm dosyaları elde eder. Onların isimlerini bir string dizisine yerleştirir ve o dizinin referansı ile geri döner. Dosya isimleri tam yol ifadesi (full path) içermektedir. Örneğin:

```
using System;
using System.IO;

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

            files = Directory.GetFiles(@"c:\windows");
            for (int i = 0; i < files.Length; ++i)
                Console.WriteLine(files[i]);
        }
    }
}
```

Sınıf Çalışması: Yukarıdaki programı öyle hale getiriniz ki dosyaların yalnızca isimleri ve uzantıları görüntülensin

Çözüm:

```
using System;
using System.IO;

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

            files = Directory.GetFiles(@"c:\windows");
            for (int i = 0; i < files.Length; ++i)
            {
                int index = files[i].LastIndexOf('\\');
                file = files[i].Substring(index + 1);
                Console.WriteLine(file);
            }
        }
    }
}
```

Daha kompakt kodlama şöyle olabilirdi:

```
using System;
using System.IO;

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

            files = Directory.GetFiles(@"c:\windows");
            for (int i = 0; i < files.Length; ++i)
                Console.WriteLine(files[i].Substring(files[i].LastIndexOf('\\') + 1));
        }
    }
}
```

Aslında bir yol ifadesinin çeşitli bileşenlerini bize veren Path isimli bir sınıf da vardır. Path sınıfı da System.IO isim alanı içerisindedir. Örneğin Path sınıfının GetFileName isimli static metodu bize yol ifadesi ile belirtilen dosyanın ismini ve uzantısını, static GetExtension isimli metodu yol ifadesi ile belirtilen dosyanın uzantısını ve GetDirectoryName isimli static metodu ise yol ifadesi ile belirtilen dosyanın içinde bulunduğu dizinin yol ifadesini verir.

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            string path = @"c:\windows\system32\a.dll";
            string result;

            result = Path.GetFileName(path);
            Console.WriteLine(result);

            result = Path.GetExtension(path);
            Console.WriteLine(result);

            result = Path.GetDirectoryName(path);
            Console.WriteLine(result);
        }
    }
}
```

Path sınıfı ilgili yol ifadesi ile belirtilen dosyanın var olup olmadığına bakmamaktadır. Yalnızca yol ifadesini bileşenlerine ayırıp bize ilgili bileşeni vermektedir.

Örneğin:

```
using System;
using System.IO;

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

```
            files = Directory.GetFiles(@"c:\windows");
            for (int i = 0; i < files.Length; ++i)
                Console.WriteLine(Path.GetFileName(files[i]));
        }
    }
}
```

Directory sınıfının GetDirectories isimli metodu o dizin içerisindeki dizinleri elde etmekte kullanılır.

Örneğin:

```
using System;
using System.IO;

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

            dirs = Directory.GetDirectories(@"c:\windows");
            for (int i = 0; i < dirs.Length; ++i)
                Console.WriteLine("{0, -40} <DIR>", dirs[i].Substring(dirs[i].LastIndexOf('\\') + 1));

            files = Directory.GetFiles(@"c:\windows");
            for (int i = 0; i < files.Length; ++i)
                Console.WriteLine(files[i].Substring(files[i].LastIndexOf('\\') + 1));
        }
    }
}
```

Sınıf Çalışması: c:\windows dizini içerisindeki yalnızca uzantısısı .exe olan dosyaları yazdırınız (yol ifadesi olmadan)

Çözüm:

```
using System;
using System.IO;

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

            files = Directory.GetFiles(@"c:\windows");
            for (int i = 0; i < files.Length; ++i)
                if (Path.GetExtension(files[i]).ToLower() == ".exe")
                    Console.WriteLine(Path.GetFileName(files[i]));
        }
    }
}
```

Soru: 3 basamaklı bir sayıyı yazı olarak yazdırınız

Yanıt:

```
using System;

namespace CSD
{
    class App
```

```
    {
        public static void Main()
        {
            int val;
            string result;

            for (;;)
            {
                Console.Write("Bir sayı giriniz:");
                val = int.Parse(Console.ReadLine());
                result = Number.ToText(val);
                Console.WriteLine(result);
                if (val == 0)
                    break;
            }
        }
    }

    class Number
    {
        public static string ToText(int val)
        {
            string[] ones = { "", "bir", "iki", "üç", "dört", "beş", "altı", "yedi", "sekiz", "dokuz" };
            string[] tens = { "", "on", "yirmi", "otuz", "kırk", "elli", "altmış", "yetmiş",
                              "seksen", "doksan" };

            int one, ten, hundred;
            string result = "";

            hundred = val / 100;
            ten = val / 10 % 10;
            one = val % 10;

            if (val == 0)
                return "sıfır";

            if (hundred > 0)
            {
                if (hundred != 1)
                    result += ones[hundred] + " ";
                result += "yüz";
                if (ten > 0 || one > 0)
                    result += " ";
            }
            if (ten > 0)
            {
                result += tens[ten];
                if (one > 0)
                    result += " ";
            }

            if (one > 0)
                result += ones[one];

            return result;
        }
    }
}
```

Sınıf Çalışması: Klavyeden ulong türden bir sayı isteyiniz. Bu sayıyı üçerli basamaklara ayırarak int türden bir diziye yerleştiriniz. Sonra diziyi dolaşarak bu üçerli basamakları alt alta yazdırınız. Örneğin sayının 123456789 olduğunu varsayalım bu durumda dizi elemanları şöyle olacaktır:

int diziyi daha büyük açıp baş kısmını doldurabilirsiniz ya da tam gerektiği kadar uzunlukta açabilirisiniz.

Yanıt:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            ulong val, temp;
            int[] threes;
            int n = 0;

            Console.Write("Bir sayı giriniz:");
            val = ulong.Parse(Console.ReadLine());

            temp = val;
            while (temp != 0)
            {
                ++n;
                temp /= 1000;
            }
            threes = new int[n];

            for (int i = n - 1; val != 0; --i)
            {
                threes[i] = (int) (val % 1000);
                val /= 1000;
            }

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

Çok Boyutlu Diziler

Bazen doğadaki bazı olgular çok boyutlu dizilerle daha iyi temsil edilebilmektedir. Örneğin bir satranç tahtası, bir bulmaca iki boyutlu bir diziyle daha iyi temsil edilebilir. Tabii aslında bellek tek boyutludur. Dolayısıyla çok boyutlu dizi kavramı doğal bir kavram değildir. Derleyiciler çok boyutlu dizileri aslında bellekte tek boyutlu dizilermiş gibi saklamaktadır. Çok boyutlu diziler uygulamada karşımıza genellikle iki boyutlu olarak çıkar. İki boyutlu dizilere matris de denilebilmektedir

C#'ta çok boyutlu dizi türleri köşeli parantez içerisinde “boyut sayısı - 1 tane” virgül atomu konularak temsil edilir. Örneğin T bir tür olmak üzere T[,,] türü T türünden üç boyutlu dizi türü anlamına gelir. Dolayısıyla T[] türü de T türünden tek boyutlu dizi anlamına gelmektedir. Örneğin:

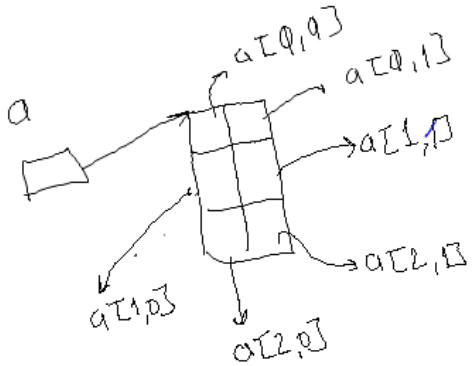
```
int[] a;            // a tek boyutlu dizi türünden referans
int[,] b;           // b iki, boyutlu dizi türünden referans
int[,,] c;          // c üç boyutlu dizi türünden referans
```

T[,,] gibi bir tür sembolik olarak "T köşeli parantez virgül, virgül" biçiminde okunmalıdır.

Çok boyutlu dizi türünden nesneler new operatöründe köşeli parantez içerisinde her boyutun uzunluğu belirtilerek yaratılır. Örneğin:

```
int[,] a = new int[3, 2];
```

Dizi elemanlarına erişirken her boyutun indeksi sıfırdan başlar. Örneğin:



Çok boyutluy dizilerin elemanlarına erişirken köşeli parantezler içerisinde her boyut için bir indeks bulundurulur.

Örneğin:

```
using System;

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

            a = new int[3, 4];

            for (int i = 0; i < 3; ++i)
                for (int k = 0; k < 4; ++k)
                    a[i, k] = i + k;

            for (int i = 0; i < 3; ++i)
            {
                for (int k = 0; k < 4; ++k)
                    Console.Write("{0} ", a[i, k]);
                Console.WriteLine();
            }
        }
    }
}
```

Çok boyutlu dizilere de new operatörü ile tahsisat sırasında ilkdeğer verilebilir. Bunun için her boyutun yine ayrıca küme parantezlerine alınması gerekir. Örneğin:

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

Örneğin:

```
using System;

namespace CSD
```

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

            a = new int[3, 4] {
                {1, 2, 3, 4},
                {5, 6, 7, 8},
                {9, 10, 11, 12}
            };


            for (int i = 0; i < 3; ++i)
            {
                for (int k = 0; k < 4; ++k)
                    Console.Write("{0,-4}", a[i, k]);
                Console.WriteLine();
            }
        }
    }
}
```

Çok boyutlu dizilerde yine boyut uzunlukları hiç belirtilmeyebilir. Tabi bu durumda yine boyut belirten her küme parantezinin içerisinde eşit sayıda eleman bulunmak zorundadır. Örneğin:

```
int[,] a;

a = new int[, ] {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

Yine çok boyutlu dizi referanslarına ilkdeğer verilirken new anahtar sözcüğü hiç kullanılmayabilir. Örneğin:

```
int[,] a = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
```

Çok boyutlu dizilerde Length property'si dizideki toplam eleman sayısını bize verir. Fakat istenirse her boyutun uzunluğu ayrı ayrı da elde edilebilmektedir. Bunun için dizi referansıyla GetLength isimli metodun çağrılması gerekir. Bu metot hangi boyutunun uzunluğunu almak istediğimizi belirten bir boyut indeksini bizden ister. İlk boyut sıfırıncı indeksle belirtilmektedir. Örneğin:

```
using System;

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

            a = new int[, ] {
                {1, 2, 3, 4},
                {5, 6, 7, 8},
                {9, 10, 11, 12}
            };

            Console.WriteLine("{0}X{1}", a.GetLength(0), a.GetLength(1));
            Console.WriteLine(a.Rank);
        }
    }
}
```

Ayrıca dizilerin read-only Rank isimli property'leri dizinin kaç boyutlu olduğu bilgisini bize verir.

Sınıf Çalışması: Bir kare matris içerisindeki her satır ve her sütunun içerisinde 1'den kare matrisin uzunluğuna kadar sayılardan yalnızca bir tane olup olmadığını belirleyen CheckMatrix metodunu yazınız.

```
public static bool CheckMatrix(int[,] a)
```

Programı aşağıdaki kodla test edebilirsiniz:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int[,] a = new int[9, 9]
            {
               {5, 3, 4, 6, 7, 8, 9, 1, 2 },
               {6, 7, 2, 1, 9, 5, 3, 4, 8 },
               { 1, 9, 8, 3, 4, 2, 5, 6, 7},
               {8, 5, 9, 7, 6, 1, 4, 5, 3 },
               {4, 2, 6,  8, 5, 3, 7, 9, 1},
               {7, 1, 3, 9, 2, 4, 8, 5, 6 },
               {9, 6, 1, 5, 3, 7, 2, 8, 4 },
               {2, 8, 7, 4, 1, 9, 6, 3, 5},
               {3, 4, 5, 2, 8, 6, 1, 7, 9 }
           };

            Console.WriteLine(CheckMatrix(a) ? "Geçerli" : "Geçersiz");
        }

        public static bool CheckMatrix(int[,] m)
        {
            //...
            return false;
        }
    }
}
```

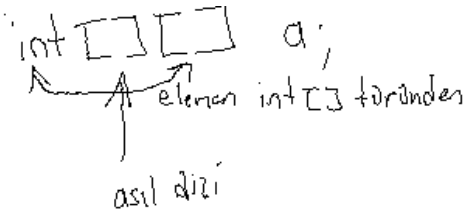
Çözüm:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int[,] a = new int[9, 9]
            {
               {5, 3, 4, 6, 7, 8, 9, 1, 2 },
               {6, 7, 2, 1, 9, 5, 3, 4, 8 },
               { 1, 9, 8, 3, 4, 2, 5, 6, 7},
               {8, 5, 9, 7, 6, 1, 4, 2, 3 },
               {4, 2, 6,  8, 5, 3, 7, 9, 1},
               {7, 1, 3, 9, 2, 4, 8, 5, 6 },
               {9, 6, 1, 5, 3, 7, 2, 8, 4 },
               {2, 8, 7, 4, 1, 9, 6, 3, 5},
               {3, 4, 5, 2, 8, 6, 1, 7, 9 }
           };

            Console.WriteLine(CheckMatrix(a) ? "Geçerli" : "Geçersiz");
        }
```

```
        public static bool CheckMatrix(int[,] m)
        {
            int size = m.GetLength(0);
            bool[] checkArray;

            for (int i = 0; i < size; ++i)      // satırlar için
            {
                checkArray = new bool[size];
                for (int k = 0; k < size; ++k)
                {
                    if (m[i, k] < 1 || m[i, k] > size)
                        return false;
                    if (checkArray[m[i, k] - 1])
                        return false;
                    checkArray[m[i, k] - 1] = true;
                }
            }

            for (int i = 0; i < size; ++i)      // sütunlar için
            {
                checkArray = new bool[size];
                for (int k = 0; k < size; ++k)
                {
                    if (checkArray[m[k, i] - 1])
                        return false;
                    checkArray[m[k, i] - 1] = true;
                }
            }

            return true;
        }
    }
}
```

Dizi Dizileri (Jagged Array)

Her elemanı bir dizinin referansını (yani adresini) tutan dizilere dizi dizileri denilmektedir. Dizi dizileri birden fazla köşeli parantez ile ifade edilir. İlk köşeli parantez her zaman asıl diziyi belirtir. Diğerleri elemanın türüne ilişkindir. Örneğin:



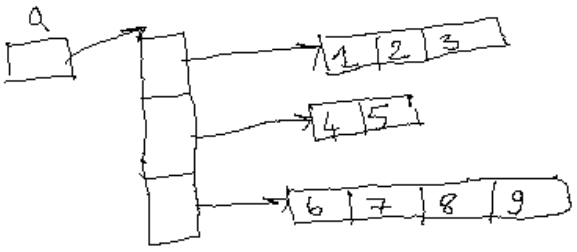
İlk köşeli parantez elle kapatılırsa görülen şey dizi elemanlarının hangi türden olduğudur. Yukarıdaki örnekte dizi int[] türünden referansları tutmaktadır.

Dizi dizileri için tahsisat yapılırken new operatöründe yalnızca ilk köşeli parantezin içerisine uzunluk yazılır. Diğerleri boş bırakılır. Çünkü onlar türe ilişkindir. Örneğin:

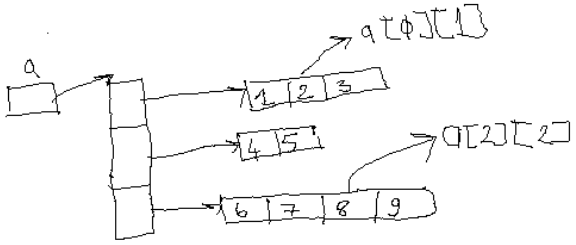
```
int[][] a = new int[3][];
```

Burada a dizisinin her elemanı int[] türündendir. Bu elemanlara biz dizilerin adreslerini atamalıyız. Örneğin:

```
a[0] = new int[] {1, 2, 3};
a[1] = new int[] {4, 5};
a[2] = new int[] {6, 7, 8, 9};
```



Burada a int[][] ("int köşeli parantez köşeli parantez" biçiminde onunur) türündendir. a[i] ise int[] türündendir. Böylece dizilerinde bir elemana birden fazla köşeli parantezle erişilir. a[i][k] ifadesi a dizisinin i'inci elemanın belirttiği dizinin k'ıncı elemanıdır. Örneğin:



Örneğin:

```
using System;

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

            a = new int[3][];
            a[0] = new int[] { 1, 2, 3 };
            a[1] = new int[] { 4, 5 };
            a[2] = new int[] { 6, 7, 8, 9};

            for (int i = 0; i < a.Length; ++i)
            {
                for (int k = 0; k < a[i].Length; ++k)
                    Console.Write("{0} ", a[i][k]);
                Console.WriteLine();
            }
        }
    }
}
```

Örneğin:

```
using System;

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

            students = new string[4][];

            students[0] = new string[5] { "Ali", "Veli", "Selami", "Ayşe", "Fatma" };
            students[1] = new string[3] { "Emine", "Sacit", "Ahmet" };
            students[2] = new string[4] { "Erhan", "Ayhan", "Sibel", "Şükrü" };
```

```
            students[3] = new string[2] { "İnci", "İsmet" };

            for (int i = 0; i < students.Length; ++i)
            {
                for (int k = 0; k < students[i].Length; ++k)
                {
                    if (k != 0)
                        Console.Write(", ");
                    Console.Write("{0}", students[i][k]);
                }
                Console.WriteLine();
            }
        }
    }
}
```

Dizi dizlerine de new operatörü sırasında ilkdeğer verilebilir:

```
using System;

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

            students = new string[4][] {
                new string[5] { "Ali", "Veli", "Selami", "Ayşe", "Fatma" },
                new string[3] { "Emine", "Sacit", "Ahmet" },
                new string[4] { "Erhan", "Ayhan", "Sibel", "Şükrü" },
                new string[2] { "İnci", "İsmet" }
            };

            for (int i = 0; i < students.Length; ++i)
            {
                for (int k = 0; k < students[i].Length; ++k)
                {
                    if (k != 0)
                        Console.Write(", ");
                    Console.Write("{0}", students[i][k]);
                }
                Console.WriteLine();
            }
        }
    }
}
```

Çok Boyutlu Dizilerle Dizi Dizileri Arasındaki Benzerlikler ve Farklılıklar

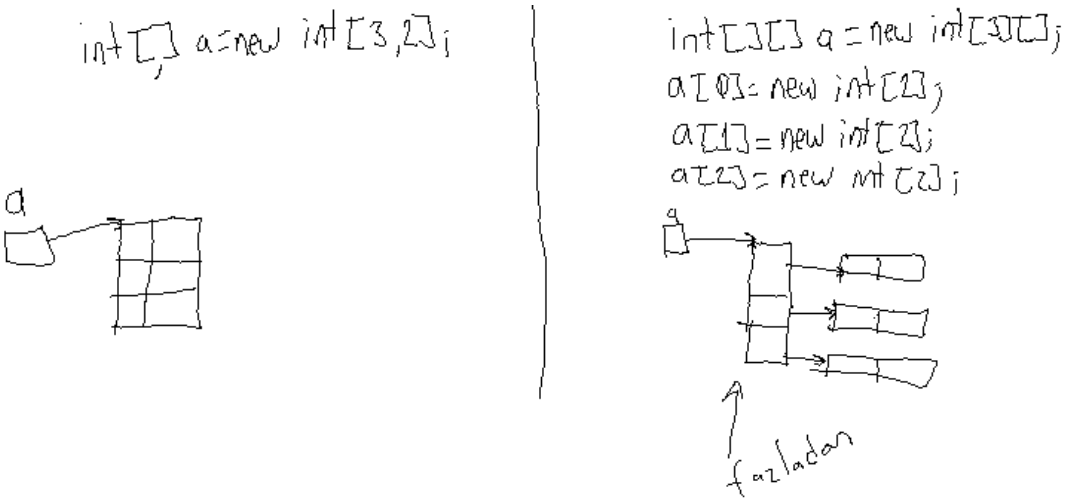
Hem çok boyutlu diziler hem de dizi dizileri matrisel kavramları ifade etmekte kullanılabilir. Bu bakımdan çok boyutlu dizilerle dizi dizileri kullanım amacı bakımından birbirlerine benzemektedir. Bunların arasındaki farklılıklar dört maddeyle açıklanabilir:

1) Çok boyutlu diziler köşeli partantez içerisinde virgülle, dizi dizileri birden fazla köşeli parantezle bildirilirler. Örneğin:

```
int[,] a;
int[][] b;
```

2) Çok boyutlu dizilerde elemanlara a[i, k] sentaksı ile dizi dizilerinde ise a[i][k] sentaksı ile erişilmektedir.

3) Dizi dizileri çok boyutlu dizilere göre toplamda bellekte daha fazla yer kaplamaktadır. Çünkü asıl dizi ide yer kaplamaktadır. Örneğin:



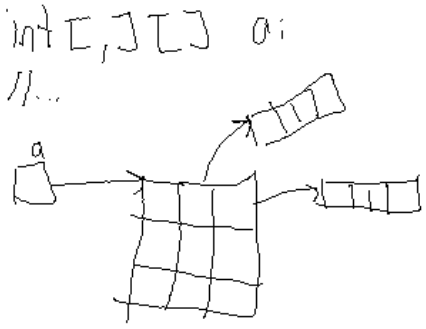
4) Çok boyutlu dizilerde her satırda aynı sayıda eleman bulunur. Fakat dizi dizilerinde her satırda farklı sayıda elemanlar bulunabilmektedir. Örneğin "5 sınıfı olan bir okulda her sınıfta 30 öğrenci varsa" onların isimlerini tutmak için çok boyutlu dizi (5 satır, 30 sütun) daha uygundur. Ancak "eğer her sınıfta aynı sayıda öğrenci yoksa" bu durumda dizi dizileri daha uygun olur.

Dizi Dizilerine İlişkin Karmaşık Durumlar

Çok boyutlu bir dizi dizisi söz konusu olabilir. Örneğin:

```
int[,][] a;
```

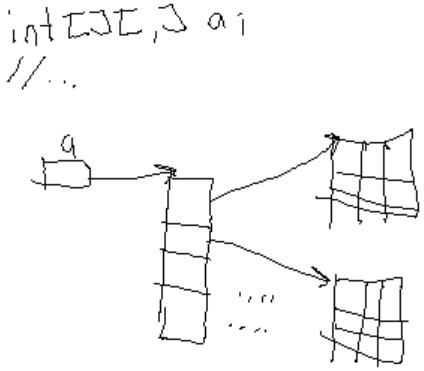
Burada asıl dizi iki boyutludur. Bu iki boyutlu dizinin her elemanı int[] türündendir:



Bir dizi dizisi çok boyutlu dizileri tutuyor olabilir. Örneğin:

```
int[][,] a;
```

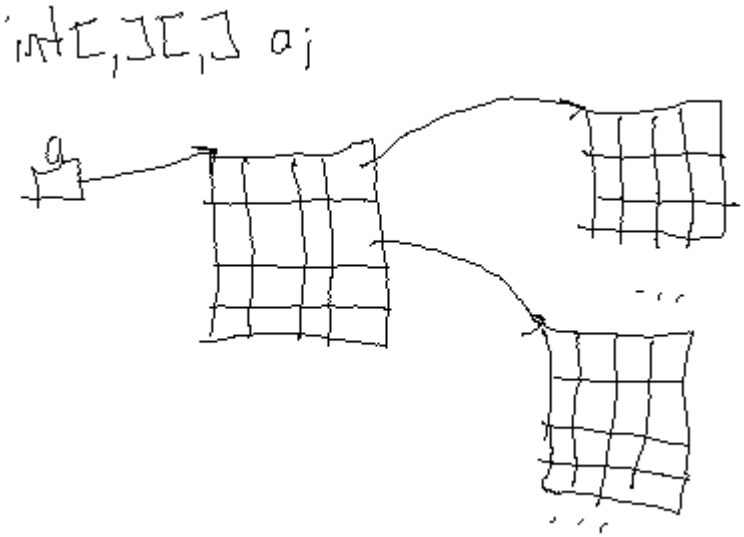
Burada asıl dizi tek boyutludur. Bu dizinin her elemanı iki boyutlu bir diziyi göstermektedir:



Tabii çok boyutlu bir dizi dizisinin her elemanı da çok boyutlu bir dizi olabilir. Örneğin:

```
int[,][,] a;
```

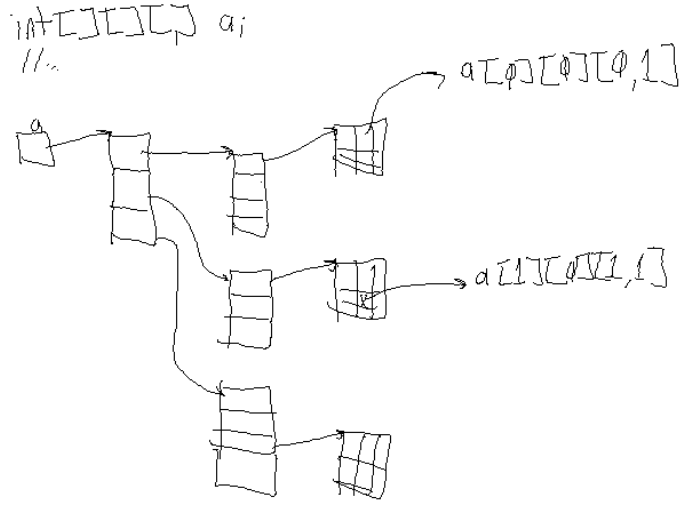
Burada asıl dizi iki boyutludur. Bu dizinin elemanları iki boyutlu dizileri göstermektedir.



Bir dizi dizisi, dizi dizilerinden oluşabilir. Örneğin:

```
int[][][,] a;
```

Burada asıl dizi tek boyutludur. Bu dizi dizi dizilerinin referanslarını tutar. Onlar da iki boyutlu dizileri tutmaktadır:



foreach Döngüleri

foreach döngüleri dizi gibi işleme sokulabilen her türlü sınıflarla ve yapılarla kullanılabilmektedir. Teknik olarak foreach IEnumerable arayüzünü destekleyen sınıflarla ve yapılarla kullanılabilir. C#'taki diziler de IEnumerable arayüzünü desteklediği için foreach onlarla da kullanılabilmektedir.

foreach döngüsünün genel biçimi şöyledir:

```
foreach (<tür> <döngü değişkeninin ismi> in <dizilim ismi>)
      <deyim>
```

foreach döngüsü şöyle çalışır: Her yinelemede dizilimin bir sonraki elemanı döngü değişkenine aktarılır. Programcı onu döngü deyiminde kullanır. Dizilimin tüm elemanları bitince döngü de bitmiş olur. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int[] a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

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

Şüphesiz foreach döngüsüyle yapılabilen her şey for döngüsüyle de yapılabilmektedir. Ancak foreach döngüleri bazı durumlarda çok pratik kullanıma sahiptir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            string[] names = { "ali", "veli", "selami", "ayşe", "fatma" };

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

Örneğin:

```
using System;
using System.IO;

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

            foreach (string file in Directory.GetFiles(@"c:\windows"))
            {
                index = file.LastIndexOf('\\');
```

```
                Console.WriteLine(file.Substring(index + 1));
            }
        }
    }
}
```

foreach döngüsünün döngü değişkeni read-only'dir. Yani biz bunun içindeki değeri kullanabiliriz fakat ona yeni değer atayamayız. Örneğin:

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

foreach (int x in a)
{
    x = 10;        // error!
}
//...
```

foreach döngüsünde dizilimin elemanları döngü değişkenine tür dönüştürme operatöryle atanmaktadır. Dolayısıyla döngü değişkeninin türüyle dizilimin türü farklı olabilir. Örneğin:

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

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

O halde:

```
foreach (T x in a)
{
      //...
}
```

işleminin eşdeğeri (fakat tam değil) şöyledir:

```
for (int i = 0; i < a.Length; ++i)
{
      T x = (T) a[i];
      //...
}
```

foreach döngüsünde döngü değişkeni yalnızca döngünün içerisinde kullanılır, döngünün dışarısından kullanılamaz.

foreach ile çok boyutlu bir dizi de dolaşılabilir. Bu durumda dizinin tüm elemanları elde edilir. Örneğin:

```
using System;

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

            a = new int[3, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 } };
            foreach (int x in a)
                Console.Write("{0} ", x);
            Console.WriteLine();
        }
    }
}
```

Dizi dizileri de foreach deyimi ile dolaşılabilir. Tabi bu durumda dizinin her elemanı bir dizi olur. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int[][] a = new int[3][]
            {
                new int[] { 1, 2, 3 },
                new int[] { 4, 5 },
                new int[] { 6, 7, 8, 9 }
            };

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

foreach döngüleri her derde deva değildir. Biz bazı işlemleri foreach döngüleriyle yapamayız. Örneğin foreach döngüsü ile bir diziyi tersten dolaşamayız. Onun için normal for döngüsü gerekir:

```
using System;

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

            for (int i = a.Length - 1; i >= 0; --i)
                Console.Write("{0} ", a[i]);
            Console.WriteLine();
        }
    }
}
```

Ya da diziyi atlaya atlaya (örneğin birer atlayarak) foreach döngüsüyle etkin dolaşamayız. Ya da örneğin dizinin yan yana elemanlarını foreach döngüsüyle işleme sokamayız. Dizinin belli bir elemanından itibaren geri kalan kısmını yibe foreach döngüsüyle dolaşamayız.

Sınıflarda Temel Erişim Kuralları

Sınıfların elemanları beş erişim belirleyicisinden birine sahip olabilir:

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

Biz bir sınıfı bu bağlamda bölümlerden oluşuyormuş gibi düşünebiliriz. Sınıfın public elemanları puıblic bölümünü, private elemanları private bölümünü vs. oluşturmaktadır.

Sınıflardaki temel erişim kuralları iki maddeyle özetlenebilir:

1) Sınıfın dışından o sınıf türünden referans ya da sınıf ismi kullanılarak sınıfın yalnızca public bölümüne erişilebilir.

2) Sınıfın kendi içerisinden o sınıfın her bölümüne erişilebilir.

Ayrıca nesne yaratılırken başlangıç metodunun da erişilebilir olması gerekmektedir. Yani biz sınıfın dışından new operatörüyle o sınıf türünden nesne yaratırken sınıfın başlangıç metodunun public bölümde olması gerekir. (Sınıf için hiçbir başlangıç metodu yazmamışsak derleyici içi boş başlangıç metodunu public bölümde yazar)

Sınıfın internal bölümü aynı assembly'de (yani dll'de ya da .exe'de) public, başka bir assembly'de private etkisi yaratan bölümdür. Örneğin A.DLL'nin içerisinde Sample ve Mample isimli iki sınıf olsun. Biz Mample içerisinden Sample'daki internal elemanlara erişebiliriz. Çünkü burada erişim aynı assembly'den (yani DLL'den) yapılmaktadır. Ancak Sample'daki internal elemanlara örneğin bir exe içerisindeki sınıftan erişemeyiz.

Sınıfın protected bölümü dışarıdan erişime kapalı fakat türemiş sınıflardan erişme açıktır. protected bölüm türetme konusunda ele alınacaktır.

Sınıfın protected internal bölümü aynı assembly'den erişimlerde public, başka assembly'den erişimlerde protected etkisi yaratıyor. (Yani aslında internal yerine private internal yazımı daha uygun olabilirdi. Fakat zaten private default olduğu için yalnızca internal bunun için yeterli görülmüştür.)

Sınıfın en korunaklı olandan en az korunaklı olana doğru bölümlerinin sıralaması şöyledir:

private
protected
internal
protected internal
public

Kapsülleme Prensibi (Encapsulation)

Kapsülleme NYPT'nin en önemli anahtar kavramlarından biridir. Kapsülleme "bir olgunun bir sınıf ile temsil edilip onun dışarıyı ilgilendirmeyen elemanlarını private bölüme yerleştirerek dışarıya yalnızca public arayüzü ile sunmak" anlamına gelir. Böylece iyi kapsüllenmiş sınıflar onu kullananların kafasını karıştırmazlar. Aslında kapsülleme dış dünyada da karşımıza çıkan bir olgudur. Örneğin bir otomobilin gaz, fren, debriyaj elemanları public elemanlardır. Kaputun içerisindekiler ise private bölümü oluşturmaktadır. Ya da örneğin televizyonun kumandası public bölümü oluşturur. Televizyonun içendekiler private bölümdedir.

Örneğin bir sınıfta dışarıdan çağrılabilecek DoSomethingImportant isimli bir metot bulunuyor olsun. Bu metodun da kendi içerisinde Foo, Bar ve Tar metotlarını çağırdığını düşünelim. Foo, Bar ve Tar metotları işin belirli parçalarını yapan metotlardır. Bu durumda onların gereksiz bir biçimde public bölüme yerleştirilmesi hem kafa karıştırır hem de onların sınıfı kullananlar tarafından yanlışlıkla çağrılması sorunlara yol açabilir. (Örneğin eğer arabanın kaputu olmasaydı, yoldan geçen birileri ona zarar verebilirdi). Örneğin:

```
using System;

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

    class Sample
```

```
    {
        public void DoSomethingImportant()
        {
            //...
            Foo();
            //...
            Bar();
            //...
            Tar();
            //...
        }

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

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

        private void Tar()
        {
            //...
        }
    }
}
```

Bir sınıf dokümante edilirken onun her bölümünün dokümante edilmesine gerek yoktur. Yalnızca public (ve protected) bölümlerinin dokümante edilmesi yeterlidir. Çünkü zaten diğer elemanlara dışarıdan erişilemeyecektir.

Bir sınıf için iki bakış açısı vardır: Sınıfı yazanların bakış açısı ve kullananların bakış açısı. Sınıfı kullananlar sınıfın yalnızca public bölümünü bilseler yeterlidir. Sınıfı yazanların ise her bölümü bilmesi gerekir.

Değişkenlerin İsimlendirilmesi ve Harflendirilmesi

Değişkenleri isimlendirirken onların telaffuz edilebilir ve anlamlı olmasına özen göstermeliyiz. İsimlendirmede harflendirme (büyük harf-küçük harf kullanımı) de önemli bir unsurdur. Üç harflendirme biçimi programlamada yaygın olarak kullanılmaktadır:

1) Klasik C Tarzı: Bu stilde değişkenler küçük harflerle isimlendirilir. Birden fazla sözcükten oluşan değişkenlerin arasına alt tire getirilir. Örneğin:

```
number_of_students
max_val
minimum_cost
create_window
```

C#'ta bu tarz hiç kullanılmamaktadır.

2) Pascal Notasyonu (Pascal Casting): Bu harflendirme biçiminde her sözcüğün ilk harfi büyük diğer harfleri küçük yazılır. Örneğin:

```
NumberOfStudents
MaxVal
CreateWindow
```

3) Deve Notasyonu (Camel Casting): Bu biçimde ilk sözcüğün tüm harfleri küçük harflerle yazılır. Sonraki sözcüklerin yalnızca ilk harfleri büyük yazılır. Örneğin:

```
numberOfSectors
maxVal
createWindow
```

C#'ta gelenksel olarak isim alanı isimleri, sınıf isimleri ve sınıfların public elamanları Pascal notasyonu ile, yerel değişkenler, parametre değişkenleri, sınıfların protected ve private elemanları deve notasyonuyla harflendirilir. Örneğin Java'da sınıf isimleri Pascal tarzı diğer bütün isimler deve notasyonuyla harflendirilmektedir. Bir yazılımcının hangi ortamdaysa onun geleneğine uyması tavsiye edilmektedir.

Sınıfın public Olmayan Veri Elemanlarının Harflendirilmesi

Bazı programcılar (fakat hepsi değil) okunabilirliği sağlamak için sınıfın public olmayan veri elemanlarını özel bazı önekler kullanarak isimlendirmektedir. Böylece bir metotta bir değişkenin yerel mi olduğu yoksa sınıfın bir veri elemanı mı olduğu kolaylıkla anlaşılmaktadır. Public olmayan veri elemanlarını m_xxx ya da d_xxx biçiminde isimlendirmek yaygındır. Biz kursumuzda (ve sonraki kurslarda) sınıfın public olmayan veri elemanlarını m_xxx biçiminde isimlendireceğiz. Böylece örneğin:

```
public void Foo()
{
      //...
      m_maxCount = 10;
      //...
}
```

gibi bir koda bakan kişi m_maxCount değişkeninin sınıfın bir veri elemanı olduğunu anlar.

Veri Elemanlarının Gizlenmesi (Data Hiding) Prensibi

NYPT'nin diğer önemli prnesibi veri elemanlarının sınıfın private bölümüne yerleştirilerek dışarıdan gizlenmesidir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Date date = new Date(10, 12, 2012);

            date.Disp();
        }
    }

    class Date
    {
        private int m_day;
        private int m_month;
        private int m_year;

        public Date(int day, int month, int year)
        {
            m_day = day;
            m_month = month;
            m_year = year;
        }

        public void Disp()
        {
            Console.WriteLine("{0}/{1}/{2}", m_day, m_month, m_year);
        }
```

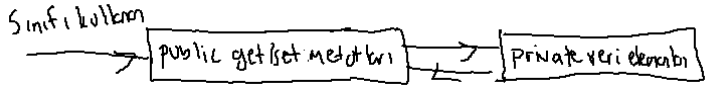
```
    }
}
```

Veri elemanları sınıfın iç işleyişine ilişkindir. Bu nedenle bunların gizlenmesinin aşağıda açıklayacağımız bazı faydaları vardır.

Sınıfın veri elemanları private bölüme yerleştirildiğinde artık onlara doğrudan erişilemez. Erişlmek istendiğinde public get ve set metotlarının kullanılması gerekir. Get metotları private veri elemanlarının değerlerini bize veren metotlardır, set metotları da onlara değer yerleştiren metotlardır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Date date = new Date();

            date.SetDay(10);
            date.SetMonth(12);
            date.SetYear(2012);

            Console.WriteLine("{0}/{1}/{2}", date.GetDay(), date.GetMonth(), date.GetYear());

        }
    }

    class Date
    {
        private int m_day;
        private int m_month;
        private int m_year;

        public int GetDay()
        {
            return m_day;
        }

        public void SetDay(int day)
        {
            m_day = day;
        }

        public int GetMonth()
        {
            return m_month;
        }

        public void SetMonth(int month)
        {
            m_month = month;
        }

        public int GetYear()
        {
            return m_year;
        }

        public void SetYear(int year)
        {
            m_year = year;
        }
    }
}
```

Böylece private veri elemanlarına doğrudan değil public metotlarla erişilmiş olur.



Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Complex z = new Complex();

            z.SetReal(10);
            z.SetImag(3);

            Console.WriteLine("{0}+{1}i", z.GetReal(), z.GetImag());
        }
    }

    class Complex
    {
        private double m_real;
        private double m_imag;

        public double GetReal()
        {
            return m_real;
        }

        public void SetReal(double real)
        {
            m_real = real;
        }

        public double GetImag()
        {
            return m_imag;
        }

        public void SetImag(double imag)
        {
            m_imag = imag;
        }

    }
}
```

Sınıfın veri elemanlarını private bölüme yerleştirerek onlara get ve set meotlarıyla erişmenin ne faydası vardır?

Sınıfın Veri Elemanlarının private Bölümde Gizlenmesinin Anlamı

1) Sınıfın veri elemanları iç işleyişe ilişkindir. Bu nedenle onların gizlenmesi olası değişiklerden daha önce yazılmış kodların etkilenmemesini sağlar. Deneyimler veri elemanlarının isim ve tür bakımından çok sık değiştirildiğini göstermektedir. Eğer biz onları public bölüme yerleştirirsek, onları herkes programında kullanabilir. Bu durumda onlarda değişiklik olduğunda eskiden yazılmış kodlar geçersiz kalır. Eğer biz onları private bölüme yerleştirip dışarıya kapatarak, erişimi public metotlarla yapmaya zorlarsak o veri elemanları değiştirildiğinde o metotların içini yeniden yazarak bu değişiklikten sınıfı kullananların etkilenmesini engelleyebiliriz. Bir sınıf için iki kod söz konusudur: Sınıfın kendi kodları ve sınıfı kullanan kodlar. Sınıfı kullanan kodların sınıfın veri elemanları değiştiğinde etkilenmemesi istenir. Örneğin yukarıdaki Date sınıfında biz bir değişiklik yaparak tarihi bir string nesnesinde tutmak isteyebiliriz. Bu durumda tek yapacağımız şey Get

ve Set metotlarının içini değiştirmektir:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Date date = new Date();

            date.SetDay(10);
            date.SetMonth(12);
            date.SetYear(2012);

            Console.WriteLine("{0}/{1}/{2}", date.GetDay(), date.GetMonth(), date.GetYear());
        }
    }

    class Date
    {
        private string m_date;      // "dd/mm/yyyy"

        public Date()
        {
            m_date = "01/01/1900";
        }

        public int GetDay()
        {
            return int.Parse(m_date.Substring(0, 2));
        }

        public void SetDay(int day)
        {
            m_date = m_date.Remove(0, 2).Insert(0, string.Format("{0:D2}", day));
        }

        public int GetMonth()
        {
            return int.Parse(m_date.Substring(3, 2));
        }

        public void SetMonth(int month)
        {
            m_date = m_date.Remove(3, 2).Insert(3, string.Format("{0:D2}", month));
        }

        public int GetYear()
        {
            return int.Parse(m_date.Substring(6, 4));
        }

        public void SetYear(int year)
        {
            m_date = m_date.Remove(6, 4).Insert(6, string.Format("{0:D2}", year));
        }
    }
}
```

Görüldüğü gibi bu değişiklikten Main metodundaki sınıfı kullanan kodar etkilenmemiştir. Burada dikkat edilmesi gereken nokta sınıfın veri elemanları değiştiğinde sınıfın Get ve Set metotlarının parametrik yapısının ve geri dönüş değerinin değiştirilmeden yalnızca onların içlerinin değiştirilmesidir.

2) Eğer veri elemanlarını public bölüme yerleştirirsek bunlara herkes erişir ve bunlara yanlışlıkla geçersiz değerler atayabilir. Örneğin Date sınıfında programcı yanlışlıkla m_day elemanına 60 gibi bir değer atayabilir. Oysa veri elemanlarını private bölüme yerleştirip onlara Set metotlarıyla değer atanmasına izin verirsek, bu Set

metotları içerisinde aralık kontrolü yapabiliriz. Yani değerlerin geçerliliğini sınayabiliriz (validation).

3) Sınıfın birbirleriyle ilşkili veri elemanları söz konusu olabilir. Yani bir veri elemanı değiştirildiğinde diğer bazı elemanların da değiştirilmesi gerekiyor olabilir. Eğet biz sınıfın veri elemanlarını public bölüme yerleştirirsek bu karmaşık ilişkiyi sınıfı kullanan kişilerin bilmesi gerekir.

4) Bazen veri elemanlarına değer atarken ve onların değerlerini alırken ilave bazı işlemlerin de yapıması gerekir. Örneğin, bazı log işlemleri yapılabilir. Bazı donanımsal ayarlamalar vs. yapılabilir. İşte veri elemanlarını public bölüme yerleştirirsek bunları sınıfı kullanan kişilerin sorumluluğuna bırakırız. Halbuki get ve set netotlarında bunlar gizilice yapılabilmektedir.

Fakat yine de bazı durumlarda (ancak %5'lik durumlar) yukarıdaki maddelerin hiçbiri geçerli olmayabilir. Bu durumda veri elemanlarını doğrudan public bölüme yerleştirebiliriz.

Sınıfların Property Elemanları

C#'ta sınıfın private elemanlarıyla ilişki kurmak için property elemanlar dile sokulmuştur. Property aslında get ve set metotlarının daha kolay kullanılan bir biçimidir. Java ve C++ dillerinde property yoktur, dolayısıyla o dillerde private elemanlara erişmek için doğrudan get ve set metotları kullanılır. Ancak C#'ta get ve set metotları yerine property'ler bu amaç için daha uygundur.

Property bildiriminin genel biçimi şöyledir:

```
[erişim belirleyicisi] <tür> <property ismi>
{
      get
      {
            //...
      }

      set
      {
            //...
      }
}
```

Bir property get ve set bölümlerinden oluşmaktadır. Bu bölümlerden yalnızca herhangi birisi bulunabilir ya da her iki bölüm de bulunabilir. Eğer property'nin hem get hem de set bölümü bulunuyorsa bunların yazılma sırası önemli değildir.

Property'ler veri elemanı gibi kullanılan metotlardır. Bir property ya içerisindeki değeri almak ya da içerisine değer yerleştirmek için kullanılır. Örneğin P bir property eleman olsun:

```
r.P = 100;                      // değer yerleştirme
result = r.P + 50;              // değer alma
Console.WriteLine(r.P);         // değer alma
```

Bir property değer alma amacıyla kullanılırsa property'nin get bölümü, değer yerleştirmek amacıyla kullanılırsa set bölümü çalıştırılır.

Property'nin set bölümü geri dönüş değeri void olan parametresi property türünden olan metot gibidir. Property'ye atanan değer set bölümüne parametre olarak aktarılır. set bölümünde value anahtar sözcüğü property'ye atanan değeri temsil eder. Biz set bölümünde tipik olarak private veri elemanına value değerini atarız.

Property'nin get bölümü parametresi olmayan, geri dönüş değeri property türünden olan bir metot gibidir. get

bölümünde tipik olarak private veri elemanıyla return edilmektedir. Propperty'nin içerisindeki değer alınmak istendiğinde get bölümü çalıştırılır, buradan geri döndürülen değer işleme sokulur. value anahtar sözcüğü get bölümünde kullanılamaz, yalnızca set bölümünde kullanılabilir.

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s = new Sample();
            int result;

            s.A = 10;
            result = s.A + 20;
            Console.WriteLine(result);
        }
    }

    class Sample
    {
        private int m_a;

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

            set
            {
                //...
                m_a = value;
            }
        }

    }
}
```

Property kullanımına diğer bir örnek:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Date date = new Date();

            date.Day = 10;
            date.Month = 12;
            date.Year = 2008;

            Console.WriteLine("{0}/{1}/{2}", date.Day, date.Month, date.Year);
        }
    }

    class Date
    {
        private int m_day;
```

```
        private int m_month;
        private int m_year;

        public int Day
        {
            get { return m_day; }
            set { m_day = value; }
        }

        public int Month
        {
            get { return m_month; }
            set { m_month = value; }
        }

        public int Year
        {
            get { return m_year; }
            set { m_year = value; }
        }
    }
}
```

Yalnızca get bölümünee sahip property'lere read-only property'ler, yalnızca set bölümüne sahip property'lere write-only property'ler, hem get hem de set bölümüne sahip property'lere read-write property'ler denir.

Anahtar Notlar: Visual Studio IDE'sinde imleci bir veri elemanının üzerine getirip bağlam menüsünden "Refactor/Encapsulate Field" seçilirse (Ctrl R + Ctrle E) IDE bize property'yi otomatik olarak yazmaktadır.

Static Property'ler

Sınıfın static veri elemanlarının da private bölüme yerleştirilmesi ve onlara property'lerle erişilmesi iyi bir tekniktir. Bunun için static property'ler kullanılır. Static property'lerin get ve set bölümleri static metotlar gibi değerlendirilmektedir. Yani static property'lere sınıf ismiyle erişilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample.Val = 10;
            Console.WriteLine(Sample.Val);
        }
    }

    class Sample
    {
        private static int m_val;

        public static int Val
        {
            get { return m_val; }
            set { m_val = value; }
        }
        //...
    }
}
```

Yapılar (Structures)

Yapılar sınıflara benzer veri yapılarıdır. (Örneğin Java'da yapı yoktur. C++'ta zaten yapı ile sınıf aynı anlamdadır.) Yapılar tamamen sınıflara benzer bir biçimde bildirilip kullanılarlar. Bildirimlerinde class yerine struct anahtar sözcüğü kullanılmaktadır. Yapı bildiriminin genel biçimi şöyledir:

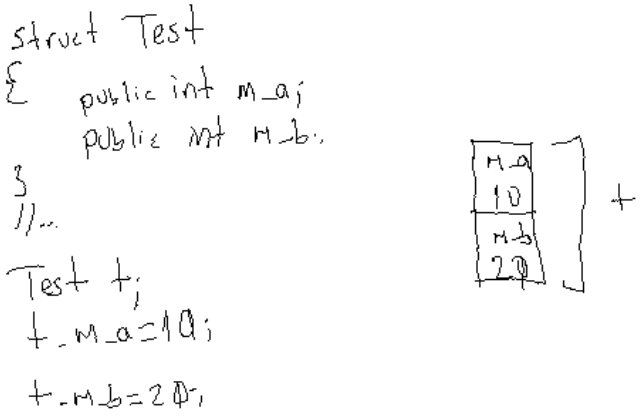
```
struct <isim>
{
      //...
}
```

Örneğin:

```
struct Test
{
      //...
}
```

Yapılar da veri elemanlarına ve metotlara sahip olabilir. Yine yapı elemanlarına da erişmek için nokta operatörü kullanılmaktadır. Yapılar türetmeye kapalı olduğu için protected ve protected internal elemanlara sahip olamazlar.

Yapılar kategori olarak değer türlerine (value types) ilişkindir. Yani bir yapı değişkeni bildirildiğinde o değişken bir adres tutmaz. Bizzat değerin kendisini tutar. Yapı değişkenleri bileşik nesnelerdir. Kendi içerisinde parçaları vardır. Yapı nesnelerinin parçalarına (elemanlarına) yine nokta operatörü ile erişilir. Yapı nesneleri için new operatörü ile tahsisat yapmaya gerek yoktur. Çünkü zaten yapı nesneleri bildirildiğinde onun parçaları için stack'te yer ayrılır:



Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Test t;

            t.m_a = 10;
            t.m_b = 20;

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

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

Bir yapının bütün veri elemanlarına değer atamadan o yapı değişkeni ile yapının bir metodu çağrılamaz ve yapı değişkeni bütünsel olarak kullanılamaz. Fakat değer atanan eleman kullanılabilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Test t;

            t.m_a = 10;
            Console.WriteLine(t.m_a);       // geçerli
            t.Foo();                // error! Yapının tüm elemanlarına değer atanmamış! */
        }
    }

   struct Test
   {
       public int m_a;
       public int m_b;

       public void Foo()
       {
           //...
       }
   }
}
```

Aynı türden iki yapı değişkeni birbirlerine atanabilir. Bu durumda yapının tüm elemanları diğer yapının karşı gelen elemanlarına atanacaktır.

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Test t;
            Test k;

            t.m_a = 10;
            t.m_b = 20;

            k = t;      // geçerli
            Console.WriteLine("{0}, {1}", k.m_a, k.m_b);
        }
    }

   struct Test
   {
       public int m_a;
       public int m_b;

       public void Foo()
       {
           //...
       }
   }
}
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Test t, k;

            t.m_a = 10;
            t.m_b = 20;

            k = t;

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

            k.m_a = 30;
            k.m_b = 40;

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

    struct Test
    {
        public int m_a;
        public int m_b;

        public void Foo()
        {
            //...
        }
    }
}
```

Yapılarda da veri elemanlarını private bölüme yerleştirip onlara public property'lerle erişmek iyi bir tekniktir. Örneğin:

```
struct Test
   {
       private int m_a;
       private int m_b;

       public int A
       {
           get { return m_a; }
           set { m_a = value; }
       }

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

Yapıların da başlangıç metotları bulunabilir. Yapılar için de new operatörü kullanılabilir. Fakat yapılar için new yapmakla sınıflar için new yapmak aynı anlama gelmez.

Bir yapı için new uygulandığında stack'te geçici bir yapı değişkeni yaratılır. O yapı değişkeni için başlangıç metodu çağrılır. Böylece new işleminden stack'te yaratılmış bir yapı nesnesi elde edilir. Biz onu başka bir yapı nesnesine atarız. Yaratılan bu geçici değişken onun kullanıldığı ifade bittiğinde otomatik yok edilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Test t;

            t = new Test(10, 20);
            Console.WriteLine("{0}, {1}", t.A, t.B);
        }
    }

   struct Test
   {
       private int m_a;
       private int m_b;

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

       public int A
       {
           get { return m_a; }
           set { m_a = value; }
       }

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

Yukarıdaki programda yapı için new işlemi şöyle yapılmıştır:

```
Test t;

t = new Test(10, 20);
```

Burada new işlemi ile geçici bir yapı nesnesi yaratılıp, onun için başlangıç metodu çağrılmış ve bu geçici nesne t'ye atanmıştır. Böylece geçici nesnenin tüm parçaları t'ye atanmış durumdadır. Atama işlemi bitince bu geçici nesne de yok edilecektir.

Bir yapı için başlangıç metodu yazarken yapının tüm elemanlarına başlangıç metodunda değer atanması zorunludur. Halbukli sınıflar için böyle bir zorunluluk yoktur. Peki neden?

Bilindiği gibi bir sınıf için new işlemi yaptığımızda new operatörü önce tüm veri elemanlarını sıfırlayıp sonra sınıfın başlangıç metodunu çağırır. Biz sınıfın başlangıç metodunda sınıfın o veri elemanına birşey atamasak bile o elemanda sıfır gözükecektir. Halbuki bir yapı için new yapıldığında new operatörü stack'te yarattığı geçici nesneyi sıfırlamamaktadır. Doğrudan başlangıç metodunu çağırmaktadır. İşte bu nedenle bizim ona değer atamamız gerekmektedir. Başlangıç metodundan çıkıldığında derleyici tüm elemanların içerisinde bir değerin bulunmasını ister.

Yapılar için default başlangıç metodu yazılamaz. Yapılar için her zaman default başlangıç metodunu derleyici yazar (yani biz herhangi bir başlangıç metodu yazsak da yazmasak da her zaman derleyici tarafından yazılmaktadır.) Derleyicinin yazdığı default başlangıç metodu da yapının tüm elemanlarını sıfırlamaktadır.

Yani başka bir deyişle tüm yapılar için default başlangıç metotları kullanıma hazır bir durumda bulunmaktadır. Bu default başlangıç metodu yapının tüm elemanlarını sıfırlar.

Yapılara Neden Gereksinim Duyulmaktadır?

Sınıf nesneleri heap'te yapı nesneleri stack'ye taratılmaktadır. Heap'te bir nesnensin yaratılması ve yok edilmesi göreli olarak çok yavaştır. Çok sayıda bileşik nesnenin kullanıldığı bir durumda bunların sınıf yerine yapı olarak ifade edilmesi daha avantajlıdır. Örneğin bir çizim yapılırken binlerce doğru çizilebilmektedir. Doğruyu temsil eden Point bir yapı olarak bildirilirse çok daha etkin bir çalışma söz konusu olur. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Graphics.DrawLine(new Point(10, 20), new Point(30, 40));
            //...
        }
    }

    class Graphics
    {
        public static void DrawLine(Point pt1, Point pt2)
        {
            pt1.Disp();
            pt2.Disp();
        }
    }

    struct Point
    {
        private int m_x;
        private int m_y;

        public Point(int x, int y)
        {
            m_x = x;
            m_y = y;
        }

        public void Disp()
        {
            Console.WriteLine("X = {0}, Y = {1}", m_x, m_y);
        }
    }
}
```

Burada Point nesneleri new ile stack'te yaratılır, parametre değişkenleri de stack'te yaratılmaktadır. Heap hiç devreye girmez ve aktarım çok hızlı olur. Yok edilme işlemleri de çok hızlıdır. Eğer Point bir sınıf olsaydı bunların yaratılması ve yok edilmesi çok daha yavaş olurdu.

Öte yandan eğer yapının çok elemanı varsa, atama sırasında bu kez göreli bir zaman kaybı oluşur. O halde "eğer az sayıda veri elemanı varsa ve bun tür nesneler çok kullanıyorsa" onların yapı ile temsil edilmesi, değilse sınıf ile temsil edilmesi uygun olur.

Ayrıca sınıfların yapılara göre önemli bir avantajı da vardır. Biz bir sınıf nesnesini referans yoluyla fonksiyona geçirerek fonksiyonun onun içerisine birşey yazmasını sağlayabiliriz. Halbuki yapılar için aynı dırım söz konusu değildir. Örneğin:

```
public static void Foo(Test t)
{
    t.a = 10;
```

```
    t.b = 20;
    //...
}
```

Burada eğer Test bir sınıfsa değerleri parametresiyle belirtilen adresteki nesneye yerleştiriyor durumdadır. Halbuki TEst bir yapıysa değer parametre değişkeni olan nesnenin içerisine yerleştirilecektir. Dolayısıyla bu durumda metot çağrısı bittiğinde çağırma işleminde kullanılan nesne bundan etkilenmeyecektir. Aslında C#'ta yapı nesnelerinin de ref ve out parametreleri yoluyla adres yoluyla metotlara aktarılması mümkündür. Bu konu ileride ele alınacaktır.

.NET'te Çok Kullanılan Bazı Yapılar

DateTime Yapısı

DateTime yapısı tarih ve zaman bilgisini tutan önemli bir yapıdır. Yapının başlangıç metotları tutulacak tarih ve zamanı bizden alır. Tipik başlangıç metotları şöyledir:

```
public DateTime(int year, int month,int day)
public DateTime( int year,int month,int day, int hour,int minute, int second)
```

Birinci başlangıç metodu ile nesne yaratıldığında yapının zaman kısmı sıfır olur, ikincisinde onu biz istediğimiz gibi veririz.

DateTime yapısının ToString isimli static olmayan metodu parametre almaz. Geri dönüş değeri olarak bize string verir. Bu metot ilgili tarih zamanı belirten ifadeyi bize yazı olarak vermektedir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            DateTime dt1 = new DateTime(2012, 10, 5);
            Console.WriteLine(dt1.ToString());

            DateTime dt2 = new DateTime(2012, 10, 5, 13, 45, 30);
            Console.WriteLine(dt2.ToString());

        }
    }
}
```

Anahtar Notlar: .NET'te pek çok sınıf ve yapıda aşağıdaki parametrik yapıya sahip static olmayan bir ToString metodu bulunmaktadır:

```
public string ToString()
```

Bu metotlar sınıfın veri elemanlarını temsil eden bir yazıyı bize verirler. Bunların verdikleri yazı bizi tam olarak tatmin etmeyebilir. Ancak bazen hızlı bir biçimde birşeyleri yazdırmak istediğimizde bunlardan faydalanabiliriz.

DateTime yapısının read-only Year, Month, Day, Hour, Minute, Second, MilliSecond isimli property'leri vardır. Yapının tuttuğu tarih ve zamanın bileşenleri bu property'lerden elde edilebilir.

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            DateTime dt = new DateTime(2012, 10, 5);
```

```
            Console.WriteLine("{0}/{1}/{2}", dt.Day, dt.Month, dt.Year);
        }
    }
}
```

DateTime yapısının static Now isimli property'si o anda bilgisayarın saatine bakarak bize o anki tarih ve zamanı DateTime olarak verir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            DateTime dt = DateTime.Now;

            Console.WriteLine(dt.ToString());
            Console.WriteLine("{0}:{1}:{2}", dt.Hour, dt.Minute, dt.Second);
        }
    }
}
```

Örneğin canlı bir saat şöyle edilebilir:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            while (!Console.KeyAvailable)
            {
                DateTime dt = DateTime.Now;
                Console.Write("{0:D2}:{1:D2}:{2:D2}\r", dt.Hour, dt.Minute, dt.Second);
            }
        }
    }
}
```

Ekrana saati daha az basma şöyle sağlanabilir:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            DateTime dt1, dt2;

            dt2 = DateTime.Now;
            while (!Console.KeyAvailable)
            {

                dt1 = DateTime.Now;
                if (dt1.Second != dt2.Second)
                    Console.Write("{0:D2}:{1:D2}:{2:D2}\r", dt1.Hour, dt1.Minute, dt1.Second);
                dt2 = dt1;
            }
        }
    }
}
```

DateTime yapısının static Today isimli property'si o günkü tarihi zaman bilgisi sıfır olacak biçimde verir.

DateTime yapısının AddDays, AddMonths, AddYears, AddHours, AddMinutes, AddSeconds ve AddMilliSeconds isimli metotları double parametre alır ilgiliyi toplamayı yaparak bize yeni DateTime nesnesi verir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            DateTime dt = DateTime.Now;
            DateTime result;

            result = dt.AddHours(3.5);
            Console.WriteLine(result.ToString());

        }
    }
}
```

DateTime yapısının karşılaştırma operatör metotları vardır. Böylece biz iki DateTime nesnesini karşılaştırma operatörleriyle karşılaştırabiliriz:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            DateTime dt1 = new DateTime(2016, 5, 16);
            DateTime dt2 = DateTime.Today;

            if (dt1 > dt2)
                Console.WriteLine("dt1 > dt2");
            else if (dt1 < dt2)
                Console.WriteLine("dt1 < dt2");
            else if (dt1 == dt2)
                Console.WriteLine("dt1 == dt2");
        }
    }
}
```

TimeSpan Yapısı

TimeSpan yapısı bir zaman aralığını ifade etmek için kullanılır. TimeSpan yapısın başlangıç metotlarıyla zaman aralığı oluşturulur. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            TimeSpan ts = new TimeSpan(3, 45, 34);
            Console.WriteLine(ts.ToString());
        }
    }
```

```
}
```

Yapının ToStirng metodu bize zaman aralğını yazısal olarak verir. TimeSpan yapısının Days, Hours, Minutes, Seconds ve MilliSeconds read-only property'leri bize zaman aralığının bileşenlerini verir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            TimeSpan ts = new TimeSpan(3, 45, 34);
            Console.WriteLine("{0}:{1}:{2}", ts.Hours, ts.Minutes, ts.Seconds);
        }
    }
}
```

TimeSpan yapısının double türden TotalDays, TotalHours, TotalMinutes, TotalSeconds ve TotalMilliseconds propertry'leri bize o zaman aralığını o cinsten verir. Örneğin 3 saat 45 dakika 34 saniye toplamda kaç saniyedir?

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            TimeSpan ts = new TimeSpan(3, 45, 34);
            Console.WriteLine(ts.TotalSeconds);
        }
    }
}
```

İki DateTime yapısı toplanamaz fakat çıkartılabilir. İki DateTime çıkartılırsa ürün olarak TimeSpan elde edilmektedir. Örneğin 17 Ağustos depreminden bu yana ne kadar zaman geçmiştir:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            DateTime dt = new DateTime(1999, 8, 17, 3, 2, 0);
            DateTime now = DateTime.Now;
            TimeSpan ts;

            ts = now - dt;
            Console.WriteLine(ts.ToString());
            Console.WriteLine(ts.TotalMinutes);
        }
    }
}
```

TimeSpan yapısının da karşılaştırma, toplama ve çıkartma operatör metotları vardır. Yazni bir iki TimeSpan yapısını toplayabiliriz, çıkartabiliriz ve karşılaştırma operatörleriyle karşılaştırabiliriz. Örneğin:

```
using System;

namespace CSD
{
    class App
```

```
    {
        public static void Main()
        {
            TimeSpan ts1 = new TimeSpan(3, 2, 21);
            TimeSpan ts2 = new TimeSpan(2, 59, 51);
            TimeSpan ts3;

            ts3 = ts1 + ts2;
            Console.WriteLine(ts3.ToString());
        }
    }
}
```

C#'ın Temel Türleri ve Yapılar

C#'ta int, long, double gibi türler de aslında birer yapı kabul edilmektedir. Örneğin int bir anahtar sözcüktür ve System.Int32 yapısını temsil eder. Diğer tür belirten anahtar sözcükler de aslında bazı yapıların kısa ismidir. Yani örneğin:

```
int a;
```

demekle,

```
System.Int32 a;
```

demek, ya da:

```
using System;
//...
Int32 a;
```

demek aynı anlamdadır. Farklı dillerde bu temel yapılara farklı anahtar sözcükler karşılık getirilmiş olabilir.

Enum Türleri ve Sabitleri

Bazen kıstlı sayıda seçenekten oluşan olgularla çalışabilmekteyiz. Örneğin ana renkler, haftanın günleri, aylar, yönler vs. gibi. Bu tür durumlarda enum'lar olmasaydı hangi türleri kullanabilirdik? Örneğin bir topu 4 yöden birinde hareket ettiren Move isimli bir metot yazacak olalım. Bu metodun parametresi hareket ettirilecek yönü belirtecek olsun. Parametre hangi türden olmalıdır?

İlk akla gelecek tür herhalde int olur. Her yöne bir sayı karşılık getiririz. Örneğin:


```
publi static void Move(int direction)
{
      switch (direction)
      {
            //...
      }
}
```

Metodu aşağıdaki gibi çağırabiliriz:

```
Move(3);         // Aşağıya git
//...
Move(1);         // Yukarıya gir
```

Bu tasarımın iki dezavantajı vardır: Birincisi sayılar yazılar kadar anlaşılır değildir. Yani okunabilirlik düşüktür. İkincisi ise biz yalnışlıkla farklı bir değeri metoda girersek bizi kimse uyarmaz.

Alternatif çözüm string olabilir:


```
publi static void Move(string direction)
{
      switch (direction)
      {
            //...
      }
}
```

Bu tasarımın da yine iki dezavantajı vardır: Birincisi yazılarla işlemler sayılarla işlemlerden aslında çok daha yavaştır. (Yani örneğin biz iki string'i == ile karşılaştırdığımızda aslında arka planda karşılaştırma bir döngüyle yapılmaktadır.) İkincisi ise metot çağrılırken yine yanlış değerlerin girilebilmesidir. (Örneğin “Up” yerine “up” girişi yaparsak kod yanlış çalışabilir.)

O halde bizim yazı gibi temsil edilen fakat aslında sayısal işlem gören, üstelik de yanlış değer girmemize olanak vermeyecek bir türe ihtiyacımız vardır. İşte enum'lar bunun için kullanılmaktadır.

enum bildiriminin genel biçimi şöyledir:

```
enum <isim>
{
      [enum sabit listesi]
}
```

enum sabitleri ',' atomu ile ayrılmaktadır.

Örneğin:

```
enum Direction
{
      Up, Right, Down, Left
}

enum Color
{
      Red, Green, Blue
}

enum Day
```

```
{
      Sunday, Monday, Teusday, Wednesday, Thurday, Friday, Saturday
}
```

Enum türlerine ilişkin değişken bildirebiliriz. Enum türleri kategori olarak değer türlerine ilişkindir. Yani enum türünden bir değişken bildirildiğinde o değişken değerin kendisini tutar bir adres tutmaz.

Bir enum sabitine (enumerator) enum ismi ve nokta operatörüyle erşilir. Örneğin Direction.Up, Direction.Left gibi.

Aynı türden iki enum birbirlerine atanabilir. Fakat farklı türden (yani farklı isimli) iki enum birbirlerine atanamaz. Örneğin:

```
Direction x, t;
//...
x = y;           // geçerli
```

Enum sabitleri ilgili enum türündendir. Örneğin Direction.Up ifadesi Direction türündendir. Day.Sunday ise Day türündendir. Bu durumda bir enum türünden değişkene bir enum sabiri atanabilir. Örneğin:

```
Direction d = Direction.Down;
```

Console sınıfının Write ve WriteLine metotlarıyla bir enum yazdırılmak istenirse bu metotlar o enum değerinin sabit yazısını yazdırır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Direction d;

            d = Direction.Right;
            Console.WriteLine(d);
        }
    }

    enum Direction
    {
        Up, Right, Down, Left
    }
}
```

Bir metodun parametre değişkeni bir enum türünden olabilir. Bu durumda biz o metodu aynı türden bir enum değeri ile çağırırız. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Move(Direction.Up);
            Move(Direction.Left);
        }

        public static void Move(Direction d)
```

```
        {
            Console.WriteLine(d);
        }
    }

    enum Direction
    {
        Up, Right, Down, Left
    }
}
```

enum sabitleri aslında birer tamsayı belirtmektedir. İlk enum sabitinin sayısal değeri sıfırdır. Sonraki her sabit öncekinden bir fazla değerdedir. Fakat bir enum sabitine = ile değer verilirse diğerleri onu izler. Örneğin:

```
enum Direction
{
    Up, Right = 3, Down, Left = 7
}
```

Burada Up = 0, Right = 3, Down = 4 ve Left = 7'dir. Farklı enum sabitlerinin aynı değerde olması da yasak değildir. Örneğin:

```
enum Direction
{
    Up, Right, Down = 5, Left = 1
}
```

Burada Up = 0, Right = 1, Down = 5 ve Left = 1'dir.

Her ne kadar enum türleri aslında birer tamsayı tutuyorsa da tamsayı türlerinden enum türlerine, enum türlerinden de tamsayı türlerine otomatik dönüştürme yoktur. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Direction d;
            int a;

            d = Direction.Right;        // geçerli
            d = 1;                      // error!
            a = d;                      // error!
        }
    }

    enum Direction
    {
        Up, Right, Down, Left
    }
}
```

Ancak temel türlerden enum türlerine, enum türlerinden de temel türlere tür dönüştürme operatörüyle dönüştürme yapılabilir. Örneğin:

```
using System;

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

```
            Direction d;
            int a;

            d = (Direction)1;
            Console.WriteLine(d);
            a = (int)d;
            Console.WriteLine(a);
        }
    }

    enum Direction
    {
        Up, Right, Down, Left
    }
}
```

Bir enum değişkenine atanan değerin o enum'un sabitlerinden biriyle desteklenmesi gerekmemektedir. Bu durumda Console sınmıfının Write ve WriteLine metotları sabitin değerini yazdırır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Direction d;

            d = (Direction)1500;
            Console.WriteLine(d);
        }
    }

    enum Direction
    {
        Up, Right, Down, Left
    }
}
```

Her enum türünün ilişkin olduğu bir tamsayı türü (underlying integer type) vardır. Enum türünün ilişkin olduğu tamsayı türü enum isminden sonra ':' atomu ile belitilir. Örneğin:

```
enum Direction : byte
{
    Up, Right, Down, Left
}
```

Enum sabitleri onların ilişkin olduğu tamsayı türlerinin sınırları dışında değer alamaz. Örneğin:

```
enum Direction : sbyte            // error!
{
    Up = -3, Right , Down = 127, Left
}
```

Burada Left = 128 olamayacağı için bildirim error ile sonuçlanır..

Eğer bildirimde enum türünün ilişkin olduğu tamsayı türü belirtilmezse default int kabul edilmektedir. Yani:

```
enum Direction
{
       Up, Right, Down, Left
}
```

ile

```
enum Direction : int
{
	Up, Right, Down, Left
}
```

aynı anlamdadır.

Pekiyi, enum türünün ilişkin olduğu tamsayı türü ne anlam ifade eder? Enum türü arka planda ilişkin olunan tamsayı türü gibi davranmaktadır. Örneğin bir enum türünden değişkenin kapladığı alan o enum türünün ilişkin olduğu tamsayı türü kadardır. Aynı zamanda tür dönüştürme işlemi yapılırken sanki o enum türünün ilişkin olduğu tamsayı türüne dönüştürme yapılıyormuş gibi kurallar uygulanır. Örneğin:

```
enum Direction : short
{
	Up, Right, Down, Left
}

//...

int a = 1234567;
Direction d;

d = (Direction) a;
```

Burada int türünden short türüne dönüştürme yapılıyormuş gibi işlemler yürütülüt.

enum Türleriyle İşlemler

Aynı türden iki enum karşılaştırma işlemine sokulabilir. Bu durumda onların içerisindeki değerler karşılaştırılmakatdır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Direction d1 = Direction.Right;
            Direction d2 = Direction.Left;

            if (d1 > d2)
                Console.WriteLine("d1 > d2");
            else if (d1 < d2)
                Console.WriteLine("d1 < d2");
            else if (d1 == d2)
                Console.WriteLine("d1 == d2");
        }
    }

    enum Direction
    {
        Up, Right , Down, Left
    }
}
```

switch parantezi içerisinde bir enum türünden değer olabilir ve case ifadeleri enum sabitlerinden oluşabilir. Örneğin:

```
using System;

namespace CSD
{
    class App
```

```
    {
        public static void Main()
        {
            Move(Direction.Right);
            //...
            Move(Direction.Left);
            //...
        }

        public static void Move(Direction d)
        {
            switch (d)
            {
                case Direction.Up:
                    Console.WriteLine("Move Up");
                    break;
                case Direction.Right:
                    Console.WriteLine("Move Right");
                    break;
                case Direction.Down:
                    Console.WriteLine("Move Down");
                    break;
                case Direction.Left:
                    Console.WriteLine("Move Left");
                    break;
            }
        }
    }

    enum Direction
    {
        Up, Right , Down, Left
    }
}
```

Örneğin Console sınıfının ReadKey metodu bize ConsoleKeyInfo yapısı türünden bir değer verir. O yapının da ConsoleKey property'si Key isimli property'si ConsoleKey isimli enum türündendir. Bu sayede biz basılan bütün tuşları tespit edebiliriz. Örneğin:

Bir yazıyı enum türünden değer dönüştürmek için aşağıdaki kalıp kullanılabilir:

```
(E) Enum.Parse(typeof(E), “Enum Sabit Yazısı”)
```

Burada E ilgili enum türünü belirtiyor. typeof operatörğ ve enum Sınıfı ileride ele alınacaktır.

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Direction d = Direction.Right;

            d = (Direction)Enum.Parse(typeof(Direction), Console.ReadLine());
            Console.WriteLine(d);
        }
    }

    enum Direction
    {
        Up, Right, Down, Left
    }
}
```

Örneğin:

Aşağıdaki anlatımlarda E bir enum türünü e bu enum türünden bir değeri, I bu enum türünün ilişkin olduğu tamsayı türünü, i de I türünden ya da I türüne otomatik dönüştürülebilen bir tür türünden değeri temsil edecektir.

1) Bir enumn türüyle bir tamsayı türü toplanabilir. Yani e + i ya da i + e işlemi geçerlidir. Bu işlemin eşdeğeri şöyledir:

```
(E) ((I)e + i)
```

Yani biz bir enum ile bir tamsayıyı topladığımızda o enum'un tamsayı değeri ile o tamsayı toplanır, değeri bu olan enum elde edilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Fruit f = Fruit.WaterMellon;
            Fruit result;

            result = f + 2;
            Console.WriteLine(result);      // Cherry
        }
    }

    enum Fruit
    {
        Apple, WaterMellon, Banana, Cherry, Strawberry
    }
}
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Day day = Day.Monday;
            Day result;

            result = day + 2;
            Console.WriteLine(result);      // Wednesday
        }
    }

    enum Day
    {
        Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
    }
}
```

Örneğin:

```
using System;

namespace CSD
```

```
{
    class App
    {
        public static void Main()
        {
            Day day = Day.Monday;
            Day result;

            result = day + 2;
            Console.WriteLine(result);      // Tuesday

            result = day + 1;
            Console.WriteLine(result);      // 3
        }
    }

    enum Day
    {
        Sunday = 0, Monday = 2, Tuesday = 4, Wednesday = 6, Thursday = 8, Friday = 10, Saturday = 12
    }
}
```

2) Bir enum değerinden bir tamsayı çıkartılabilir ancak bir tamsayıdan bir enum çıkartılamaz. Yani e - i işlemi geçerlidir, i - e işlemi geçerli değildir. e - i işleminin eşdeğeri şöyledir:

```
(E) ((I)e -  i)
```

Yani biz bir enum'dan bir tamsayıyı çıkarttığımızda o enum'un tamsayı değerinden o tamsayı çıkartılır, değeri bu olan enum elde edilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Fruit f = Fruit.Cherry;
            Fruit result;

            result = f - 2;
            Console.WriteLine(result);      // WaterMellon
        }
    }

    enum Fruit
    {
        Apple, WaterMellon, Banana, Cherry, Strawberry
    }
}
```

3) Aynı türden iki enum birbirlerinden çıkartılabilir fakat iki enum toplanamaz. Aynı türden iki enum birbirlerinden çıkartılırsa sonuç o enum türünün ilişkin olduğu tamsayı türünden olur. Yani e1 ve e2 E türünden olmak üzere e1 - e1 ifadesinin eşdeğeri:

```
(I)e1 - (I)e2
```

biçimindedir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
```

```
        public static void Main()
        {
            Fruit f1 = Fruit.Cherry;
            Fruit f2 = Fruit.WaterMellon;
            int result;

            result = f1 - f2;
            Console.WriteLine(result);      // 2
        }
    }

    enum Fruit
    {
        Apple, WaterMellon, Banana, Cherry, Strawberry
    }
}
```

DateTime yapısının DayOfWeek isimli property elemanı DayOfWeek enum türündendir. O tarihin hangi güne karşılık geldiğini bize vermektedir. DayOfWeek isimli enum haftanın günlerini belirtir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            DateTime dt = new DateTime(1920, 4, 23);

            Console.WriteLine(dt.DayOfWeek);
            Console.WriteLine(DateTime.Today.DayOfWeek);
            Console.WriteLine(new DateTime(1453, 5, 29).DayOfWeek);
        }
    }
}
```

Bir enum türünün içerisindeki tüm enum sabitlerinin yazıları bir string dizisi biçiminde aşağıdaki gibi elde edilebilir:

```
string[] names;

names = Enum.GetNames(typeof(E));
```

Burada E ilgili enum türünü temsil etmektedir. Örneğin:

```
using System;

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

            names = Enum.GetNames(typeof(Fruit));
            foreach (string name in names)
                Console.WriteLine(name);
        }
    }

    enum Fruit
    {
        Cherry, Strawberry, Banana, Blueberry, Raspberry, WaterMelon
    }
}
```

Örneğin:

```
using System;

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

            names = Enum.GetNames(typeof(DayOfWeek));
            foreach (string name in names)
                Console.WriteLine(name);
        }
    }
}
```

Bir enum türünün tüm elemanlarını bir enum dizisi olarak aşağıdaki gibi elde edilebilir:

```
(E[]) Enum.GetValues(typeof(E))
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Fruit[] fruits;

            fruits = (Fruit[]) Enum.GetValues(typeof(Fruit));
            foreach (Fruit fruit in fruits)
                Console.WriteLine("{0}: {1}", fruit, (int)fruit);
        }
    }

    enum Fruit
    {
        Cherry, Strawberry, Banana, Blueberry, Raspberry, WaterMelon
    }
}
```

Anahtar Notlar: UML (Unified Modeling Language) özellikle nesne yönelimli yazılım projelerinin modellenmesi ve planlanması için kullanılan diyagramatik bir dildir. Booch, Jacobson ve Rumbaugh isimli kişiler daha önce yaptıkları çalışmaları birleştirdiler. UML diyagramlardan oluşmaktadır. Bu diyagramları çizmenin ve anlamlandırmanın kuralları vardır. Diyagramlar projeye farklı açılardan bakıldığındaki durumu betimlemektedir. UML diyagramlarından biri de Sınıf Diyagramlarıdır. Bu diyagramlarda proje içerisindeki sınıflar ve onların arasındaki ilişkiler betimlenir.

Sınıflar Arasındaki İlişkiler

Daha önceden belirtildiği gibi bir proje nesne yönelimli olarak tasarlanacaksa önce proje içerisindeki kavramlar sınıflarla temsil edilir. Daha sonra o sınıflar türünden nesneler yaratılarak gerçek varlıklar elde edilir. Sınıflar arasında da birtakım ilişkiler söz konusu olabilmektedir. Örneğin hastane otomasyonunda Doktor sınıfı ile Hastane sınıfı, Doktor sınıfı ile Hasta sınıfı arasında ilşkiler vardır.

Sınıflar arasında dört ilişki biçimi tanımlanabilmektedir:

1) **İçerme İlişkisi (Composition):** Bir sınıf türünden nesne başka bir sınıf türünden nesnenin bir parçasını

oluşturuyorsa bu iki sınıf arasında içerme ilişkisi vardır. Örneğin Araba ile Motor sınıfları arasında, İnsan ile Karaciğer sınıfları arasında içerme ilişkisi vardır. İçerme ilişkisi için iki koşulun sağlanması gerekir:

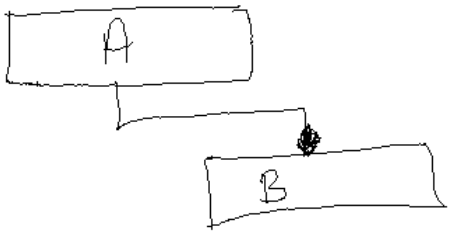
1) İçerilen nesne tek bir nesne tarafından içerilmelidir.

2) İçeren nesneyle içerilen nesnenin ömürleri yaklaşık aynı olmalıdır.

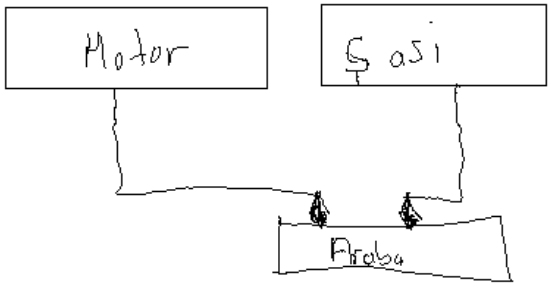
Tabi bu ölçütler tipik durumlar için düşünülmelidir. Aksi taktide biz doğadaki hiçbir şeyi tam olarak modelleyemeyiz. Örneğin İnsan öldüğünde karaciğeri başka bir insana takılabilmektedir. Fakat bu durum tasarım konusu da dikkate alınarak gözardı edilebilir.

Örneğin Oda ile Duvar sınıfları arasında içerme ilişkisi yoktur. Çünkü her ne kadar bunların ömürleri aynı ise de o duvar aynı zamanda yandaki odanın da duvarıdır. Satranç tahtası ile tahtanın kareleri arasında içerme ilişkisi vardır. Fakat satranç taşları ile kareler arasındaki ilişki içerme ilişkisi değildir. Saat ile akrep, yelkovan arasında içerme ilişkisi vardır. Fakat bilgisayar ile fare sınıfları arasındaki ilişki içerme ilişkisi değildir. Benzer biçimde örneğin bir diyalog penceresi ile onun üzerindeki düğmeler arasında içerme ilişkisi vardır.

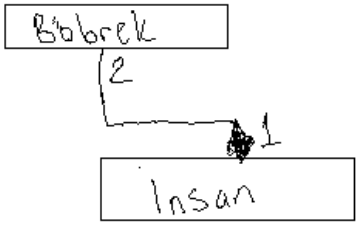
UML Sınıf diyagramlarında içerme ilişkisi içeren sınıf tarafında içi dolu bir baklavacıkla gösterilmektedir. Örneğin:



Burada B sınıfı A sınıfını içermektedir. Örneğin:



İçerme ilişkisi bire-bir olabileceği gibi bire-n de olabilir. Örneğin:



İçerme ilişkisine İngilizce "has a" ilişkisi de denilmektedir.

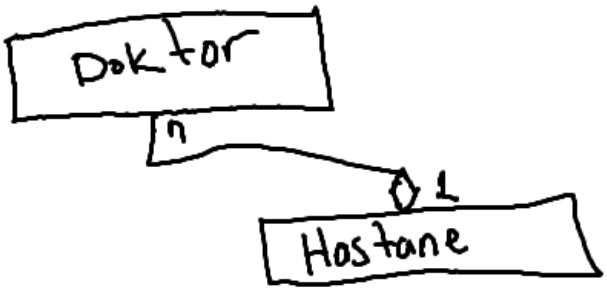
İçerme ilişkisi C#'ta şöyle oluşturulur: İçeren sınıfın içerilen sınıf türünden private bir referans veri elemanı olur. Bu private eleman dışarıya property ile verilmez. Bunun yaratımı da içeren sınıfın başlangıç metodunda yapılır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Automobile a = new Automobile();
            //...
        }
    }

    class Engine
    {
        //...
    }

    class Automobile
    {
        private Engine m_engine;
        //...
        public Automobile()
        {
            m_engine = new Engine();
            //...
        }
        //...
    }
}
```

2) Birleşme İlişkisi (Aggregation): Birleşme ilişkisinde bir sınıf nesnesi başka türden bir sınıf nesnesini bünyesine katarak kullanmaktadır. Fakat kullanan nesneyle kullanılan nesnenin ömürleri aynı olmak zorunda değildir. Kullanılan nesne başka nesneler tarafından da kullanılıyor olabilir. Örneğin, Araba sınıfıyla Tekerlek sınıfı arasında, Bilgisayar sınıfı ile Fare sınıfı arasında, Oda sınıfıyla Duvar sınıfı arasında, Ağaç sınıfıyla Yaprak sınıfı arasında, Hastane sınıfıylaDoktor sınıfı arasında böyle bir ilişki vardır. İçerme ilişkisine uymayan pek çok olgu birleşme ilişkisine uymaktadır.

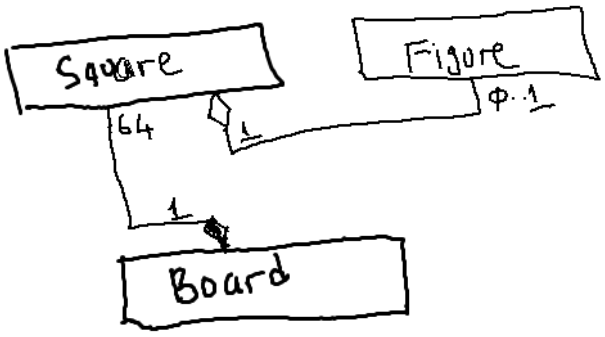
Birleşme ilişkisi UML sınıf diyagramlarında kullanan sınıf tarafında içi boş bir baklavacık (diamond) ile gösterilir. Örneğin:

Örneğin:



Birleşme ilişkisine İngilizce "Holds a" ilişkisi de denilmektedir.

Örneğin bir satranç tahtasını modellemeye çalışalım. Tahta Board sınıfıyla temsil edilsin. Tahta üzerindeki kareler Square sınıfı ile temsil edilsin. Board sınıfı ile Square sınıfı arasında içerme ilişkisi vardır. Tahta üzerindeki taşlar da Figure sınıfıyla temsil ediliyor olsun. Bu durumda Square sınıfı ile Figure sınıfı arasında da birleşme ilişkisi söz konusudur. UML sınıf diyagramı şöyle oluşturulabilir:



C#'ta birleşme ilişkisi şöyle oluşturulabilir: Kullanan sınıf içerisinde kullanılan sınıf türünden bir private referans veri elemanı bildirilir. Bu eleman için bir property oluşturulur. Böylece bu eleman dışarıdan get ve set edilebilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Mouse mouse = new Mouse();
            //...
            Computer myComputer = new Computer();
            myComputer.Mouse = mouse;
            //..
            myComputer.Mouse = null;
            Computer yourComputer = new Computer();
            yourComputer.Mouse = mouse;
            //...
        }
    }

    class Mouse
    {
        //...
    }

    class Computer
    {
        private Mouse m_mouse;
        //...
        public Computer()
        {
            //...
        }

        public Mouse Mouse
        {
            get { return m_mouse; }
            set { m_mouse = value; }
        }
        //...
    }
}
```

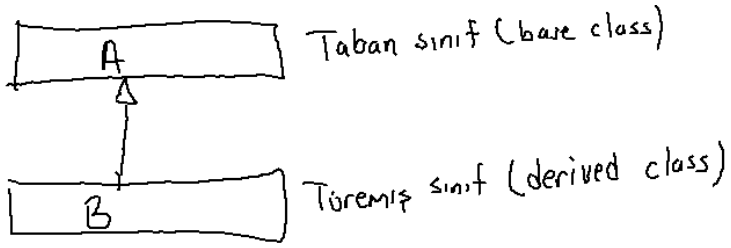
Örneğin bir satranç tahtası kabaca aşağıdaki gibi oluşturulabilir:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Board board = new Board();
            //...
        }
    }

    enum FigureType
    {
        King, Queen, Rook, Bishop, Knight, Pawn
    }

    enum Color
    {
        White, Black
    }

    class Figure
    {
        private FigureType m_figureType;
        private Color m_color;

        public Figure(FigureType figureType, Color color)
        {
            m_figureType = figureType;
            m_color = color;
        }

        public FigureType FigureType
        {
            get { return m_figureType;  }
            set { m_figureType = value; }
        }

        public Color Color
        {
            get { return m_color; }
            set { m_color = value; }
        }
    }

    class Square
    {
        private Figure m_figure;

        //...
        public Square()
        {
            //...
        }

        public Figure Figure
        {
            get { return m_figure; }
            set { m_figure = value; }
        }

        //...
    }

    class Board
```

```
    {
        private Square[,] m_squares;
        //...
        public Board()
        {
            m_squares = new Square[8, 8];

            for (int i = 0; i < 8; ++i)
                for (int k = 0; k < 8; ++k)
                    m_squares[i, k] = new Square();

            // Taşlar yerleştiriliyor
            m_squares[0, 0].Figure = new Figure(FigureType.Rook, Color.Black);
            m_squares[0, 1].Figure = new Figure(FigureType.Knight, Color.Black);
            //...
        }
        //...
    }
}
```

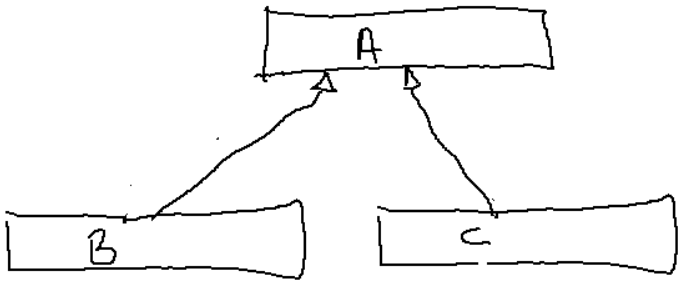
3) Kalıtım (Türetme) İlişkisi (Inheritance): Türetme mevcut bir sınıfa ona dokunmadan ekleme yapmak anlamına gelmektedir. Elimizde bir A sınıfı bulunuyor olsun. Biz buna birtakım elemanlar eklemek isteyelim. Fakat A'nın kaynak kodu elimizde olmayabilir ya da onu bozmak istemeyebiliriz. Bu durumda A sınıfından bir B sınıfı türetiriz. Eklemeleri B'ye yaparız. Böylece B sınıfı hem A sınıfı gibi kullanılır hem de fazlalıklara sahiptir.

Türetme işleminde işlevini genişletmek istediğimiz asıl sınıfa taban sınıf (base class) denilmektedir. Ondan türettiğimiz yani eklemeleri yaptığımız sınıfa da türemiş sınıf (derived class) denir. UML sınıf diyagramlarında türetme ilişkisi türemiş sınıftan taban sınıfa çekilen içi boş bir okla belirtilmektedir. Örneğin:



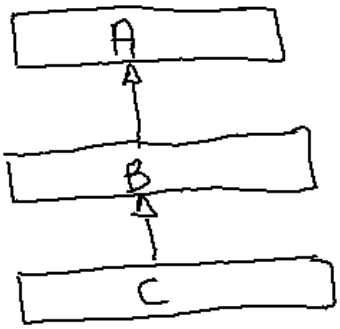
Burada B sınıfı hem A sınıfı gibi davranır hem de fazlalıkları vardır. Türetme ilişkisine İngilizce "is a" ilişkisi denilmektedir. (B bir çeşit A'dır fakat fazlalıkları da vardır.)

Bir sınıf birden fazla sınıfın taban sınıfı durumunda olabilir. Örneğin:



Burada B ile C arasında bir ilişki yoktur. B de C de A'dan türetilmiştir. Yani B sınıfı türünden bir nesne hem B gibi hem de A gibi kullanılabilir. C sınıfı türünden bir nesne de hem C gibi hem de A gibi kullanılabilir. Ancak B ile C arasında böyle bir ilişki yoktur.

Türemiş bir sınıftan yeniden türetme yapılabilir. Örneğin:



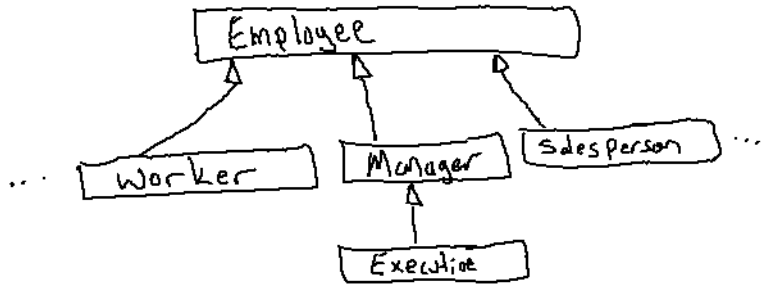
Burada C sınıfı hem B gibi hem de A gibi kullanılabilir. Ancak fazlalıkları da vardır.

Türetmeye çeşitli örnekler verebiliriz:

- Örneğin tüm taşıtların ortak birtakım özellikleri Taşıt sınıfında toplanabilir (plakası, trafiğe çıkış tarihi, motor gücü vs.). Bundan Otomobil, Kamyon, Motosiklet gibi sınıflar türetilebilir. Otomobil de bir taşıttır (is a ilişkisi), kamyon da, motosiklet de bir taşıttır. Fakat kamyonda olan özellikler otomobilde olmayabilir:

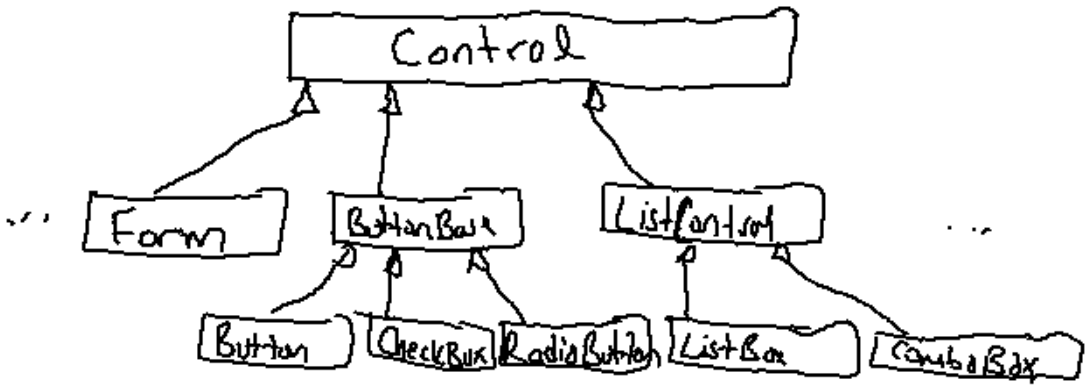
- Satranç taşlarının ortak özelliklerini Taş sınıfında toplayıp ondan Şahi Vezir, Kale, Fil, At ve Piyon sınıflarını türetebiliriz. Örneğin:

- Bir iş yerinde çalışanları sınıflarla temsil edebiliriz. Tüm çalışanların ortak özellikleri Employee isimli bir sınıfta toplanabilir. İşçiler bu sınıftan türetilmiş Worker sınıfıyla, yöneticiler Manager sınıfıyla temsil edilebilir. Üst düzey yöneticiler de bir çeşit yöneticidir. Bunlar da Executive isimli sınıfla temsil edilebilirler. Bu durumda Executive sınıfının da Manager sınıfından türetilmesi uygun olur:



Türetmeye neden gereksinim duyulmaktadır? Programlamadaki temel prensiplerden biri kod ve veri tekrarını engellemektir. Örneğin bir kod parçası aynı programda ikinci kez kopyala-yapıştır yapılmamalıdır. Bunu engellemek için en normal yöntem o kod parçasını bir metoda yerleştirmek ve o metodu çağırmaktır. Yani

programımız içerisinde aynı kod parçalarının farklı yerde bulunması durumundan şüphelenmeliyiz ve bunu gidermeye çalışmalıyız. Böylece hem programımız daha az yer kaplar hale gelir, hem daha algılanabilir olur hem de hataların analiz edilmesi daha kolaylaşır. İşte NYPT'de iki sınıfın içerisinde ortak elemanlar varsa (veri elemanları, property'ler ve metotlar) bunlar ortak bir taban sınıfta toplanmalı ve ondan türetme yapılarak bu iki sınıf oluşturulmalıdır. Örneğin GUI uygulamalarında ekranda bağımsız olarak kontrol edilebilen dikdörtgensel alanlara pencere (window) denilmektedir. Düğmeler, edit alanları, listeleme kutuları, ana pencereler hep birer penceredir. .NET Form kütüphanesinde tüm pencerelerin ortak özellikleri taban bir Control sınıfında toplanmıştır. Diğer sınıflar bundan türetilmiştir:



Button, CheckBox ve RadioButton pencerelerinin de birtakım ortak özellikleri vardır. Bu özellikler de ButtonBase sınıfında toplanmıştır. Benzer biçimde ListBox ve ComboBox sınıflarının ortak elemanları da ListControl sınıfında toplanmış durumdadır.

Bir türetme şemasında yukarıya çıkıldıkça genelleşme, aşağıya inildikçe özelleşme oluşur.

Bir sınıfın birden fazla taban sınıfı olması durumu ilginç ve özel bir durumdur. Buna çoklu türetme (multiple inheritance) denilmektedir. C# ve Java'da çoklu türetme özelliği yoktur. Dolayısıyla bu dillerde bir sınıfın tek bir taban sınıfı olabilir. Fakat C++'ta çoklu türetme vardır. Örneğin hava taşıtları bir sınıfla, deniz taşıtları başka bir sınıfla temsil edilebilir. Aircraft sınıfı bunlardan çoklu türetilebilir:

4) Çağrışım İlişkisi (Association): Bu ilişki biçiminde bir sınıf bir sınıfı bünyesine katarak değil, yüzeysel biçimde, bir ya da birkaç metodunda kullanıyor durumdadır. Örneğin Taksi ile Müşteri arasında ciddi bir ilişki yoktur. Taksi müşteriyi alır ve bir yere bırakır. Halbuki Taksi ile şoförü arasında önemli bir ilişki vardır. İşte Taksi ile Müşteri ilişkisi çağrışım ilişkisi iken, Taksi ile Şoför arasındaki ilişki birleşme (aggregation) ilişkisidir. Benzer biçimde Hastane sınıfı ReklamŞirketi sınıfını yalnızca reklam yaparken kullanmaktadır. Bunların arasında da çağrışım ilişkisi vardır. Çağrışım ilişikisi UML sınıf diyagramlarında kullanan sınıftan kullanılan sınıfa çekilen ince bir okla gösterilir. Örneğin:

C#'ta çağrışım ilişkisi genellikle şöyle oluşturulur: Metodun parametresi kullanılan sınıf türünden bir referans

olur, metot da o türden referansla çağrılır. Örneğin:

```
class Hastane
{
      //...
      public void ReklamYap(ReklamŞiketi rş)
      {
            //...
      }
      //...
}
```

Çağrışım ilişkisinde kullanılan nesnenin referansının kullanan sınıfın içerisinde saklanmadığına dikkat ediniz.

C#'ta Türetme İşlemleri

C#'ta türetme işleminin genel biçimi şöyledir:

```
class <türemiş sınıf ismi> : <taban sınıf ismi>
{
      //...
}
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            A a = new A();
            B b = new B();

            a.Foo();

            b.Foo();
            b.Bar();
        }
    }

    class A
    {
        //...
        public void Foo()
        {
            Console.WriteLine("A.Foo");
        }
    }

    class B : A
    {
        public void Bar()
        {
            Console.WriteLine("B.Bar");
        }
    }
}
```

Türemiş sınıftan yeniden sınıf türetilebilir:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            C c = new C();

            c.Foo();
            c.Bar();
            c.Tar();
        }
    }

    class A
    {
        //...
        public void Foo()
        {
            Console.WriteLine("A.Foo");
        }
    }

    class B : A
    {
        public void Bar()
        {
            Console.WriteLine("B.Bar");
        }
    }

    class C : B
    {
        public void Tar()
        {
            Console.WriteLine("C.Tar");
        }
    }
}
```

Örneğin:

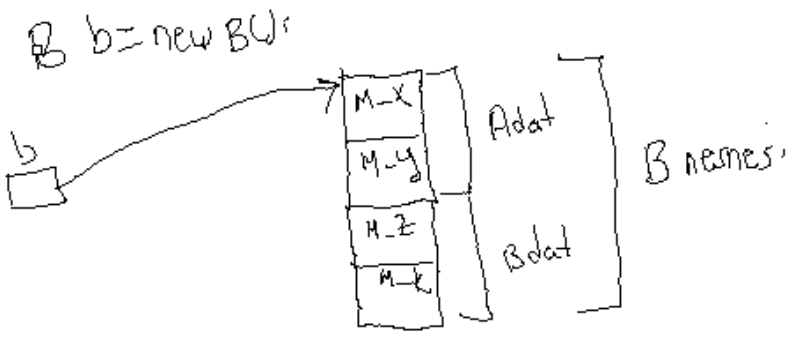
```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Executive e = new Executive();
            e.DispExecutive();
            e.DispManager();
            e.DispEmployee();

            Worker w = new Worker();
            w.DispWorker();
            w.DispEmployee();
            //...
        }
    }

    class Employee
    {
        public void DispEmployee()
        {
            Console.WriteLine("Employee info");
```

```
        }
        //...
    }

    class Worker : Employee
    {
        public void DispWorker()
        {
            Console.WriteLine("Worker info");
        }
        //...
    }

    class Manager : Employee
    {
        public void DispManager()
        {
            Console.WriteLine("Manager info");
        }
        //...
    }

    class Executive : Manager
    {
        public void DispExecutive()
        {
            Console.WriteLine("Executive info");
        }
    }
}
```

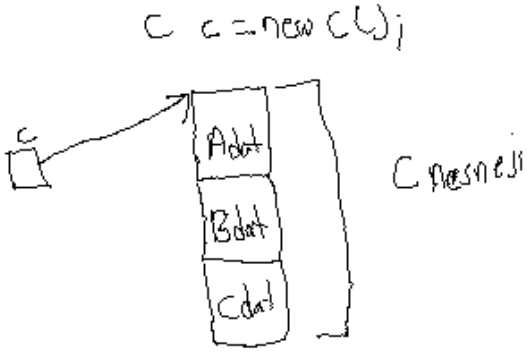
Türemiş Sınıflarda Veri Elemanlarının Organizasyonu

Türemiş sınıf nesnesi hem türemiş sınıfın kendi static olmayan veri elemanlarını hem de taban sınıfın static olmayan veri elemenalrını içermektedir. Türemiş sınıf nesnesi içerisinde taban sınıfın ve türemiş sınıfın veri elemanları ardışıl bir blok oluşturur. Düşük adreste (yani daha yukarıda) taban sınıfın veri elemanları vardır. Örneğin:

```
class A
{
    private int m_x;
    private int m_y;
    //...

}

class B : A
{
    private int m_z;
    private int m_k;
    //...
}
```



C sınıfı B sınıfından, B sınıfı da A sınıfından türetilmiş olsun:



Örnek:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new B();

            b.m_x = 10;
            b.m_y = 20;
            b.m_z = 30;
            b.m_k = 40;

            Console.WriteLine("{0}, {1}, {2}, {3}", b.m_x, b.m_y, b.m_z, b.m_k);
        }
    }

    class A
    {
        public int m_x;
        public int m_y;
        //...

    }

    class B : A
    {
        public int m_z;
        public int m_k;
        //...
    }
}
```

Türemiş sınıf erişim belirleyicisi ne olursa olsun taban sınıflarının tüm static olmayan veri elemanlarını içermektedir.

Türemiş Sınıflarda Erişim Kuralları

Türemiş sınıflarda eirşim kuralları iki maddeyle özetlenebilir:

1) Dışardan (türemiş sınıf dışından) türemiş sınıf türünden bir referansla ya da türemiş sınıf ismiyle türemiş sınıfın ve taban sınıfın yalnızca public elemanlarına erişilebilir.

2) Türemiş sınıf metotları içerisinde (genel olarak türemiş sınıfın içerisinde) taban sınıfın public ve protected elemanları doğrudan türemiş sınıfın elemanlarıymış gibi kullanılabilir. Ancak taban sınıfın private bölümüne türemiş sınıf tarafından erişilemez.

```
class A
```

```
{
    public int m_x;
    protected int m_y;
    private int m_z;

    public int Z
    {
        get { return m_z; }
        set { m_z = value; }
    }
    //...

}

class B : A
{
    public void Foo()
    {
        m_x = 10;       // geçerli
        m_y = 20;       // geçerli
        z = 30;         // error!
        Z = 30;         // geçerli
    }
    //...
}
```

Bu işlem türetme şeması boyunca aynı biçimde geçerlidir. Yani türemiş bir sınıfın metotları içerisinde doğrudan tüm taban sınıfların public ve protected elemanları kullanılabilir.

protected Bölümün Anlamı

protected bölüm yalnızca sınıfın kendisi ve türemiş sınıflar tarafından erişilebilen bölümdür. protected bölümdeki elemanlara dışarıdan erişilemez. Sınıfın public bölümü en az korunan bölümüdür. Bunu protected bölüm izler. Nihayet private bölüm sınıfın en çok korunan bölümdür. Biz daha bir sınıf tasarlarken o sınıftan türetme yapılabileceğini kestirip bazı elemanları türemiş sınıfı yazanlar kolay erişsin diye protected bölüme yerleştirebiliriz. Sınıfın public ve protected bölümleri o sınıfı kullanacak kişiler için dokümante edilmelidir. private bölümün dokümante edilmesine gerek yoktur.

Bir elemanı protected bölüme yerleştirdiğimizde artık onu değiştirirsek türemiş sınıflar bundan etkilenecektir. Bu durumda protected bölüme veri elemanlarını yerleştirirken dikkat etmeliyiz. Bazı programcılar protected bölümü bu nedenle hiç kullanmazlar. Veri elemanlarının hepsini private bölüme yerleştirirler. Böylece türemiş sınıf da taban sınıfın elemanlarına hep property'lerle erişmek zorunda kalır.

Türemiş Sınıflarda Başlangıç Metotlarının Çağrılması

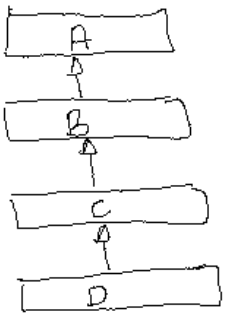
Türemiş sınıf türünden bir nesne new operatörüyle yaratıldığında türemiş sınıfn başlangıç metodu çağrılır. Türemiş sınıf taban sınıfın private bölümüne erişemediğine göre taban sınıfın private veri elemanlarına nasıl ilkdeğer verilecektir? İşte bu durum türemiş sınıfın başlangıç metodunun taban sınıfın başlangıç metodunu otomatik çağırması ile çözülmüştür. Türemiş sınıf türünden bir nesne new opeatörüyle yaratıldığında yalnızca türemiş sınıfın değil taban sınıfın da başlangıç metotları çağrılmaktadır. Böylece nesnenin taban sınıf kısmına bizzat taban sınıfın başlangıç metodu ilkdeğer vermiş olur. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new B();

            b.DispA();
            b.DispB();
```

```
        }
    }

    class A
    {
        private int m_a;

        public A()
        {
            Console.WriteLine("A default constructor");
            m_a = 10;
        }

        public void DispA()
        {
            Console.WriteLine(m_a);
        }
    }

    class B : A
    {
        private int m_b;

        public B()
        {
            Console.WriteLine("B default constructor");
            m_b = 20;
        }

        public void DispB()
        {
            Console.WriteLine(m_b);
        }

    }

}
```

Taban sınıfın başlangıç metodu türemiş sınıfın başlangıç metodunun ana bloğunun başında derleyicinin gizlice yerleştirdiği bir çağırma kodu yoluyla çağrılır. Yani böylelikle önce taban sınıfın sonra türemiş sının başlangıç metotları çalıştırılmış olur. Bir dizi türetme söz konusu olduğunda başlangıç metotlarının çağrılması sırası yukarıdan aşağıya doğrudur.

Anahtar Notlar: Bir sınıfın taban sınıfları (base classes) denildiğinde onun tüm taban sınıfları anlaşılır. Bir sınıfın doğrudan taban sınıfı (direct base class) denildiğinde onun hemen bir yukarısındaki taban sınıfı anlaşılır. Sınıfın dolaylı taban sınıfları (indirect base classes) sınıfın doğrudan taban sınıfının taban sınıflarıdır. Örneğin:



Burada D'nin tüm taban sınıfları A, B ce C'dir. Doğrudan taban sınıfı C, dolaylı taban sınıfları B ve A'dır.

Türemiş sınıf taban sınıfın yalnızca doğrudan taban sınıfının başlangıç metodunu çağırır.

Pekiyi taban sınıfın birden fazla başlangıç metodu varsa türemiş sınıfın başlangıç metodu taban sınıfın hangi başlangıç metodunu çağıracaktır? İşte bu durum : base sentaksıyla belirlenmektedir. Sınıfın başlangıç metodunun kapanış parametre parantezinden sonra, önce bir ':' atomu sonra da base anahtar sözcüğü yazılırsa bu sentaks türemiş sınıfın taban sınıfın hangi başlangıç metodunu çağıracağını belirtir. Örneğin:

```
class B : A
{
	public B(...) : base(...)
	{

	}
	//...
}
```

Taban sınıfın tüm başlangıç metotları aday metotlardır. En uygun başlangıç metodu overload resolution kurallarına göre seçilir. Tabii base sentaskı hiç kullanılmayabilir de. Bu durumda taban sınıfın default başlangıç metodu çağrılır. Yani,

```
public B(...)
{
	//...
}
```

ile,

```
public B(...) : base()
{
	//...
}
```

aynı anlamdadır. base sentaksı yalnızca başlangıç metotlarında kullanılan bir sentakstır. Başlangıç metodunun parametreleri base senktasına argüman olarak verilebilir. Örneğin:

```
public B(int a, int b) : base(a)
{
	m_b = b;
}
```

Faaliyet alanı bakımından base sentaksının sanki başlangıç metodunun ana bloğunun başına yerleştirildiği varsayılmaktadır. Yani base parantezi içerisinde biz parametre değişkenlerini, kendi sınıfımızın ve taban sınıfın elemanlarını kullanabiliriz. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b1 = new B();
            B b2 = new B(100, 200);

            Console.WriteLine("m_a = {0}, m_b = {1}", b1.ValA, b1.ValB);
            Console.WriteLine("m_a = {0}, m_b = {1}", b2.ValA, b2.ValB);
        }
    }

    class A
    {
        private int m_a;

        public A()
        {
            Console.WriteLine("A default constructor");
            m_a = 10;
        }

        public A(int a)
```

```
        {
            Console.WriteLine("A int constructor");
            m_a = a;
        }

        public int ValA
        {
            get { return m_a; }
        }

        //...
    }

    class B : A
    {
        private int m_b;

        public B()
        {
            Console.WriteLine("B constructor");
            m_b = 20;
        }
        public B(int a, int b) : base(a)
        {
            Console.WriteLine("B int, int constructor");
            m_b = b;
        }

        public int ValB
        {
            get { return m_b; }
        }

        //...
    }
}
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Executive m = new Executive("Salih Tan", 30000, "Üretim", "Asya");

            Console.WriteLine("{0}, {1}, {2}, {3}", m.Name, m.Salary, m.Department, m.Region);
        }
    }

    enum Shift
    {
        Morning, Noon, Evening
    }

    class Employee
    {
        private string m_name;
        private int m_salary;

        public Employee()
        { }

        public Employee(string name, int salary)
        {
            m_name = name;
```

```
            m_salary = salary;
        }

        public string Name
        {
            get { return m_name; }
            set { m_name = value; }
        }

        public int Salary
        {
            get { return m_salary; }
            set { m_salary = value; }
        }
    }

    class Worker : Employee
    {
        private Shift m_shift;

        public Worker()
        { }

        public Worker(string name, int salary, Shift shift) : base(name, salary)
        {
            m_shift = shift;
        }

        public Shift Shift
        {
            get { return m_shift; }
            set { m_shift = value;}
        }
    }

    class Manager : Employee
    {
        private string m_department;

        public Manager()
        {}

        public Manager(string name, int salary, string department) : base(name, salary)
        {
            m_department = department;
        }

        public string Department
        {
            get { return m_department;}
            set {m_department = value;}
        }
    }

    class Executive : Manager
    {
        private string m_region;

        public Executive()
        { }

        public Executive(string name, int salary, string department, string region) : base(name, salary,
department)
        {
            Region = region;
        }

        public string Region
        {
            get { return m_region; }
```

```
            set { m_region = value; }
        }
    }
}
```

Türemiş sınıf için hiçbir başlangıç metodu yazılmamışsa derleyici default başlangıç metodunu içi, boş olarak kendisi yazacağına göre bu da taban sınıfın default başlangıç metodunu çağıracaktır. Yani biz türemiş sınıf için hiçbir başlangıç metodu yazmamış olsak bile yine de taban sınıfın başlangıç metodu çağrılacaktır. Örneğin:

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new B();      // error!
        }
    }

    class A
    {
        private int m_a;

        public A(int a)
        {
            m_a = a;
        }
    }

    class B : A
    {
        //...
    }
}
```

Başlangıç Metotlarında this Sentaksı

Başlangıç metotları normal metotlar gibi çağrılamaz. Başlangıç metotları ya new operatörü vasıtasıyla çağrılır ya da this sentaksıyla çağrılır. this sentaksı "kendi sınıfının başka bir başlangıç metodunu çağır" anlamına gelir. Örneğin:

```
using System;

namespace CSD
{

    class App
    {
        public static void Main()
        {
            A a = new A();
            Console.WriteLine(a.Val);
        }
    }

    class A
    {
        private int m_val;

        public A() : this(10)
        {
            Console.WriteLine("A default constructor");
        }

        public A(int a)
        {
            Console.WriteLine("A int constructor");
            m_val = a;
```

```
        }

        public int Val
        {
            get { return m_val; }
            set { m_val = value;}
        }
        //...
    }
}
```

Başlangıç metodunda hem base hem de this kullanılamaz. Ya base ya da this kullanılabilir. (Anımsanacağı gibi hiçbir şeyin kullanılmaması : base() sentaksının kullanıldığı anlamına gelmektedir.) this sentaksında da çağrı yine başlangıç metodunun ana bloğunun başında yapılmaktadır. Bir başlangıç metodu diğerini çağırdığında o da diğerini çağırdığında en sonunda ne olur? Akış döngüye girmeyeceğine göre en sonunda base çağırması yapılacaktır. Yani yine en önce taban sınıfın başlangıç metodu çalıştırılır. Örneğin:

```
using System;

namespace CSD
{

    class App
    {
        public static void Main()
        {
            B b = new B();
            Console.WriteLine("{0}, {1}", b.ValA, b.ValB);
        }
    }

    class A
    {
        private int m_valA;

        public A() : this(30)
        {
            Console.WriteLine("A default constructor");
        }

        public A(int a)
        {
            Console.WriteLine("A int constructor");
            m_valA = a;
        }

        public int ValA
        {
            get { return m_valA; }
            set { m_valA = value;}
        }

        //...
    }

    class B : A
    {
        private int m_valB;

        public B() : this(10, 20)
        {
        }

        public B(int a, int b)
        {
            Console.WriteLine("B int, int constructor");
            m_valB = b;
        }
```

```
        public int ValB
        {
            get { return m_valB; }
            set { m_valB = value; }
        }
    }
}
```

Pekiyi this sentaksına neden gereksinim duyulmaktadır? Bazen sınıfın pek çok başlangıç metodu bulunuyor olabilir. Bunlar da bazı ortak işlemler yapılıyor olabilir. Bu tür durumlarda kod tekrarını engellemek için bir başlangıç metodunun diğerini çağırması gerekebilmektedir. Örneğin:

```
using System;

namespace CSD
{

    class App
    {
        public static void Main()
        {
            Sample s = new Sample();
            Sample k = new Sample(1);
            Sample t = new Sample(1, 2);
            //...

        }
    }

    class Sample
    {
        private int m_a;
        private int m_b;
        private string m_name;

        public Sample() : this(10, 10, "No Name")
        {

        }

        public Sample(int a) : this(a, 10, "No Name")
        {
        }

        public Sample(int a, int b) : this(a, b, "No Name")
        {
        }

        public Sample(int a, int b, string name)
        {
            m_a = a;
            m_b = b;
            m_name = name;
        }
    }
}
```

System.Object Sınıfı

C#'ta (aslında genel olarak .NET ortamında) her sınıf doğrudan ya da dolaylı olarak System isim alanı içerisindeki Object sınıfından türetilmiş durumdadır. System.Object sınıfı çok kullanıldığı için object anahtar sözcüğü ile de temsil edilmiştir. Yani object anahtar sözcüğü ile System.Object ifadesi aynı anlamdadır.

C#'ta biz bir sınıfı hiçbir sınıftan türetmiş olmasak bile derleyici onun System isim alanı içerisindeki Object sınıfından türetilmiş olduğunu varsayar. Örneğin:

```
class Sample
```

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

ile,

```
class Sample : object
{
	//...
}
```

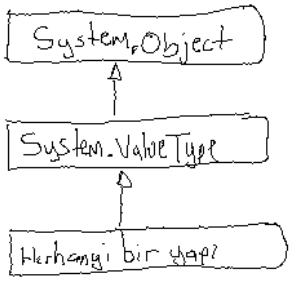
aynı anlamdadır. Biz sınıfın açıkça object sınıfından türetildiğini belirtebiliriz ya da belirtmeyebiliriz.

Madem ki tüm sınıflar System.Object sınıfından türetilmiş durumdadır. O halde biz bir sınıf türünden referansla yalnızca o sınıfın değil object sınıfının da public elemanlarını kullanabiliriz. (Visual Studio IDE'sinde intelli sense özelliğinin bu elemanları da birer seçenek olarak listelediğine dikkat ediniz.) Object sınıfının da static elemanları ve static olmayan elemanları vardır. Bu elemanlar üzerinde daha ileride durulacaktır.

Anahtar Notlar: Aslında her nesnenin bir object kısmı vardır. Ancak çizimlerde biz onu belirtmeyeceğiz.

Yapıların Türetme Durumları

C#'ta yapılar türetmeye kapalıdır. Yani bir yapıdan bir sınıf ya da yapı türetilemez. Yapı da bir sınıftan türetilemez. Ancak C#'ta her yapının System isim alanı içerisindeki ValueType isimli bir sınıftan türetildiği varsayılmaktadır. System.ValueType sınıfı da System.Object sınıfından türetilmiştir.



Ayrıca yapı bildiriminde bu türetmeyi açıkça belirtmek de yasaklanmıştır. Örneğin:

```
struct Test
{
	//...
}
```

Burada Test yapısının System.ValueType sınıfından türetildiği varsayılmaktadır. Fakat:

```
struct Test : System.ValueType		// error!
{
	//...
}
```

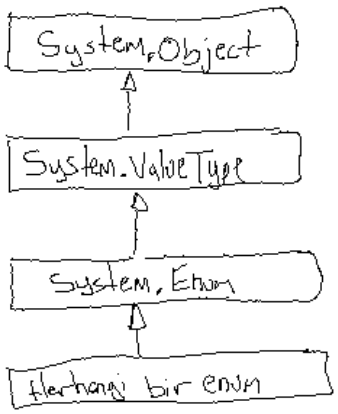
Biz bunu açıkça ifade edemeyiz. Başka bir deyişle yapılarda ':' sentaksı kullanılamaz.

Yapılar türetmeye kapalı olduğu için C#'ta yapıların protected ve protected internal elemanlara sahip olması da yasaklanmıştır.

Anımsanacağı gibi C#'ta aslında int, long, double gibi anahtar sözcükler de Int32, Int64, Double gibi yapı türlerini temsil etmektedir. Bu durumda örneğin int türü de System.ValueType sınıfından ve dolayısıyla System.Object sınıfından türetilmiş durumdadır.

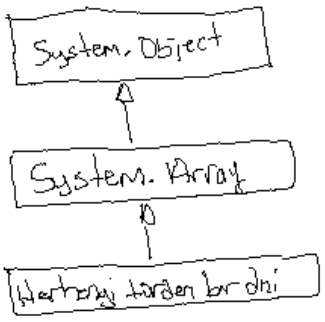
Enum'ların Türetme Durumları

Enum türleri de türetmeye kapalıdır. Yani bir enum'dan türetme yapılamaz, enum da bir sınıf ya da yapıdan türetilemez. Ancak .NET'te tüm enum türlerinin System.Enum isimli bir sınıftan türetildiği varsayılmaktadır. Bu sınıf da System.ValueType sınıfından türetilmiş durumdadır:



Dizilerin Türetme Durumları

Diziler de türetmeye kapalıdır. Yani dizilerden bir şey türetilemez, dizi de başka bir şeyden türetilemez. Ancak .NET'te tüm dizilerin -türü ne olursa olsun- System isim alanı içerisindeki Array sınıfından türetilmiş olduğu varsayılmaktadır.



Türemiş Sınıf Türünden Taban Sınıf Türüne Otomatik Dönüştürme

Normal olarak C#'ta bir sınıf türünden referans başka bir sınıf türünden referansa atanamaz. Ancak istisna olarak türemiş sınıf türünden bir referans doğrudan taban sınıf türünden bir referansa atanabilir. Yani türemiş sınıftan taban sınıfa otomatik tür dönüştürmesi vardır. Bunun tersi olan durum yani taban sınıftan türemiş sınıfa otomatik dönüştürme mevcut değildir.

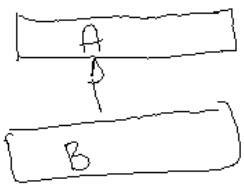
```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new B();
            A a;

            a = b;      // geçerli
            //...
        }
    }

    class A
```

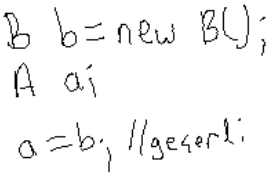
```
    {
        //...
    }

    class B : A
    {
        //...
    }
}
```

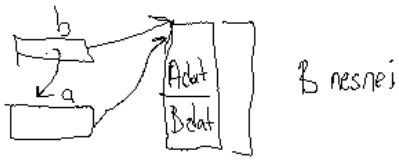
Herhangi bir türemiş sınıf referansı onun herhangi bir taban sınıfına atanabilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            C c = new C();
            B b;
            A a;

            b = c;      // geçerli
            a = b;      // geçerli
            a = c;      // geçerli
        }
    }

    class A
    {
        //...
    }

    class B : A
    {
        //...
    }

    class C : B
    {
        //...
    }
}
```

Türemiş sınıf referansı taban sınıf referansına atandığında artık taban sınıf referansı bağımsız bir taban sınıf nesnesini değil, türemiş sınıf nesnesinin taban sınıf kısmını gösteriyor durumda olur. Örneğin:

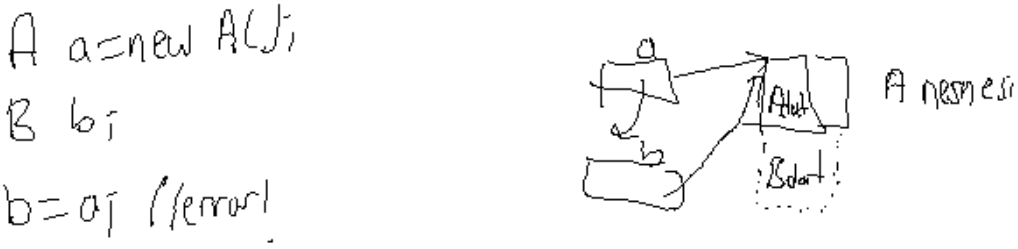






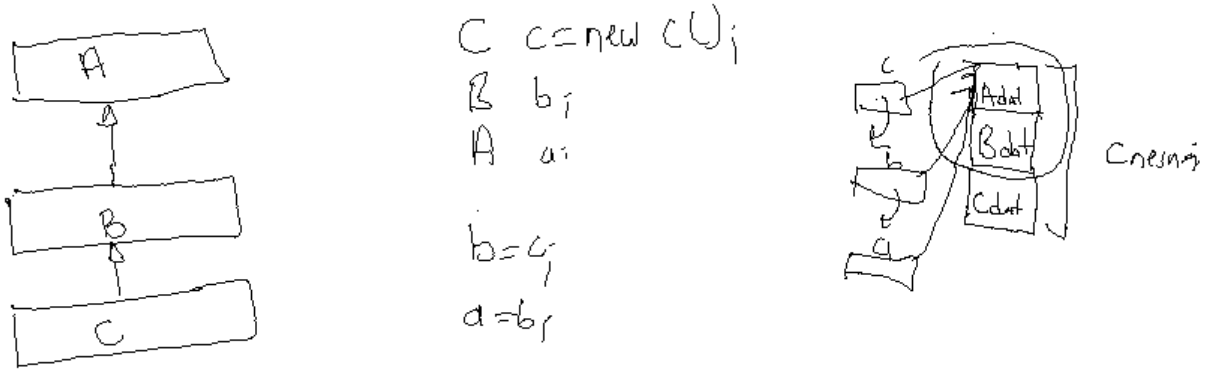
Burada a referansı B nesnesinin A kısmını gösteriyoır durumdadır. Yani biz a referansıyla işlem yaptığımızda bundan B nesnesinin A kısmı etkilenir. Zaten türemiş sınıf referansının taban sınıf referansına atanabilmesinin nedeni türemiş sınıfın taban sınıf elemanlarını içermesindendir.

Taban sınıf referansı türemiş sınıf referansına atanamaz. Eğer atanabilseydi olmayan elemanlara erişmek gibi potansiyel bir tehlike oluşurdu. Örneğin:



Burada b = a ataması mümkğn olsaydı b referansı yalnızca A kısmı olan bir nesneyi gösteriyor olurdu. Halbuki biz b referansıyla B'nin kendi elemanalrını da kullanabilmekteyiz.

Örneğin:



Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new B();
            A a;

            b.m_a = 10;
            b.m_b = 20;

            a = b;
            Console.WriteLine(a.m_a);           // 10

            a.m_a = 30;
            Console.WriteLine("{0}, {1}", b.m_a, b.m_b);      // 30, 20
        }
    }

    class A
    {
        public int m_a;
        //...
    }

    class B : A
    {
        public int m_b;
        //...
    }
}
```

Tabii türemiş sınıf referansını taban sınıf referansına atadığımızda biz artık taban sınıf referansıyla türemiş sınıf

nesnesinin yalnızca taban sınıf kısmına erişebiliriz. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            C c = new C(10, 20, 30);
            B b;
            A a;

            Console.WriteLine("{0}, {1}, {2}", c.ValA, c.ValB, c.ValC); // 10, 20, 30

            b = c;
            Console.WriteLine("{0}, {1}", b.ValA, b.ValB);  // 10, 20

            a = b;
            Console.WriteLine("{0}", a.ValA);   // 10
        }
    }

    class A
    {
        private int m_valA;

        public A(int a)
        {
            m_valA = a;
        }

        public int ValA
        {
            get { return m_valA; }
            set { m_valA = value; }
        }
        //...
    }

    class B : A
    {
        private int m_valB;

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

        public int ValB
        {
            get { return m_valB; }
            set { m_valB = value; }
        }
        //...
    }

    class C : B
    {
        private int m_valC;

        public C(int a, int b, int c) : base(a, b)
        {
            m_valC = c;
        }

        public int ValC
        {
            get { return m_valC; }
```

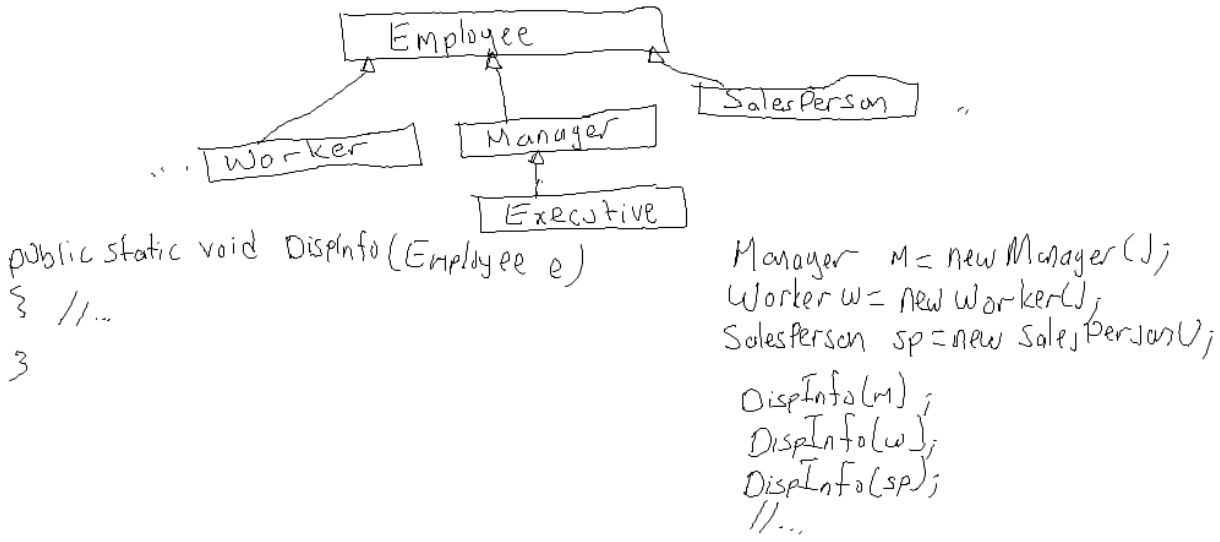
```
            set { m_valC = value; }
        }
        //...
    }
}
```

Madem ki doğrudan ya da dolaylı olarak tüm sınıflar, yapılar ve enum'lar object sınıfından türetilmiştir. O halde biz herhangi bir sınıf, yapı ya da enum türünden değişkeni object türünden referansa atayabiliriz. Örneğin:

```
object o = "ankara";        // geçerli
```

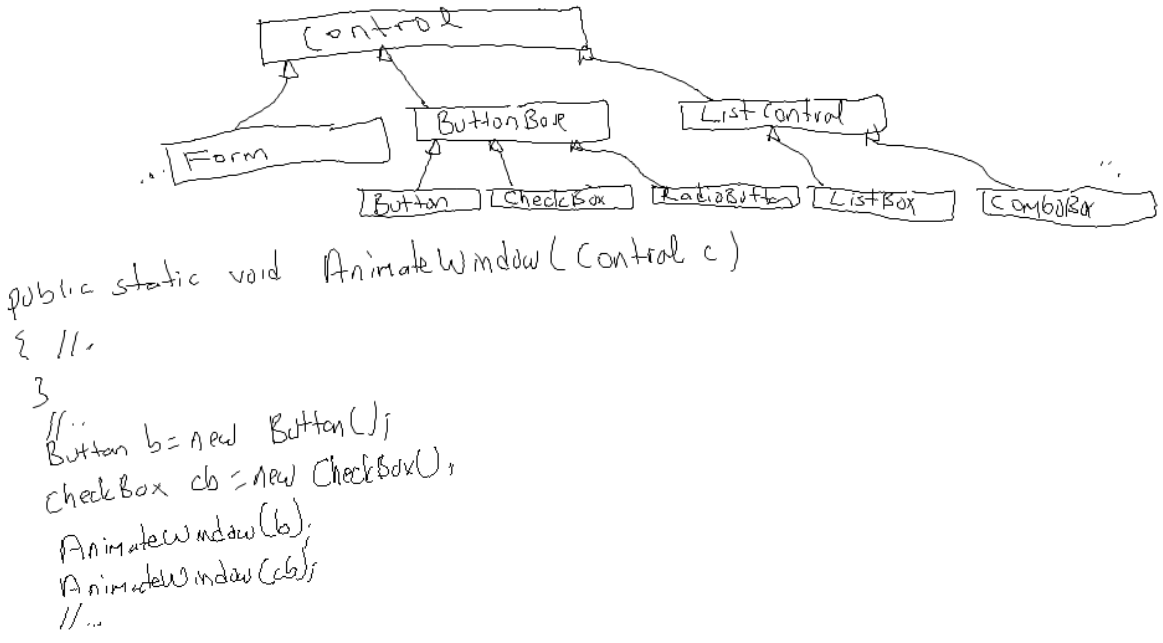
Türemiş Sınıf Referansının Taban Sınıf Referansına Atanmasının Anlamı

Türemiş sınıf referansının taban sınıf referansına atanması sayesinde biz bir türetme şeması üzerinde genel işlemler yapan metotlar yazabiliriz. Örneğin:



Burada DispInfo metodu Manager gibi, Worker gibi, Executive gibi nesnelerin Employee kısmını yazdırabilir. Yani DispInfo tüm bu sınıflarla çalışabilmektedir.

Örneğin Windows'ta görsel öğeler birer penceredir ve tüm pencereler Control sınıfından türetilmiş sınıflarla temsil edilir:

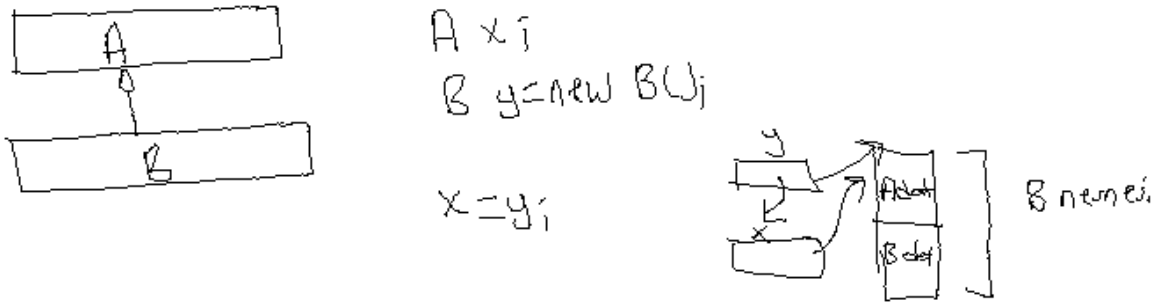


Biz Control sınıfı türünden parametre alan bir AnimateWindow isimli bir metod yazmış olalım. Bu metodu

Control sınıfından türetilmiş tüm sınıf türleriyle çağırabiliriz. Bir türetme şeması içerisinde taban sınıf türünden referans alan metotlar o türetme şeması bağlamında genel ve ortak işlemler yapmaya adaydır.

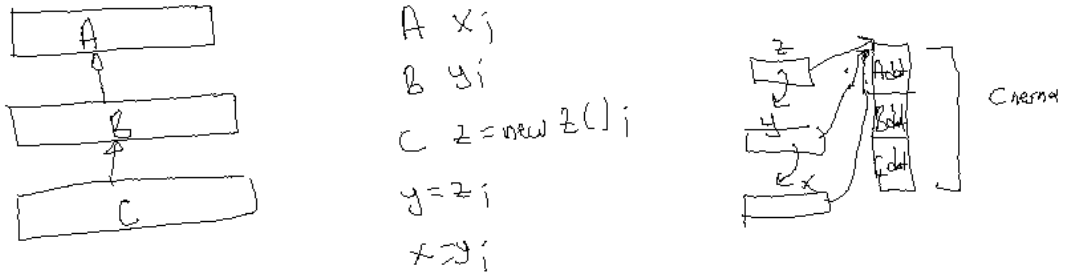
Refransların Statik ve Dinamik Türleri

Referansların (fakat yapı değişkenlerinin değil) statik ve dinamik türleri vardır. Bir referansın statik türü bildirimde belirtilen türüdür. Statik tür değişmez. Bir referansın dinamik türü ise o referansın gösterdiği nesnenin bütünün türüdür. Yani bir referans bir nesneyi gösteriyor ise gösterdiği nesnenin bütünü hangi türdense referansın dinamik türü o türdendir. Örneğin:



Burada x'in statik türü A, dinamik türü B'dir. Çünkü burada x referansı bağımsız bir A nesnesini göstermiyor, aslında bir B nesnesinin A kısmını gösteriyor. x'in gösterdiği yerdeki nesnenin bütünü B türündendir. Burada y'nin statik türü de dinamik türü de B'dir.

Örneğin:



Burada z'nin statik ve dinmik türü C'dir. y'nin static türü B, dinamik türü C'dir. x'in statik türü A, dinamik türü C'dir.

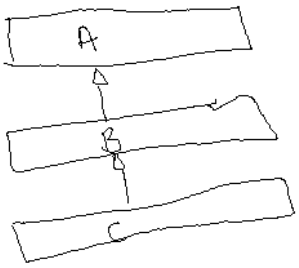
Aslında normal olarak referansın statik ve dinamik türleri birbirleriyle aynıdır. Türemiş sınıf referansı taban sınıf referansına atandığı zaman bunlar farklılaşır.

Örneğin:

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

o = s;
```

Burada o'nun statik türü object, dinamik türü Sample'dır.

Refransın dinamik türü sürekli değiebilir. Örneğin:




```
A a;
B b=new B();
C c=new C();
//..
a=new A();
//a'nın dinamik türü A'dır
a=b;
//a'nın dinamik türü B'dir
a=c;
//a'nın dinamik türü c'dir.
```

Anahtar Notlar: Bir referansın dinamik türünü yazı olarak elde etmek için r.GetType().Name ifadesi kullanılabilir.

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            A x;
            B y = new B();
            C z = new C();

            y = z;
            x = y;

            Console.WriteLine("{0}, {1}, {2}", x.GetType().Name, y.GetType().Name, z.GetType().Name);
        }
    }

    class A
    {
        //...
    }

    class B : A
    {
        //...
    }

    class C : B
    {
        //...
    }
}
```

Örneğin:

```
using System;

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

```
            A a = new A();
            B b = new B();
            C c = new C();

            o = "Test";
            Console.WriteLine(o.GetType().Name);    // String
            o = a;
            Console.WriteLine(o.GetType().Name);    // A
            o = b;
            Console.WriteLine(o.GetType().Name);    // B
            o = c;
            Console.WriteLine(o.GetType().Name);    // C
        }
    }

    class A
    {
        //...
    }

    class B : A
    {
        //...
    }

    class C : B
    {
        //...
    }
}
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            A x = new A();
            B y = new B();
            C z = new C();

            Foo(x);
            Foo(y);
            Foo(z);
        }

        public static void Foo(object o)
        {
            Console.WriteLine(o.GetType().Name);
        }
    }

    class A
    {
        //...
    }

    class B : A
    {
        //...
    }

    class C : B
    {
        //...
```

```
    }
}
```

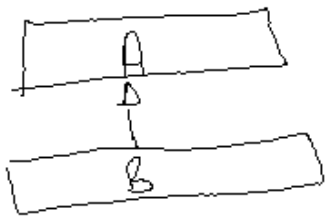
Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            object[] objs;

            objs = new object[] { "Ankara", new A(), new B(), new C(), new Random()};
            foreach (object o in objs)
                Console.WriteLine(o.GetType().Name);
        }
    }

    class A
    {
        //...
    }

    class B : A
    {
        //...
    }

    class C : B
    {
        //...
    }
}
```

Yukarıdan Aşağıya Yapılan Referans Dönüştürmeleri (Downcasting)

Normal olarak türemiş sınıf referansı taban sınıf referansına atanabilir. Yani türemiş sınıftan taban sınıfa otomatik dönüştürme (implicit) vardır. Bu normal dönüştürmeye yukarıya doğru dönüştürme (upcasting) de denilmektedir. Fakat taban sınıf referansı türemiş sınıfın referansına doğrudan atanamaz. Ancak eğer istenirse tür dönüştürme operatörü ile bu işlem yapılabilir. Yani taban sınıftan türemiş sınıfa otomatik (implicit) dönüştürme yoktur ancak tür dönüştürme operatörü ile dönüştürme (explicit) vardır.

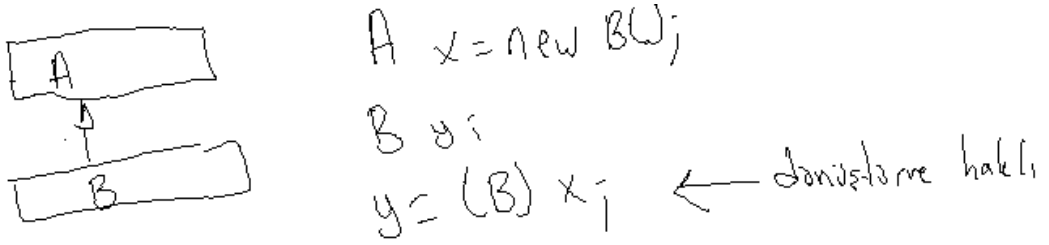
Bazen taban ısnıf referansı bir türemiş sınıf nesnesini gösteriyor olabilir. Yani taban sınıf referansının dinamik türü bir türemiş sınıf türünden olabilir. Bizim onu yeniden orijinal türe dönüştürmemiz gerekebilir. Örneğin:



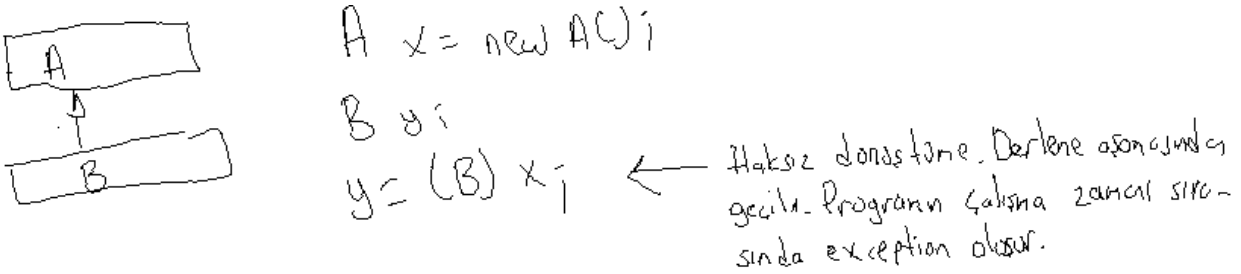

```
A x;
B y=new B();

B z;

x=y;
//...
z=x; //hata!
z=(B)x;
```

Buarada x referansı aslında B nesnesini göstermektedir. Onun yeniden B olarak kullanılabilmesi için aşağıya doğru dönüştürme gerekir.

Aşağıya dönüştürmelerde derleme aşamasından her zaman geçilir. Ancak program çalışırken dönüştürme noktasında ayrıca haklılık kontrolü yapılmaktadır. Eğer dönüştürme haklıysa sorun çıkmaz. Fakat haksızsa exception oluşur ve program çöker. Eğer dönüştürülmek istenen referansın dinamik türü dönüştürülmek istenen türü içeriyorsa dönüştürme haklıdır, içermiyorsa haksızdır. Başka bir deyişle taban sınıf referansı türemiş sınıf referansına dönüştürülürken o referansın gösterdiği yerdeki nesne dönüştürülecek türü içeriyorsa dönüştürme haklıdır. Örneğin:



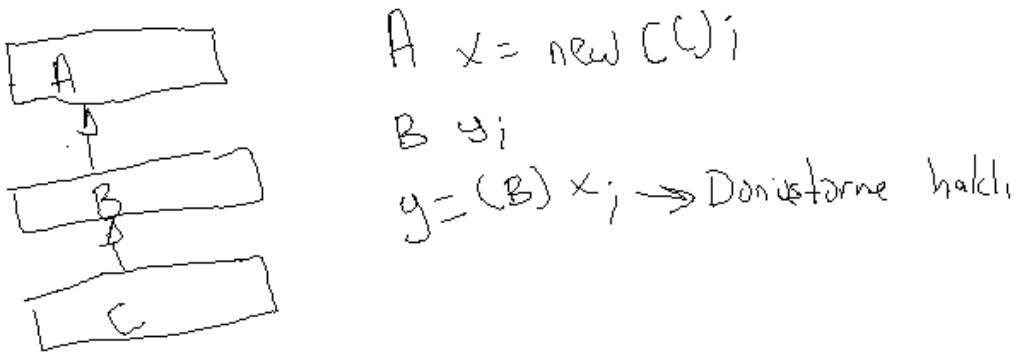
Burada x referansının gösterdiği yerde B nesnesi vardır. Dönüştürme haklıdır. Başka bir deyişle x'in dinamik türü dönüştürülmek istenen B sınıfını içermektedir. Örneğin:



Örneğin:

```
using System;

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

            y = (B)x;       // haksız dönüştürme! exception oluşur!
        }
    }

    class A
    {
        //...
    }

    class B : A
    {
        //...
    }
}
```

Örneğin:




```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            A x = new C();
            B y;

            y = (B)x;       // Dönüştürme haklı!
        }
    }

    class A
    {
        //...
    }

    class B : A
    {
        //...
    }

    class C : B
    {
        //...
    }
}
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            object o = new B();
            A a;
            B b;
            C c;
            a = (A)o;
            Console.WriteLine("Passed 1");        // Haklı dönüştürme
            b = (B)o;
            Console.WriteLine("Passed 2");        // Haklı dönüştürme
            c = (C)o;                            // Haksız dönüştürme, exception oluşur
            Console.WriteLine("Passed 3");        // Haklı dönüştürme
        }
    }
```

```
    class A
    {
        //...
    }

    class B : A
    {
        //...
    }

    class C : B
    {
        //...
    }
}
```

Örneğin:

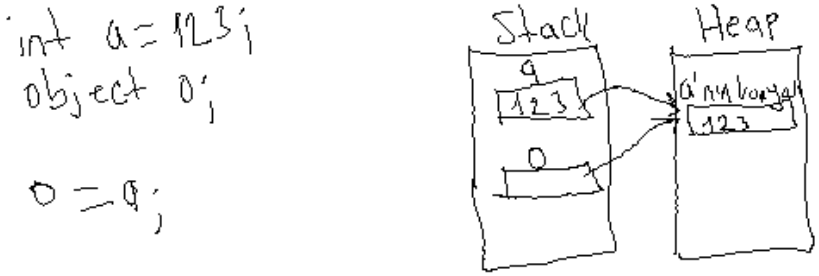
```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            object o = "This is a test";
            string s = (string)o;       // haklı dönüştürme
            Console.WriteLine(s);
        }
    }
}
```

Aralarında türetme ilişkisi olmayan iki sınıf arasında tür dönüştürme operatörüyle de dönüştürme yapılamaz. Bu durumda derleme sırasında error oluşur. Örneğin:

```
Random r = new Random();
string s;

s = (string)r;            // error! Random ile string arasında bir türetme ilişkisi yok!
```

Kutulama Dönüştürmesi (Boxing Conversion) ve Kutuyu Açma Dönüştürmesi (Unboxing Conversion)

.NET'te referanslar stack'teki bir nesneyi gösteremezler. Yalnızca heap'teki nesneleri gösterebilirler. Bilindiği gibi yapı nesneleri stack'te yaratılmaktadır. Yapılar System.ValueType ve System.Object sınıflarından türetilmiş durumdadırlar. Bu durumda yapıların object referanslarına atanabilmesi gerekir (türemişten tabana yapılan atamalar). Fakat öte yandan referanslar da ancak heap'teki nesneleri gösterebilirler. Peki ne olacaktır? İşte C#'ta ne zaman bir yapı değişkeni System.ValueType ya da System.Object referansına atansa önce o değişkeninin otomatik olarak heap'te bir kopyası çıkartılır ve referans heap'teki bu kopyayı gösterir. Bu sürece "kutulama dönüştürmesi (boxing conversion)" denilmektedir. Örneğin:



Burada o stack'teki a'yı değil, heap'teki onun kopyasını gösteriyor durumdadır.

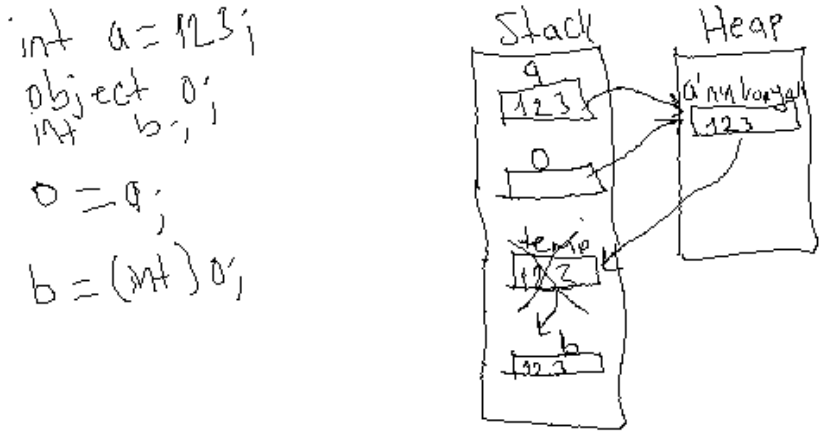
Kutulama dönüştürmesinden sonra stack'teki yapı nesnesiyle heap'teki kopyası tamamen iki ayrı nesne

durumundadır. Stack'teki yapı nesnesi akış bloktan çıktığında yok edilir. Heap'teki kopyası ise çöp toplayıcı (garbage collector) sistem tarafından yok edilmektedir.

Kutulama dönüştürmesiyle heap'te kopyası çıkartılan nesne aşağıya doğru dönüştürme yoluyla yeniden orijinal türe dönüştürülebilir. Buna kutuyu açma dönüştürmesi (unboxing conversion) denilmektedir. Örneğin:

```
int a = 123;
object o;
int b;

o = a;          // kutulama dönüştürmesi
b = (int)o;     // kutuyu açma dönüştürmesi
```

Kutuyu açma dönüştürmesi sırasında stack'te ilgili yapı türünden geçici bir nesne oluşturulur, sonra heap'teki yapı nesnesi bu nesneye kopyalanır. Böylece dönüştürme işleminden (örneğin (int) o işleminden) stack'te yaratılmış geçici bir nesne elde edilmiş olur. Biz onu aynı türden başka bir yapı nesnesine atarız. İlgili ifade bittiğinde bu geçici yapı nesnesi de yok edilmektedir.

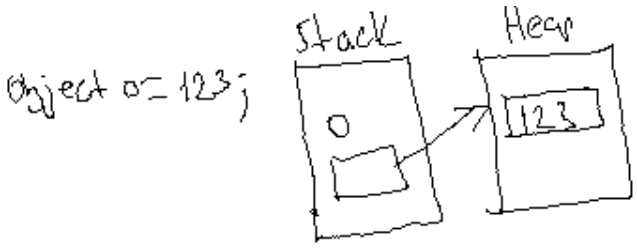


Kutuyu açma dönüştürmesi ile heap'teki yapı nesnesi yok edilmez. O nesne normal olarak çöp toplayıcı tarafından yok edilmektedir.

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int a = 123, b;
            object o;

            o = a;          // kutulama dönüştürmesi
            b = (int)o;     // kutuyu açma dönüştürmesi
            Console.WriteLine(b);

        }
    }
}
```

object referansına doğrudan sabit de atanabilir. Bu durumda yine heap'te o sabit için o sabitin türünden bir nesne yaratılır, sabit o nesneye yerleştirilir. Referans da yine o sabitin yerleştirildiği nesneyi gösterecektir. Örneğin:



Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            object[] objs = new object[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            int val;

            for (int i = 0; i < objs.Length; ++i)
            {
                val = (int)objs[i];
                Console.Write("{0} ", val);
            }
            Console.WriteLine();
        }
    }
}
```

foreach deyimi tür dönüştürmesini zaten kendisi yapmaktadır. O halde yukarıdaki kod şöyle de yazılabilir:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            object[] objs = new object[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

            foreach (int val in objs)
                Console.Write("{0} ", val);
            Console.WriteLine();
        }
    }
}
```

T1 türünden T2 türüne otomatik dönüştürmenin var olması dinamik türü T1 olan referansın T2'ye dönüştürülebileceği anlamına gelmez. Örneğin:

```
int a = 123;
object o;
long b;

o = a;       // kutulama dönüştürmesi
b = (long)o; // haksız dönüştürme, exception oluşur
```

Burada int'in long'a doğrudan atanabilmesi o'nun long'a dönüştürülebileceği anlamına gelmez. int ve long bağımsız iki yapıdır. Tabi biz bu işlemi şöyle yapabilirdik:

```
int a = 123;
object o;
```

```
long b;

o = a;       // kutulama dönüştürmesi
b = (int)o;  // geçerli, herşey normak
```

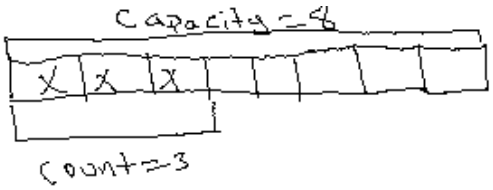
Collection Sınıf Kavramı

Bir grup nesneyi belli bir algoritmaya göre tutan ve bize verebilen özel sınıflara collection denilmektedir. .NET'te pek çok collection sınıf vardır. Aslında diziler de bu bakımdan collection olarak değerlendirilebilirler. Diziler dilin sentaksı tarafından doğrudan desteklenmektedir. .NET'in collection sınıfları kütüphaneyi yazanlar tarafından bir sınıf olarak gerçekleştirilmişleridr. Biz burada yalnızca ArrayList sınıfını göreceğiz. Diğer collection sınıflar "C# İle .NET Ortamında Uygulma Geliştirme-1" numaralı kursta ele alınmaktadır.

ArrayList Sınıfı

C#'ta new operatörüyle yaratılmış olan bir dizi büyütülemez ya da küçültülemez. Ancak şüphesiz her zaman istenilen uzunlukta yeni bir dizi yaratılabilir. Fakat bazı durumlarda biz tutmak istediğimiz elemanların sayısını daha programı yazarken bilemeyiz. Örneğin bir çizim programında kullanıcının çizdiği çizginin noktalarını bir dizide tutacak olalım. Kullanıcının ne kadar çizim yapacağını programı yazarken bilemeyiz. Ya da örneğin bir yerden gelen bilgileri bir dizide saklayacak olalım. Ne kadar bilgi geleceğini bilemeyebiliriz. Benzer biçimde bir dosyadaki belli yazıların yerlerini bir dizide tutacak olalım. Dosyada bu yazıdan kaç tane olduğunu başlangıçta bilemeyiz. İşte böylesi durumlarla başa çıkabilmek için C#'ta şöyle bir yol izlenebilir: Başlangıçta küçük bir dizi alınır. Bu diziye ekleme yapılır. O dizi dolunca yeni ve daha büyük bir dizi yaratılır. Eski dizideki değerler de bu yeni diziye kopyalanır ve yeni diziden devam edilir. Kullanılmayan eski dizileri çöp toplayıcı mekanizma arka planda silecektir. İşlemler böyle devam ettirilir... İşte System.Collection isim alanındaki ArayList sınıfı bu sıkıcı işlemleri arka planda bizim için bu biçimde yapmaktadır.

ArrayList sınıfının içerisinde object türünden bir dizi vardır. Böylece her şey object'miş gibi onun içerisinde tutulmaktadır. ArrayList içerisindeki object dizisinin tahsis edilmiş uzunluğuna Capacity denir ve Capacity isimli read/write propererty ile bu değer alınıp, set edilebilir. Dizinin dolu olan miktarına da Count denilmektedir. Sınıfın Count isimli read-only property'si ile bu değer alınabilir. Örneğin:



Count değeri Capacity değerine geldiğinde Capacity eskisinin iki katı artırılmaktadır. Eski dizideki değerler yeni diziye kopyalanır ve artık oradan devam edilir. Eski dizi de çöp toplayıcı sistem tarafından silinecektir.

ArrayList'e eleman eklemek için Add isimli metot kullanılmaktadır:

```
public virtual int Add(Object value)
```

Metot eklenen elemanın dizi içerisindeki indeksiyle geri döner. ArrayList sınıfının object türünden int parametreli indeksleyicisi bize ilgili indeks'teki elemanı verir. (İndeksleyiciler bir sınıf türünden referansı ya da bir yapı değişkenini bizim köşeli parantez operatörü ile kullanmamaıza olanak sağlayan elemanlardır. İndeksleyicilerin yazımı ve kullanımı ileride ayrıntılarıyla ele alınmaktadır.) Yani biz ArrayList içerisindeki bir elemanı köşeli parantez operatörüyle indeks numarası vererek object biçiminde geri alabiliriz. Örneğin:

```
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 * i);

            for (int i = 0; i < al.Count; ++i)
            {
                int val = (int)al[i];
                Console.Write("{0} ", val);
            }
            Console.WriteLine();
        }
    }
}
```

Sınıf Çalışması: Bir döngü içerisinde klavyeden Console.ReadLine metodu ile isimler okuyunuz. Bu isimleri bir ArrayList'e string nesneleri olarak ekleyiniz. İsim yerine "exit" yazıldığında döngüden çıınız ve bu ArrayList'i dolaşarak isimleri aralarına ',' koyarak yazdırınız.

Çözüm:

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            ArrayList names = new ArrayList();
            string name;

            for (;;)
            {
                Console.Write("Bir isim giriniz:");
                name = Console.ReadLine();
                if (name == "exit")
                    break;
                names.Add(name);
            }
            for (int i = 0; i < names.Count; ++i)
            {
                if (i != 0)
                    Console.Write(", ");
                name = (string)names[i];
                Console.Write(name);
            }
            Console.WriteLine();
        }
    }
}
```

ArrayList sınıfı IEnumerable arayüzünü desteklediği için foreach deyimiyle kullanılabilir. Örneğin:

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();
```

```
            string name;

            for (; ; )
            {
                Console.Write("Adı Soyadı:");
                name = Console.ReadLine();
                if (name == "exit")
                    break;
                al.Add(name);
            }

            foreach (object o in al)
            {
                string s = (string)o;
                Console.Write(s);
            }
            Console.WriteLine();
        }
    }
}
```

foreach döngüsünde döngü değişkenine atama sırasında tür dönüştürmesi yapıldığına göre kod daha pratik şöyle de yazılabilir:

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();
            string name;

            for (; ; )
            {
                Console.Write("Adı Soyadı:");
                name = Console.ReadLine();
                if (name == "exit")
                    break;
                al.Add(name);
            }

            foreach (string s in al)
                Console.Write(s);
            Console.WriteLine();
        }
    }
}
```

ArrayList sınıfında Count ve Capacity değerlerinin değişimi aşağıdaki programda görülebilir:

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();

            for (int i = 0; i < 100; ++i)
            {
                Console.WriteLine("Count = {0}, Capacity = {1}", al.Count, al.Capacity);
```

```
                al.Add(i);
            }
        }
    }
}
```

ArrayList sınıfının Insert metodu araya ekleme yapmak için kullanılır:

```
public void Insert(int index, Object value)
```

Metodun birinci parametresi insert edilecek indeksi, ikinci parametresi insert edilecek değeri belirtir. Metot yeni değer belirtilen indekste olacak biçimde insert işlemini yapar. Örneğin:

```
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(5, 1000);

            foreach (int x in al)
                Console.Write("{0} ", x);
            Console.WriteLine();
        }
    }
}
```

ArrayList sınıfının RemoveAt isimli metodu belli bir indeskteki elemanı silmek için kullanılır:

```
public void RemoveAt(int index)
```

Örneğin:

```
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);

            foreach (int x in al)
                Console.Write("{0} ", x);
            Console.WriteLine();

            al.RemoveAt(4);

            foreach (int x in al)
                Console.Write("{0} ", x);
            Console.WriteLine();
        }
    }
```

```
}
```

Eleman RemoveAt metoduyla silindiğinde collection nesnesinin Count değeri bir azaltılır ancak Capacity değeri değişmez.

ArrayList sınıfının Clear isimli metodu collection içerisindeki tüm elemanları silmek için kullanılır. Bu durumda Capacity düşümü yapılmaz. Yani nesnenin Cpacity değişmez fakat Count değeri 0 olur.

ArrayList sınıfının Sort isimli isimli metodu dizilim içerisindeki elemanları sıraya dizmekte kullanılır:

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();

            al.Add(13);
            al.Add(21);
            al.Add(2);
            al.Add(9);

            al.Sort();

            foreach (int x in al)
                Console.Write("{0} ", x);
            Console.WriteLine();
        }
    }
}
```

ArrayList sınıfının Capacity property'sine değer atayarak onun kapasitesini değiştirebiliriz. Ancak Capacity değerini Count değerinin altına düşüremeyiz. Eğer bunu yapmaya çalışırsak exception oluşur. Bazen kaç elemana gereksinim duyulabileceği baştan az çok kestirilebiliyorsa Capacity değerini değiştirerek gereksiz yeniden tahsisatın (reallocation) önüne geçilebilir. ArrayList sınıfının ayrıca TrimToSize metodu kapasiteyi Count değerine çekmektedir.

ArrayList sınıfının Reverse metodu diziyi ters yüz eder. Örneğin:

```
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);

            foreach (int x in al)
                Console.Write("{0} ", x);
            Console.WriteLine();

            al.Reverse();

            foreach (int x in al)
```

```
                Console.Write("{0} ", x);
            Console.WriteLine();
        }
    }
}
```

Pekiyi ArrayList benzeri bir sınıfı biz yazabilir miyiz? Aşağıda bir fikir vermek amacıyla böyle bir sınıf yazılmıştır. (Aşağıdaki kodda indeskleyici kullanılmıştır. İndeskleyici konusu ileride ele alınacaktır).

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            MyArrayList ma = new MyArrayList();

            for (int i = 0; i < 1000; ++i)
                ma.Add(i);

            for (int i = 0; i < ma.Count; ++i)
            {
                int val = (int)ma[i];
                Console.Write("{0} ", val);
            }
            Console.WriteLine();
        }
    }

    class MyArrayList
    {
        private object[] m_objs;
        private int m_count;
        private int m_capacity;

        public MyArrayList()
        {
            m_count = 0;
            m_capacity = 4;
            m_objs = new object[m_capacity];
        }

        public int Count
        {
            get { return m_count; }
        }

        public int Capacity
        {
            get { return m_capacity; }
            }

        public int Add(object o)
        {
            if (m_count == m_capacity)
            {
                m_capacity *= 2;
                object[] temp = new object[m_capacity];
                for (int i = 0; i < m_count; ++i)
                    temp[i] = m_objs[i];
                m_objs = temp;
            }

            m_objs[m_count++] = o;
            return m_count - 1;
        }
```

```
        public object this[int index]
        {
            get { return m_objs[index];  }
            set { m_objs[index] = value;  }
        }
        //...
    }
}
```

Taban Sınıf ve Türemiş Sınıfta Aynı İsimli Elemanlar ve new Belirleyicisi

Taban ve türemiş sınıflarda aynı isimli elemanlar bulunabilir. Genel olarak bu durum yasaklanmamıştır. Böylesi bir durumda türemiş sınıf referansıyla ya da türemiş sınıf içerisinde o isim kullanıldığında türemiş sınıftaki eleman anlaşılır. Yani türemiş sınıftaki eleman taban sınıftaki elemanı gizlemiş olur. Fakat bu tür durumlarda C# derleyicisi bir uyarı mesjıyla duruma dikkat çekmektedir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new B();
            b.X = 10;       // Dikkat B'nin X'i

            A a = new A();
            a.X = 20;       // A'nın X'i
        }
    }

    class A
    {
        public int X;
        //...
    }

    class B : A
    {
        public int X;
        //...
    }
}
```

Eğer programcı bu çakışmayı bilerek ve isteyerek yapmışsa uyarıyı kesmek için bildirimde new anahtar sözcüğünü kullanmalıdır. new anahtar sözcüğü burada tahsisat görevinde değil tamamen başka bir görevdedir. Sentaks bakımından burada kullanılan new belirleyicisi erişim belirleyicilerle aynı grupta olduğu için yer değiştirmeli olarak kullanılabilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new B();
            b.X = 10;       // Dikkat B'nin X'i

            A a = new A();
            a.X = 20;       // A'nın X'i

        }
    }
```

```
    class A
    {
        public int X;
        //...
    }

    class B : A
    {
        public new int X;
        //...
    }
}
```

Ayrıca new belirleyicisinin gereksiz bir biçimde kullanılması da uyarı oluşturmaktadır. Yani eğer böyle bir gizleme durumu yoksa new kullanmamalıyız. Örneğin:

```
class A
{
    //...
}

class B : A
{
    public new int X;       // uyarı, new belirleyicisine gerek yok
    //...
}
```

Eğer taban sınıftaki aynı isimli eleman taban sınıfın private bölümündeyse bu anlamda bir gizleme söz konusu değildir. Bu nedenle new belirleyicisinin kullanılması gerekmez. Zaten böylesi bir durumda new belirleyici kullanılırsa uyarı oluşur. Tabii taban sınıftaki aynı isimli eleman protected ya da public bölümdeyse yine new beirleyicisi uyarıyı kesmek için gerekir.

Taban ve türemiş sınıflarda aynı isimli metotların bulunması durumunda metotların parametrik yapısı farklıysa bir gizleme durumu oluşmaz. Örneğin aşağıdaki durumda new belirleyici kullanılmamalıdır:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new B();

            b.Foo();        // A.Foo
            b.Foo(10);      // B.Foo
        }
    }

    class A
    {
        public void Foo()
        {
            //...
        }
        //...
    }

    class B : A
    {
        public void Foo(int a)
        {
            //...
        }
        //...
    }
```

```
}
```

Bu tür durumlarda taban ve türemiş sınıflardaki aynı isimli metotlar birlikte "overload resolution" işlemine sokulmamaktadır. Önce türemiş sınıfa bakılır. Orada bu çağrıyı kabul edecek (yani uygun (applicable)) bir metot varsa artık taban sınıfa hiç bakılmaz. Ancak türemiş sınıfta o çağrıyı kabul edecek bir metot yoksa (yani uygun bir metot yoksa) bu durumda taban sınıfa bakılmaktadır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new B();

            b.Foo(10);     // B.Foo çağrılrı
        }
    }

    class A
    {
        public void Foo(int a)
        {
            //...
        }
        //...
    }

    class B : A
    {
        public void Foo(long b)
        {
            //...
        }
        //...
    }
}
```

Fakat örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new B();

            b.Foo(10.2);     // A.Foo çağrılrı
        }
    }

    class A
    {
        public void Foo(double a)
        {
            //...
        }
        //...
    }

    class B : A
    {
        public void Foo(long b)     // Uygun metot değil
```

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

Eğer taban ve türemiş sınıflarda aynı isimli ve aynı parametrik yapıya sahip metotlar varsa bir gizleme söz konusudur. Bu durumda uyarıyı kesmek için yine türemiş sınıfta new belirleyicisinin kullanılması gerekir. Örneğin:

```
class A
{
    public void Foo(int a)
    {
        //...
    }
    //...
}

class B : A
{
    public new void Foo(int b)      // new uyarıyı kesmek için gerekir
    {
        //...
    }
    //...
}
```

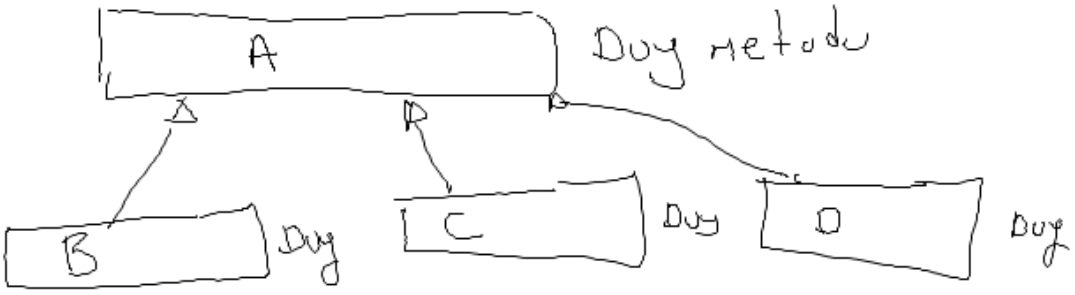
Çokbiçimlilik (Polymorphism)

Bir dilin nesne yönelimli (object oriented) olarak kabul edilmesi için çokbiçimlilik özelliğine sahip olması gerekir. Eğer bir dilde sınıflar varsa, türetme varsa fakat çokbiçimlilik yoksa ona nesne yönelimli yerine "nesne tabanlı (object based)" dil denilmektedir. C++, C#, Java, VB.NET vs. nesne yönelimlidir. Fakat örneğin klasik VB 6.0 nesne tabanlıdır.

Çok biçimlilik biyolojiden aktarılmış bir terimdir. Biyolojide çokbiçimlilik canlıların çeşitli doku ve organlarının temel işlevleri aynı kalmak üzere türlere göre farklılıklar göstermesine denilmektedir. Örneğin kulak pek çok canlıda vardır ve temel işlevi duymaktır. Fakat her canlının kulağı onların yaşam koşullarına göre az çok farklılıklar göstermektedir.

Yazılımda çokbiçimliliği üç değişik yönden bakarak tanımlayabiliriz:

1) Biyolojik tanım: Bu tanım biyolojideki çokbiçimlilikten alınan tanımdır. Buna göre çokbiçimlilik taban sınıfın belli bir metodunun türemiş sınıflarda o sınıflara göre temel işlevi aynı kalmak üzere farklı biçimlerde yeniden tanımlanmasıdır. Örneğin:



2) Yazılım Mühendisliği Tanımı: Çokbiçimlilik türden bağımsız kod parçalarının oluşturulması için kullanılan bir tekniktir.

3) Aşağı Seviyeli Tanım: Çokbiçimlilik önceden yazılan kodların sonradan yazılan kodları çağırabilmesi

özelliğidir. (Tabi normalde sonradan yazılan kodların önceden yazılan kodları çağırabilmesi gerekir. Halbuki burada tam ters bir durum söz konusudur)

Sanal (Virtual) Metotlar ve Override İşlemi

C#'ta çokbiçimlilik sanal metotlarla gerçekleştirilmektedir. Bir metodu sanal (virtual) yapabilmek için metot bildiriminde virtual anahtar sözcüğü kullanılır. virtual anahtar sözcüğü erişim belirleyici anahtar sözcüklerle aynı sentaktik grup içerisindedir. Bu nedenle onlarla yer değiştirmeli olarak yazılabilir. (Yani örneğin public virtual ya da virtual public yazılabilir.)

Eğer taban sınıftaki bir sanal metot türemiş sınıfta aynı erişim belirleyici ile, aynı geri dönüş değeri türü ile aynı isim ve parametrik yapıyla bildirilirse fakat override anahtar sözcüğü kullanılırsa bu duruma "taban sınıftaki sanal metodun türemiş sınıfta override edilmesi" denilmektedir. override anahtar sözcüğü de erişim belirleyici anahtar sözcüklerle aynı sentaktik grup içerisindedir. Yer değiştirmeli olarak yazılabilir. Örneğin:

```
class A
{
    public virtual void Foo(int a)
    {
        Console.WriteLine("A.Foo");
    }
    //...
}

class B : A
{
    public override void Foo(int a)
    {
        Console.WriteLine("B.Foo");
    }
    //...
}
```

Burada new belirleyicisi kullanılmaz. A sınıfındaki sanal Foo metodu B'de override edilmiştir.

Override etme işleminde şunlara dikkat edilmelidir:

- Taban ve türemiş sınıftaki metotların erişim belirleyicileri aynı olmak zorundadır. Örneğin taban sınıftaki public metot türemiş sınıfın protected bölümünde override edilemez.

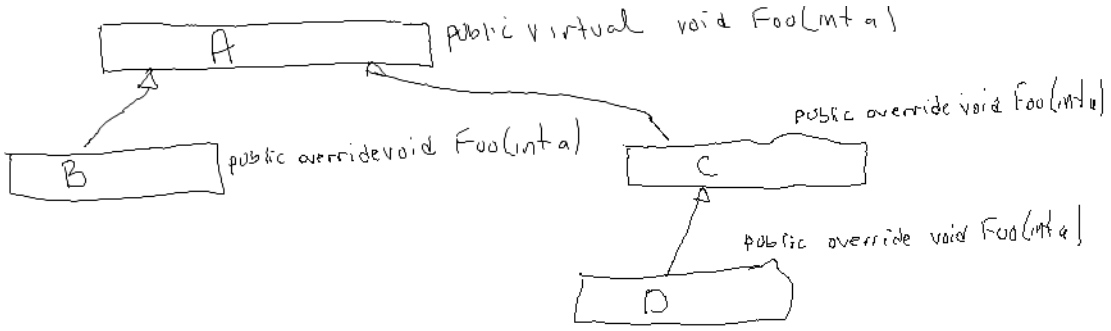
- Override etme işleminde metotların geri dönüş değerleri aynı türden olmak zorundadır.

- Override etme işleminde metotların imzaları (yani isimleri ve parametrik yapıları) aynı olmak zorundadır. (Tabi parametre değişken isimlerinin önemi yoktur.)

- Taban sınıfta sanal olmayan metot override edilemez. Yani override işlemi ancak taban sınıflardaki sanal metotlar için yapılabilir.

-Sınıfın static metotları virtual (ya da override) yapılamaz. Ancak static olmayan metotlar sanal olabilir.

Aslında override metotlar da sanal metotlardır. virtual anahtar sözcüğü ile override anahtar sözcüğü arasındaki tek fark virtual anahtar sözcüğünün sanallığı başlatmak için override anahtar sözcüğünün ise sanallığı devam ettirmek için kullanılmasıdır. Override edilmiş bir metot türemiş sınıflarda yeniden override edilebilir. Örneğin:




```
class A
{
    public virtual void Foo(int a)
    {
        Console.WriteLine("A.Foo");
    }
    //...
}

class B : A
{
    public override void Foo(int a)
    {
        Console.WriteLine("B.Foo");
    }
    //...
}

class C : B
{
    public override void Foo(int a)
    {
        Console.WriteLine("C.Foo");
    }
    //...
}

class D : C
{
    public override void Foo(int a)
    {
        Console.WriteLine("D.Foo");
    }
    //...
}
```

Taban sınıftaki sanal metot türemiş sınıfta override edilmek zorunda değildir. Örneğin istenirse sanal metot türemiş sınıfta override edilmez fakat ondan da türemiş sınıf da override edilebilir. Örneğin:

```
class A
{
    public virtual void Foo(int a)
    {
        Console.WriteLine("A.Foo");
    }
    //...
}

class B : A
{
    //...
}

class C : B
{
    public override void Foo(int a)
```

```
    {
        Console.WriteLine("C.Foo");
    }
    //...
}
```

Ancak bunun mümkün olabilmesi için sanal metodun görünür (visible) olması gerekmektedir. Yani Yukarıdaki örnekte A'daki sanal metodu gizleyen B'de aynı isimli ve parametrik yapıya sahip sanal olmayan bir Foo metodu olsaydı C'de override işlemini yapamazdık.

Çokbiçimli Mekanizma

Bir referansla static olayan bir metodu çağırdığımız düşünelim. Metot referansın statik türüne ilişkin sınıfta arfanır. Orada bulunamazsa yukarıya doğru taban sınıflara da bakılır. Sonra metodun sanal olup olmadığına bakılır. Eğer metot sanal değilse bulunan metot çağrılır. Metot sanalsa referansın dinamik türüne ilişkin override edilmiş metot çağrılır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            A a;
            C c = new C();

            a = c;
            a.Foo(0);
            a.Bar(0);
        }
    }

    class A
    {
        public virtual void Foo(int a)
        {
            Console.WriteLine("A.Foo");
        }

        public void Bar(int a)
        {
            Console.WriteLine("A.Bar");
        }
        //...
    }

    class B : A
    {
        public override void Foo(int a)
        {
            Console.WriteLine("B.Foo");
        }

        public new void  Bar(int a)
        {
            Console.WriteLine("B.Bar");
        }
        //...
    }

    class C : B
    {
        public override void Foo(int a)
        {
            Console.WriteLine("C.Foo");
```

```
        }

        public new void Bar(int a)
        {
            Console.WriteLine("C.Bar");
        }
        //...
    }
}
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            A a = new A();
            B b = new B();
            C c = new C();

            DoSomething(a);
            DoSomething(b);
            DoSomething(c);
        }

        public static void DoSomething(A a)
        {
            a.Foo(0);
        }
    }

    class A
    {
        public virtual void Foo(int a)
        {
            Console.WriteLine("A.Foo");
        }
        //...
    }

    class B : A
    {
        public override void Foo(int a)
        {
            Console.WriteLine("B.Foo");
        }
        //...
    }

    class C : B
    {
        public override void Foo(int a)
        {
            Console.WriteLine("C.Foo");
        }
        //...
    }
}
```

Burada DoSomething metodunun parametre değişkeni olan a'nın dinamik türü ilk çağrıda A, sonrakinde B ve sonrakinde de C'dir.

Eğer referansın dinamik türüne ilişkin sınıfta ilgili sanal metot override edilmemişse yukarıya doğru bu sanal metodun override edildiği ilk taban sınıfın sanal metodu çağrılır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            A a = new C();

            a.Foo(0);       // B.Foo çağrılır
        }
    }

    class A
    {
        public virtual void Foo(int a)
        {
            Console.WriteLine("A.Foo");
        }
        //...
    }

    class B : A
    {
        public override void Foo(int a)
        {
            Console.WriteLine("B.Foo");
        }
        //...
    }

    class C : B
    {
        //...
    }
}
```

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 A());

            foreach (A a in al)
                a.Foo();
        }
    }

    class A
    {
        public virtual void Foo()
        {
            Console.WriteLine("A.Foo");
        }
        //...
```

```
    }

    class B : A
    {
        public override void Foo()
        {
            Console.WriteLine("B.Foo");
        }
        //...
    }

    class C : B
    {
        public override void Foo()
        {
            Console.WriteLine("C.Foo");
        }
        //...
    }
}
```

Burada ArrayList türemişl sınıf referanslarını object dizisinde tutup bize vermektedir. foreach döngüsünde elde ettiğimiz referansların dinamik türleri sırasıyla A, B, C, A olacaktır.

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Plus plus = new Plus();
            Multiply mul = new Multiply();
            Divide div = new Divide();
            Minus minus = new Minus();

            Test(plus, 10, 2);
            Test(mul, 10, 2);
            Test(div, 10, 2);
            Test(minus, 10, 2);
        }

        public static void Test(Operator op, double a, double b)
        {
            double result;

            result = op.Calc(a, b);
            Console.WriteLine(result);
        }
    }

    class Operator
    {
        public virtual double Calc(double a, double b)
        {
            return 0;
        }
        //...
    }

    class Plus : Operator
    {
        public override double Calc(double a, double b)
        {
            return a + b;
        }
```

```
        //...
    }

    class Minus : Operator
    {
        public override double Calc(double a, double b)
        {
            return a - b;
        }
        //...
    }

    class Multiply : Operator
    {
        public override double Calc(double a, double b)
        {
            return a * b;
        }
        //...
    }

    class Divide : Operator
    {
        public override double Calc(double a, double b)
        {
            return a / b;
        }
        //...
    }
}
```

Burada Operator sınıfından türetilmiş olan sınıfların Calc metotları farklı işlemler yapacak biçimde override edilmiştir. Taban sınıftaki Calc metodu türemiş sınıflarda hesaplama yapmaktadır ancak yapılan hesaplamalar türemiş sınıflarda farklılık göstermektedir.

object sınıfının aşağıdaki gibi bir virtual metodu vardır:

```
public virtual string ToString()
```

Bu metot .NET'in sınıflarının ve yapılarının büyük çoğunluğunda override edilmiştir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            object o;
            Sample s = new Sample();

            o = s;
            Console.WriteLine(o.ToString());        // Sample.ToString çağrılır.
        }
    }

    class Sample
    {
        public override string ToString()
        {
            return "this is a test";
        }
        //...
    }

}
```

Örneğin:

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();

            al.Add(new Sample());
            al.Add(new Mample());
            al.Add(new Tample());

            foreach (object o in al)
                Console.WriteLine(o.ToString());
        }
    }

    class Sample
    {
        public override string ToString()
        {
            return "This is a Sample";
        }
    }

    class Mample
    {
        public override string ToString()
        {
            return "This is a Mample";
        }
    }

    class Tample
    {
        public override string ToString()
        {
            return "This is a Tample";
        }
    }

}
```

Pekiyi yukarıdaki örneklerde biz ToString metodunu override etmeseydik ne olurdu? Bu durumda object sınıfının ToString metodu çağrılırdı değil mi? Object sınıfının ToString metodu "reflection" denilen mekanizmayla referansın dinamik türüne ilişkin sınıfın ismini elde eder ve onu bir yazı olarak verir.

Yapılar her ne kadar türetmeye kapalıysa da Object sınıfından gelen sanal metotları override edebilirler. .NET'te pek çok ve yapıda ToString metotları o sınıf ya da yapının tuttuğu değerleri bize yazı olarak vermektedir. Örneğin int rürünün (yani Int32 türünün) ToString metodu bize o int değeri yazı olarak verir. double türünün ToString metodu bize o double değişkenin içerisineki değeri yazı olarak verir. DateTime yapısının ToString metodu bize o nesnenin tuttuğu tarih ve zamanı yazı olarak verir. Örneğin:

```
using System;

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

```
            Console.WriteLine("Sayı = " + a.ToString());
        }
    }

}
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            object[] objs = new object[] { 123, 23.56, new Sample(), DateTime.Today };

            foreach (object o in objs)
            {
                string s = o.ToString();
                Console.WriteLine(s);
            }
        }
    }

    class Sample
    {
        public override string ToString()
        {
            return "this is a test";
        }
    }
}
```

Aynı işlemi ArrayList ile de yapabilirdik:

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();

            al.Add(123);
            al.Add(23.56);
            al.Add(new Sample());
            al.Add(DateTime.Today);

            foreach (object o in al)
            {
                string s = o.ToString();
                Console.WriteLine(s);
            }
        }
    }

    class Sample
    {
        public override string ToString()
        {
            return "this is a test";
        }
    }
}
```

Bu örnekte en çok dikkat çeken nokta şudur: ArrayList içerisindeki nesneler farklı türlendir. Yani heterojendir. Biz ToString metotlarını çağırdığımızda object referansının gösterdiği nesnenin dinamik türü ne ise onun ToString metodu çağrılır. Çokbiçimlilik farklı türlere aynı türmüş muamelesi yapmaktır

Console. sınıfının temel türlere ilişkin parametrelere sahip olan Write ve WriteLine metotlarının yanı sıra object parametreli Write ve WriteLine metotları da vardır. Bu metotlar aldıkları object referansı ile ToString sanal metodunu çağırıp elde ettikleri yazıyı ekrana basmaktadırlar. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            DateTime dt = new DateTime(1995, 2, 20);
            Sample s = new Sample();

            Console.WriteLine(dt);
            Console.WriteLine(s);
        }
    }

    class Sample
    {
        public override string ToString()
        {
            return "this is a Sample";
        }
    }
}
```

C# standartlarına göre + operatörünün bir operandı string türünden fakat diğeri değilse derleyici diğer operand ile ToString metodunu çağırır elde ettiği yazıyı string operamdıyla birleştirip yeni yazıyı elde eder. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s = new Sample();
            string str;

            str = "Text: " + s;     // Eşdeğeri str = "Text: " + s.ToString();
            Console.WriteLine(str);
        }
    }

    class Sample
    {
        public override string ToString()
        {
            return "This is a Sample";
        }
    }
}
```

Örneğin:

```
using System;

namespace CSD
```

```
{
    class App
    {
        public static void Main()
        {
            int a = 10;

            Console.WriteLine("Sayı: = " + a);
        }
    }
}
```

ToString metodunun tam ters işlemini static Parse metotları yapmaktadır. Parse metotları bizden bir yazı alıp onu ilgili türden değere dönüştürür. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            string s = "123";
            int val;

            val = int.Parse(s) + 1;

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

Eğer yazı ilgili tür ile ifade edilemeyen karakterlere ya da değere sahipse exception oluşur ve program çöker. Artık klavyeden değerin aslında nasıl okunduğunu anlayabiliriz. Örneğin:

```
using System;

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

            Console.Write("Bir değer giriniz:");
            val = int.Parse(Console.ReadLine());
            Console.WriteLine(val);
        }
    }
}
```

Burada aslında klavyeden bir yazı okunup o yazı int yapısının static Parse metodu ile sayıya dönüştürülmektedir.

Bu tür sayı-yazı dönüştürmeleri için Convert isimli bir sınıf da vardır. Convert sınıfının ToXXX (burada XXX bir türdür) isimli static metotları her temel türden parametreyi alıp bize XXX türünden değe rolarak verir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
```

```
        public static void Main()
        {
            string s = "123";
            int val;

            val = Convert.ToInt32(s);
            Console.WriteLine(val); ;
        }
    }
}
```

Benzer biçimde Convert sınıfının ToString static metotlarıyla sayıları da yazılara dönüştürebiliriz. Örneğin:

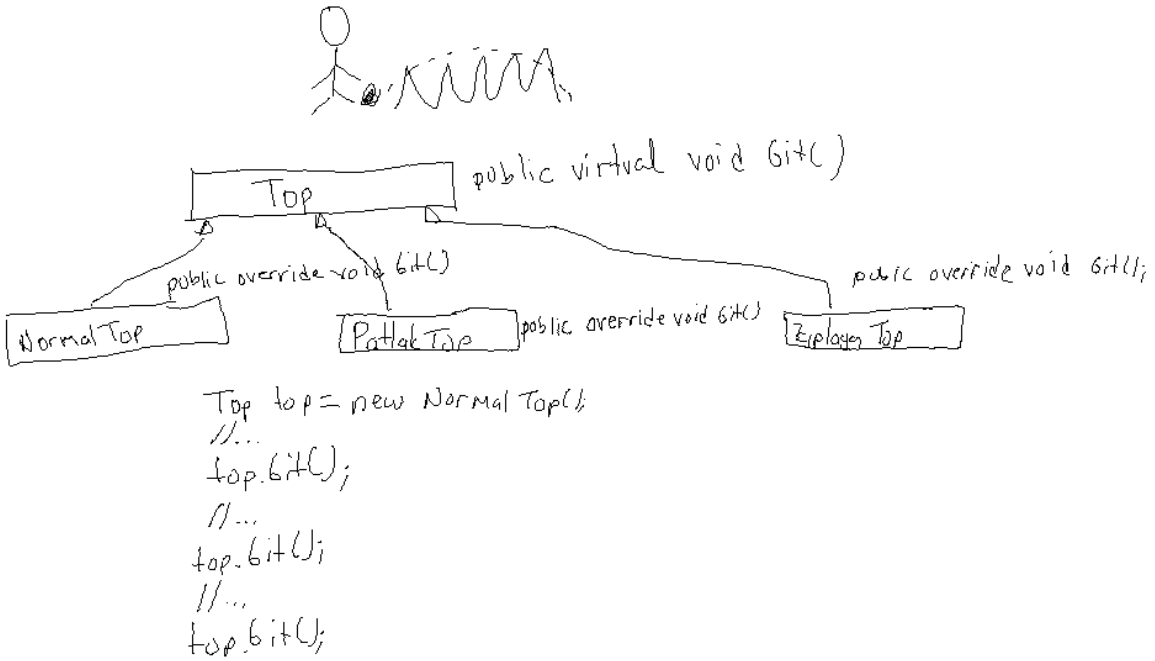
```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int val = 123;
            string str;

            str = Convert.ToString(val);
            Console.WriteLine(str); ;
        }
    }
}
```

Çokbiçimliliğe Çeşitli Örnekler

Çokbiçimliliğe ilişkin örneklerin hepsinde tepede bir taban sınıf (ya da ileride göreceğimiz gibi arayüz) bulunur. Bu taban sınıftan sınıflar türetilip taban sınıfın sanal metotları türemiş sınıflarda override edilir. Biz de taban sınıf referansını kullanrak kodumuzu türden bağımsız bir biçimde yazarız. Taban sınıf referansı ile sanal metotlar çağrıldığında onun dinamik türüne ilişkin türemiş sınıfın override edilmiş metotları metotları çağrılacaktır.

Bir projede değişebilecek birtakım öğeler varsa onlara doğrudan değil taban sınıf yoluyla çokbiçimli olarak erişmek uygun olur. Böylece onlar değişseler bile biz kodumuzu değiştirmek zorunda kalmayız.

1) Top Oyunu Programı: Topla oynanan bir oyun programı yazacak olalım. Oyunun içerisindeki top normal bir top olabilir, zıplayan bir top olabilir ya da patlak bir top olabilir. Oyunda topu değiştirdiğimizde kodda önemli bir değişiklik yapmak istemeyelim. Topun gitmesi çokbiçimli bir eylemdir. Yani her top gider fakat kendine göre gider. İşte topu temsil eden bir Top taban sınıfı oluşturabiliriz. Diğer sınıfları bu sınıftan türetebiliriz. Top sınıfında Git isimli bir virtual metot olabilir. Bu da diğer sınıflarda override edilmiş olabilir:



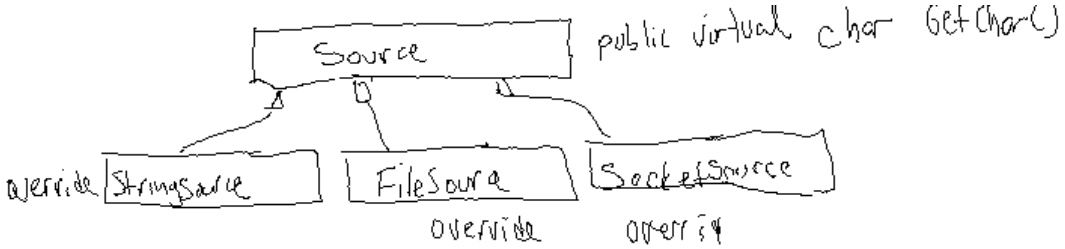
Buırada normalde NormalTop sınıfının Git metodu çağrılır. O da topu normal olarak hareket ettirir. Ancak eğer istenirse top referansına new PatlakTop() ifadesi ile PatlakTop nesnesi atanabilir. Bu durumda Git'ler artık PatlakTop sınıfının Git'leri olacaktır. Görüldüğü gibi programın belirli kısımları Top kavramına dayalı olarak türden bağımsız biçimde oluşturulmuştur. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            NormalTop nt = new NormalTop();
            ZıplayanTop zt = new ZıplayanTop();
            PatlakTop pt = new PatlakTop();

            Test(nt);
            Test(zt);
            Test(pt);
        }

        public static void Test(Top top)
        {
            top.Git();
            //...
            top.Git();
            //...
            top.Git();
            //...
        }
    }

    class Top
    {
        public virtual void Git()
        {
            Console.WriteLine("Top gidiyor...");
        }
        //...
    }

    class PatlakTop : Top
    {
        public override void Git()
```

```
        {
            Console.WriteLine("Patlak top gidiyor...");
        }
        //...
    }

    class ZıplayanTop : Top
    {
        public override void Git()
        {
            Console.WriteLine("Top zıplayarak gidiyor...");
        }
        //...
    }

    class NormalTop : Top
    {
        public override void Git()
        {
            Console.WriteLine("Top normal gidiyor...");
        }
        //...
    }
}
```

2) Parser Örneği: Bir yazının belirli karakterlerden parçalara ayrılmasına parse işlemi (parsing) denir. Böyle bir Parser sınfını yazacak olalım. Parser sınıfı yazıyı herhangi bir kaynaktan alabilecek biçimde yazılabilir. Parse etmek için ilgili kaynaktan karakter karakter okuma yapmak gerekir. Burada çokbiçimli tema şöyle kullanılabilir: Bilginin alınacağı kaynak değişebilmektedir. Çokbiçimlilik yazacağımız kodun değişen kaynaklardan etkilenmemesini sağlayabilir. Yazıların alıncağı söz konusu kaynak Source isimli bir sınıfla temsil edilebilir. Bu sınıfın bir virtual GetChar metodu olabilir. Bu sınıftan türetilen sınıflar bu metodu override edebilirler. GetChar çokbiçimli bir metottur. Çünkü GetChar kaynaktan karakter verir ancak kaynağa göre bu karakteri değişik biçimde vermektedir. Örneğin:

```
class Parser
{
	private Source m_source;

	public Parser(Source source)
	{
		m_source = source;
		//...
	}

	public void DoParse()
	{
		char ch;
		//...
		ch = m_source.GetChar();
		//...
		ch = m_source.GetChar();
		//...
		ch = m_source.GetChar();
		//...
	}
	//...
}
```



Parser sınıfı içerisindeki DoParse metodu hangi kaynak için GetChar metodunu çağırmaktadır? m_source'un dinamik türü ne ise ona ilişkin sınıfınkini değil mi?

```
//...
FileSource fs = new FileSource("a.txt");
Parser parser = new Parser(fs);
parser.DoParse();
```

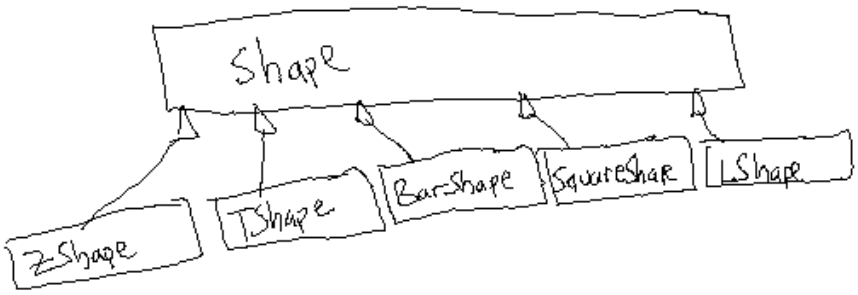
Biz böylece kaynaktan bağımsız yani her kaynak için çalışabilen bir Parser sınıf yazmış olduk. Parser sınıfına hangi kaynak sınıfını verirsek Parser o kaynaktan okuma yapacaktır. Hatta seneler sonra yeni bir kaynak sınıfını daha Source sınıfından türebiliriz. Ve Parser sınıfına onu verebiliriz. "Önceden yazılmış kodların sonradan yazılan kodları çağırması" bu anlamdadır. Aşağıda Parser sınıfı temsili olarak yazılmıştır:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileSource fs = new FileSource();
            StringSource ss = new StringSource();

            Parser parser = new Parser(fs);
            parser.DoParse();

            Parser parser2 = new Parser(ss);
            parser2.DoParse();
        }
    }

    class Source
    {
        public virtual char GetChar()
        { return ' '; }
    }

    class FileSource : Source
    {
        public override char GetChar()
        {
            return 'f';
        }
        //...
    }

    class StringSource : Source
    {
        public override char GetChar()
        {
            return 's';
        }
        //...
    }

    class NetworkSource : Source
    {
        public override char GetChar()
        {
            return 'n';
        }
        //...
    }

    class Parser
```

```
    {
        private Source m_source;

        public Parser(Source source)
        {
            m_source = source;
        }

        public void DoParse()
        {
            char ch;

            //...
            ch = m_source.GetChar();
            Console.WriteLine(ch);
            //...
            ch = m_source.GetChar();
            Console.WriteLine(ch);
            //...
            ch = m_source.GetChar();
            Console.WriteLine(ch);
            //...
        }
        //...
    }
}
```

Anahtar Notlar: NYPT'de proje içerisindeki değişne öğeler belirlenir. Değişen öğelere doğrudan değil çokbiçimli olarak erişilir. Böylece sınıf o değişkenlikten etkilenmez.

3) Tetris Örneği: Bir tetris oyununda çeşitli şekiller düşmektedir. Tüm şekillerin zemin rengi gibi, konumu gibi ortak birtakım özellikleri vardır. Bu ortak özellikler tepedeki bir Shape sınıfında toplanabilir ve diğer şekiller bundan türetilmiş sınıflarla temsil edilebilir.



Programın kendisini Tetris isimli bir sınıf temsil ediyor olalım. Şekillerin düşmesi çokbiçimli bir eylemdir. Yani her şekil düşer fakat kendine göre düşer. Her şekil sola, sağa hareket eder fakat kendine göre bu hareketi yapmaktadır. İşte Shape sınıfında şekilleri sağa, sola hareket ettiren, aşağıya düşüren ve döndüren dört sanal metot bulunuyor olabilir:

```
public virtual void MoveLeft()
public virtual void MoveRight()
public virtual void MoveDown()
public virtual void Rotate()
```

Bu metotlar türemiş sınıflarda override edilmiştir. Böylece şekillerin düşürülmesi ve hareket ettirilmesi türden bağımsız olarak yapılabilir. Örneğin Tetris sınıfının Run metodu temsili olarak şuna benzemektedir:

```
using System;

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

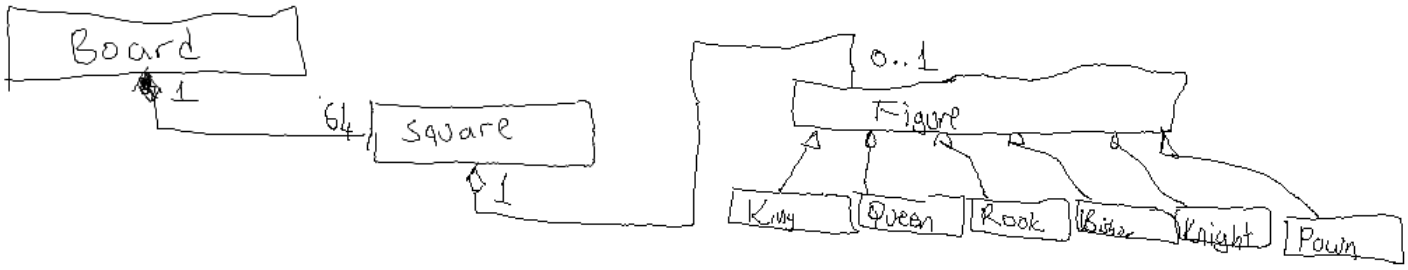
```
        Tetris tetris = new Tetris();
        tetris.Run();
    }
}

class Tetris
{
    private Random m_rand;

    public Tetris()
    {
        m_rand = new Random();
    }

    public void Run()
    {
        Shape shape;

        for (;;)
        {
            shape = GetRandomShape();
            for (int i = 0; i < 30; ++i)
            {
                shape.MoveDown();
                System.Threading.Thread.Sleep(300);
                if (Console.KeyAvailable)
                {
                    switch (Console.ReadKey(false).Key)
                    {
                        case ConsoleKey.LeftArrow:
                            shape.MoveLeft();
                            break;
                        case ConsoleKey.RightArrow:
                            shape.MoveRight();
                            break;
                        case ConsoleKey.Enter:
                            shape.Rotate();
                            break;
                        case ConsoleKey.Q:
                            goto EXIT;

                    }
                }
            }
        }
    EXIT:
        ;
    }

    private Shape GetRandomShape()
    {
        Shape shape = null;

        switch (m_rand.Next(5))
        {
            case 0:
                shape = new BarShape();
                break;
            case 1:
                shape = new SquareShape();
                break;
            case 2:
                shape = new TShape();
                break;
            case 3:
                shape = new LShape();
                break;
            case 4:
                shape = new ZShape();
                break;
```

```
        }

        return shape;
    }
}

class Shape
{
    public virtual void MoveDown()
    { }

    public virtual void MoveLeft()
    { }

    public virtual void MoveRight()
    { }

    public virtual void Rotate()
    { }
}

class BarShape : Shape
{
    public override void MoveDown()
    {
        Console.WriteLine("BarShape.MoveDown");
    }

    public override void MoveLeft()
    {
        Console.WriteLine("<<BarShape.MoveLeft>>");
    }

    public override void MoveRight()
    {
        Console.WriteLine("<<BarShape.MoveRight>>");
    }

    public override void Rotate()
    {
        Console.WriteLine("<<BarShape.Rotate>>");
    }
}

class ZShape : Shape
{
    public override void MoveDown()
    {
        Console.WriteLine("ZShape.MoveDown");
    }

    public override void MoveLeft()
    {
        Console.WriteLine("<<ZShape.MoveLeft>>");
    }

    public override void MoveRight()
    {
        Console.WriteLine("<<ZShape.MoveRight>>");
    }

    public override void Rotate()
    {
        Console.WriteLine("<<ZShape.Rotate>>");
    }
}

class TShape : Shape
{
    public override void MoveDown()
```

```
        {
            Console.WriteLine("TShape.MoveDown");
        }

        public override void MoveLeft()
        {
            Console.WriteLine("<<TShape.MoveLeft>>");
        }

        public override void MoveRight()
        {
            Console.WriteLine("<<TShape.MoveRight>>");
        }

        public override void Rotate()
        {
            Console.WriteLine("<<TShape.Rotate>>");
        }
    }

    class LShape : Shape
    {
        public override void MoveDown()
        {
            Console.WriteLine("LShape.MoveDown");
        }

        public override void MoveLeft()
        {
            Console.WriteLine("<<LShape.MoveLeft>>");
        }

        public override void MoveRight()
        {
            Console.WriteLine("<<LShape.MoveRight>>");
        }

        public override void Rotate()
        {
            Console.WriteLine("<<LShape.Rotate>>");
        }
    }

    class SquareShape : Shape
    {
        public override void MoveDown()
        {
            Console.WriteLine("SquareShape.MoveDown");
        }

        public override void MoveLeft()
        {
            Console.WriteLine("<<SquareShape.MoveLeft>>");
        }

        public override void MoveRight()
        {
            Console.WriteLine("<<SquareShape.MoveRight>>");
        }

        public override void Rotate()
        {
            Console.WriteLine("<<SquareShape.Rotate>>");
        }
    }

}
```

4) Satranç Tahtası Örneği: Bir satranç programında bir tahta, tahtanın üzerinde kareler ve karelerin üzerinde

de taşlar vardır. Tüm taşların ortak özellikleri Figure isimli bir sınıfla temsil edilmiştir. Bu sınıftan King, Queen, Rook, Bishop, Knight, Pawn sınıfları türetilmiştir. Uygulamanın UML sınıf diagramı şöyle olabilir:



Taşların hareket etmesi çokbiçimli bir eylemdir. Her taş hareket eder fakat değişik biçimde hareket eder. (Örneğin at L çizer, fil çapraz gider, kale düz gider). İşte böyle bir tahta uygulamasında fare ile bir taş hareket ettirildiğinde onun hangi taş olduğunun bilinmesine gerek yoktur. Figure sınıfının GetValidMoves isimli bir sanal metodu olabilir. Bu metot bir taşın o anda gidebileceği kareleri bize bir ArrayList içerisine yerleştirerek verebilir:

```
public virtual ArrayList GetValidMoves()
```

Bu metot türemiş sınıflarda override edilerek her taşın kendi gidebileceği kareleri verecek biçimde düzenlenebilir. Böylece fare ile bir taşa tıkladığımızda o taşın hangi taş olduğunu bilmeden GetValidMoves metodunu çağırarak onun gidebileceği karelerin yerlerini elde etmiş oluruz.

NYPT'de Test İşlemleri

İdeal olarak NYPT'de daha önce yazılmış olan bir kod değiştirilmez. Hep ekleme yapılarak proje devam ettirilir. Birtakım öğeler iyi test edilmişse artık onlar değişmediğine göre bir sorun oluştuğunda hata yeni eklenen öğelerde aranır. Bu da test aşamasını sağlamlaştırmakta ve kolaylaştırmaktadır. Klasik test yönteminde (şelale modelinde) kodda bir değişiklik yapıldığında tüm testlerin yinelenmesi gerekir. Çünkü programcılar bir yeri düzeltirken yanlışlıkla başka yerleri bozabilmektedir. Oysa NYPT'de hep ekleme yapıldığı için yalnızca son eklenen öğeler test edilirler.

Abstract Metotlar ve Sınıflar

Bazı çokbiçimli uygulamalarda türetme şemasının yukarısında bulunan taban sınıfın sanal metotları aslında hiç çağrılmaz. Bu sınıf türden bağımsız işlemler yapabilmek için tepede bulundurulmuştur. Yani aslında o metotlar çağrıldığında zaten hep dinamik türlere ilişkin override edilmiş metotlar çağrılır. İşte bu tür durumlarda o metotlara boşuna gövde yerleştirmeye gerek olmaz. Örneğin Tetris programında Shape sınıfının MoveDown, MoveLeft, MoveRight ve Rotate metotları çağrıldığında aslında hep dinamik türe ilişkin türemiş sınıfların ilgili metotları çalıştırılmaktadır. Yani Shape sınıfının bu metotları aslında hiç çalıştırılmamaktadır. O halde bunların gövdelerinin bulunmasına gerek yoktur. İşte bunlar abstract yapılabilir. Örneğin:

```
abstract class Shape
{
    //...
    public abstract void MoveLeft();
    public abstract void MoveRight();
    public abstract void MoveDown();
    public abstract void Rotate();
}
```

Metot bildiriminde abstract anahtar sözcüğü metodun sanal fakat gövdesiz oluduğunu belirtir. Abstract metotlar sanallığı başlatmak için kullanılabilirler. En az bir abstract elemana sahip sınıfa abstract sınıf denir. Abstract sınıflarda abstract anahtar sözcüğü ayrıca sınıf bildiriminin başına da getirilmek zorundadır. Metot bildiriminde abstract anahtar sözcüğü ile erişim belirleyici anahtar sözcükler yer değiştirmeli olarak yazılabilirler (yani public abstract ya da abstract public aynı anlamdadır.) abstract anahtar sözcüğüyle virtual anahtar sözcüğü bir arada kullanılamaz. Zaten abstract anahtar sözcüğü virtual olma durumunu da içermektedir. Abstract sınıflar

veri elemanlarına, abstract olmayan metotlara, static metotlara sahip olabilirler. Fakat sınıfın en az bir elemanı abstract ise sınıf abstract'tır.

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            A a = new B();

            a.Foo();        // B.Foo çağrılır
            a.Bar();        // A.Bar çağrılır
        }
    }

    abstract class A
    {
        int m_a;

        public abstract void Foo();
        public void Bar()
        {
            Console.WriteLine("A.Bar");
        }
        //...
    }

    class B : A
    {
        public override void Foo()
        {
            Console.WriteLine("B.Foo");
        }
        //...
    }
}
```

Abstract sınıflar türünden referanslar bildirilebilir. Ancak new operatörüyle nesneler yaratılamaz. (Eğer yaratılabilseydi olmayan bir metodun çağrılması gibi potansiyel bir durum oluşurdu). O halde abstract sınıf türünden referanslara onların türemiş sınıf referansları atanabilir.

Abstract bir sınıftan türetilen sınıflarda taban abstract sınıfın tüm abstract elemanları override edilmelidir. Eğer bu yapılmazsa türemiş sınıf da abstract olur. Bu durumda türemiş sınıf bildiriminin başına da abstract anahtar sözcüğü getirilmek zorundadır. Tabii bu durumda türemiş sınıf türünden de new operatörü ile nesneler yaratılamaz. Örneğin:

```
abstract class A
{
    int m_a;

    public abstract void Foo();
    public void Bar()
    {
        Console.WriteLine("A.Bar");
    }
    //...
}

abstract class B : A
{
    //...
```

```
}
```

Burada özel bir durumu belirtmek gerekir. Taban A sınıfının bir grup abstract elemanı ondan türetilmiş B'de override edilmiş olsun. Geri kalanı da B'den türetilmiş C'de override edilmiş olsun. Bu durumda B abstract olur, fakat C abstract değildir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            A a;        // geçerli

            a = new C();

            a.Foo();        // B.Foo çağrılır
            a.Bar();        // C.Bar çağrılır
        }
    }

    abstract class A
    {
        public abstract void Foo();
        public abstract void Bar();
        //...
    }

    abstract class B : A                // abstract
    {
        public override void Foo()
        {
            Console.WriteLine("B.Foo");
        }
        //...
    }

    class C : B                         // abstract değil
    {
        public override void Bar()
        {
            Console.WriteLine("B.Bar");
        }

    }
}
```

Pekiyi biz bir sınıfın abstract olduğunu gördüğümüzde ne düşünmeliyiz? Öncelikle bu sınıf türünden bir nesne yaratamayacağımızı düşünmeliyiz. Sonra bu sınıfın tek başına bir işe yaramayacağını mutlaka bundan türetilmiş ve abstract elemanların override edildiği sınıfların var olması gerektiğini anlalıyız. abstract sınıf türden bağımsız işlem yapmak için bildirilmiştir.

Pekiyi biz sanal metotları hep virtual yerine abstract yaparsak ne olur? Bu durumda biz bu sınıf türünden nesneler yaratamaz hale geliriz. Ayrıca bu sınıftan türettiğimiz her sınıfta bu metotları override etmek zorunda kalırız. Örneğin object sınıfının ToString metodu virtual değil de abstract olsaydı her sınıfta biz bunu override etmek zorunda kalırdık. "virtual" metotlar "sen override etmezsen bu çağrılır" anlamına gelmektedir. Halbuki "abstract" metotlar "sen override etmelisin" anlamına gelir.

Aslında bir sınıfın hiçbir abstract elemanı olmasa bile eğer biz sınıf bildiriminin başına yine abstract anahtar sözcüğünü yerleştirirsek sınıf yine abstract olur. Bu durumda biz bu sınıf türünden new operatörüyle nesneler yaratamayız.

Yukarıda verdiğimiz Tetris örneğinde taban Shape sınıfının abstract sınıf olması ve MoveDown, MoveLeft,

MoveRight, Rotate metotlarının da abstract metotlar olması anlamlıdır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Tetris tetris = new Tetris();
            tetris.Run();
        }
    }

    class Tetris
    {
        private Random m_rand;

        public Tetris()
        {
            m_rand = new Random();
        }

        public void Run()
        {
            Shape shape;

            for (;;)
            {
                shape = GetRandomShape();
                for (int i = 0; i < 30; ++i)
                {
                    shape.MoveDown();
                    System.Threading.Thread.Sleep(300);
                    if (Console.KeyAvailable)
                    {
                        switch (Console.ReadKey(false).Key)
                        {
                            case ConsoleKey.LeftArrow:
                                shape.MoveLeft();
                                break;
                            case ConsoleKey.RightArrow:
                                shape.MoveRight();
                                break;
                            case ConsoleKey.Enter:
                                shape.Rotate();
                                break;
                            case ConsoleKey.Q:
                                goto EXIT;

                        }
                    }
                }
            }
        EXIT:
            ;
        }

        private Shape GetRandomShape()
        {
            Shape shape = null;

            switch (m_rand.Next(5))
            {
                case 0:
                    shape = new BarShape();
                    break;
                case 1:
                    shape = new SquareShape();
```

```
                break;
            case 2:
                shape = new TShape();
                break;
            case 3:
                shape = new LShape();
                break;
            case 4:
                shape = new ZShape();
                break;
        }

        return shape;
    }
}

abstract class Shape
{
    public abstract void MoveDown();
    public abstract void MoveLeft();
    public abstract void MoveRight();
    public abstract void Rotate();
}

class BarShape : Shape
{
    public override void MoveDown()
    {
        Console.WriteLine("BarShape.MoveDown");
    }

    public override void MoveLeft()
    {
        Console.WriteLine("<<BarShape.MoveLeft>>");
    }

    public override void MoveRight()
    {
        Console.WriteLine("<<BarShape.MoveRight>>");
    }

    public override void Rotate()
    {
        Console.WriteLine("<<BarShape.Rotate>>");
    }
}

class ZShape : Shape
{
    public override void MoveDown()
    {
        Console.WriteLine("ZShape.MoveDown");
    }

    public override void MoveLeft()
    {
        Console.WriteLine("<<ZShape.MoveLeft>>");
    }

    public override void MoveRight()
    {
        Console.WriteLine("<<ZShape.MoveRight>>");
    }

    public override void Rotate()
    {
        Console.WriteLine("<<ZShape.Rotate>>");
    }
}
```

```
    class TShape : Shape
    {
        public override void MoveDown()
        {
            Console.WriteLine("TShape.MoveDown");
        }

        public override void MoveLeft()
        {
            Console.WriteLine("<<TShape.MoveLeft>>");
        }

        public override void MoveRight()
        {
            Console.WriteLine("<<TShape.MoveRight>>");
        }

        public override void Rotate()
        {
            Console.WriteLine("<<TShape.Rotate>>");
        }
    }

    class LShape : Shape
    {
        public override void MoveDown()
        {
            Console.WriteLine("LShape.MoveDown");
        }

        public override void MoveLeft()
        {
            Console.WriteLine("<<LShape.MoveLeft>>");
        }

        public override void MoveRight()
        {
            Console.WriteLine("<<LShape.MoveRight>>");
        }

        public override void Rotate()
        {
            Console.WriteLine("<<LShape.Rotate>>");
        }
    }

    class SquareShape : Shape
    {
        public override void MoveDown()
        {
            Console.WriteLine("SquareShape.MoveDown");
        }

        public override void MoveLeft()
        {
            Console.WriteLine("<<SquareShape.MoveLeft>>");
        }

        public override void MoveRight()
        {
            Console.WriteLine("<<SquareShape.MoveRight>>");
        }

        public override void Rotate()
        {
            Console.WriteLine("<<SquareShape.Rotate>>");
        }
    }

}
```

Sınıfların virtual ve abstract Propery Elemanları

Aslında biz property elemanlarını metot gibi düşünebiliriz. Dolayısıyla property'ler de virtual ve abstract olabilirler. Eğer bir property virtual ise biz onu override edebiliriz ya da etmeyebiliriz. Ancak read-only bir virtual property read-only olarak, write-only bir virtual property write-only olarak read/write bir virtual property read-only, write-only ya da read/write olarak override edilebilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            A a;
            B b = new B();

            a = b;
            a.Val = 10;
            Console.WriteLine(a.Val);
        }
    }

    class A
    {
        private int m_a;

        public virtual int Val
        {
            get { return m_a; }
            set { m_a = value; }
        }
        //...
    }

    class B : A
    {
        private int m_b;

        public override int Val
        {
            get { return m_b; }
            set { m_b = value; }
        }
        //..
    }
}
```

Property'ler abstract da olabilirler. Bu durumda get ve set bölümlerinin gövdesi olmaz, noktalı virgül ile bu bölümler kapatılmalıdır. Örneğin:

```
public abstract int Val
{
    get;
    set;
}
```

abstract property read-only ise biz onu read-only olarak, write-only ise biz onu write-only olarak read/write ise biz onu read/write olarak override edebiliriz. Örneğin:

```
using System;

namespace CSD
{
```

```
    class App
    {
        public static void Main()
        {
            A a = new B();

            a.Val = 10;     // B.Val çalıştırılır
            Console.WriteLine(a.Val);       // B.Val çalıştırılır
        }
    }

    abstract class A
    {
        private int m_a;

        public abstract int Val
        {
            get;
            set;
        }
        //...
    }

    class B : A
    {
        private int m_b;

        public override int Val
        {
            get { return m_b; }
            set { m_b = value; }
        }
        //..
    }
}
```

Tabii read/write bir abstract property yalnızca read-only ya da yalnızca write-only olarak override edilebilir. Ancak bu durumda override işleminin yapıldığı sınıf da taban sınıfın tüm abstract elemanlarının override edilmemiş olduğu nedeniyle abstract olacaktır.

null Referans Kavramı

null referans hiçbir nesnenin adresi olmayan boş bir adres belirtir. C#'ta null referans null anahtar sözcüğü ile temsil edilmektedir. null referans her türden referansa doğrudan atanabilir. Örneğin:

```
Sample  s;
Random r;
string str;

s = null;           // geçerli
r = null;           // geçerli
str = null;         // geçerli
```

Bir referansın içerisinde null değeri olup olmadığı == ya da != operatörleriyle sorgulanabilir. Örneğin:

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

ya da örneğin:

```
if (s != null)
{
      //...
```

```
}
```

null referans kategori olarak değer türlerine ilişkin değişkenlere atanamaz. Yalnızca referans değişkenlerine atanabilir. Örneğin:

```
int a = null;       // error!
```

Anımsanacağı gibi bir değişkene henüz değer atamadan onun içerisindeki değerler kullanılmak istenirse bu durum derleme aşamasında error oluşturmaktadır. Fakat içerisinde null referans olan bir değişken kullanılarsa derleme aşamasından başarıyla geçilir. Ancak programın çalışma zamanı sırasında değişkenin kullanıldığı noktada "exception" oluşur (NullReferenceException). Örneğin:

```
using System;

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

            Console.WriteLine(s.Length);        // error değil, exception oluşur
        }
    }
}
```

Bazen null referansı mecburen kullanmak zorunda kalırız. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Random r = new Random();
            string name /* = null */ ;

            switch (r.Next(5))
            {
                case 0:
                    name = "Ali";
                    break;
                case 1:
                    name = "Veli";
                    break;
                case 2:
                    name = "Selami";
                    break;
                case 3:
                    name = "Ayşe";
                    break;
                case 4:
                    name = "Fatma";
                    break;
            }

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

C# derleyicisi bir metodun ne yaptığını bilmez. Çünkü metotlar değiştirilebilir, kütüphanelerden çıkartılabilir vs. Dolayısıyla C# derleyicisi metotların yalnızca parametrik yapılarını ve geri dönüş değerlerini bilerek birtakım

kontrolleri yapmaktadır. Metotların ne yaptığını yalnızca programcılar bilmektedir. Yukarıdaki kodda name referansına ilkdeğer atanmazsa error oluşacaktır. Çünkü derleyici değer almama olasılığı olan bir değişkenin kullanıldığını gördüğünde kodu derlemez. Bu durum error oluşturmaktadır. Tabi eğer derleyici Next metodunun ne yaptığını bilseydi belki bizi affederdi. Fakat bunu bilmemektedir. İşte biz bu tür durumlarda böyle bir referansa null değeri vererek onun değer atanmış gibi işlem görmesini sağlayabiliriz.

Bazen null referans başarısızlığı anlatmak için de kullanılmaktadır. Örneğin bir metodun geri dönüş değeri bir referanstır. Ancak metot başarısız da olabilmektedir. İşte bu durumda metot null referansa geri dönebilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            string name;
            int no;

            Console.Write("Lütfen bir numara giriniz:");
            no = int.Parse(Console.ReadLine());

            name = Student.ConvertNumberToName(no);
            if (name == null)
                Console.WriteLine("Böyle bir öğrenci numarası yok!");
            else
                Console.WriteLine(name);
        }
    }

    class Student
    {
        public static string ConvertNumberToName(int number)
        {
            int[] numbers = { 123, 567, 345, 765, 789 };
            string[] names = { "Ali Serçe", "Kaan Aslan", "Necati Ergin", "Güray Sönmez", "Oğuz
Karan" };

            for (int i = 0; i < numbers.Length; ++i)
                if (number == numbers[i])
                    return names[i];

            return null;
        }
    }
}
```

Anımsanacağı gibi bir sınıf türünden nesne new operatörü ile yaratıldığında new operatörü sınıfın static olmayan veri elemanları için heap'te tahsisat yapıp oradaki yapı veri elemanlarını sıfırladıktan sonra başlangıç metodunu çağırıyordu. İşte bir sınıf nesnesi new ile yaratıldığında sınıfın referans türünden veri elemanlarına da null değeri atanmaktadır. Örneğin:

```
class Sample
{
    public string Msg;

    public Sample()
    { }
    //...
}

Sample s = new Sample();
```

Burada s.Msg elemanında null değeri bulunmaktadır.

this Referansı

Aslında makinanın ve arakodun çalışma prensibine bakıldığında static olmayan metot kavramı yapay bir kavramdır. Derleyici tüm metotları sanki static metotlarmış gibi derler. Biz de örneğin static olmayan bir metodu static metot haline getirebiliriz. Bunun için tek yapacağımız şey static metoda bir parametre daha eklemektir. Bu parametre static olmayan metodun çağrılmasında kullanılan referansı argüman olarak alır, erişimi o referansı kullanarak yapar. Örneğin:

```
using System;

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

            s.Set(10, 20);
            s.Disp();
        }
    }

    class Sample
    {
        private int m_a;
        private int m_b;

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

        public void Disp()
        {
            Console.WriteLine("{0}, {1}", m_a, m_b);
        }
    }
}
```

Static olmayan metotların eşdeğer static karşılıkları yazılabilir. Tek yapılacak şey yukarıda da belirtildiği gibi bunlara ekstra kendi sınıfı türünden bir parametre geçirmektir. Bunlar çağrılırken de sınıf referansı bunlara argüman olarak verilir. Örneğin yukarıdaki kodun static eşdeğeri şöyle oluşturulabilir:

```
using System;

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

            Sample.Set(s, 10, 20);
            Sample.Disp(s);
        }
    }

    class Sample
    {
        private int m_a;
        private int m_b;
```

```
        public static void Set(Sample s, int a, int b)
        {
            s.m_a = a;
            s.m_b = b;
        }

        public static void Disp(Sample s)
        {
            Console.WriteLine("{0}, {1}", s.m_a, s.m_b);
        }
    }
}
```

Aslında biz static olmayan metotları yazıp çağırdığımızda derleyici bunları static metot gibi derlemektedir. Örneğin sınıfın Foo isimli static olmayan bir metodu o sınıf türünden r isimli referansla çağrılıyor olsun:

```
r.Foo();
```

Aslında derleyici metodu static metot gibi derler ve bizim onu çağırmakta kullandığımız referansı metoda argüman olarak geçirir:

```
Sample.Foo(r);
```

İşte static olmayan metotları static yaparken derleyicinin gizlice geçirdiği referans parametresine biz metodun içerisinde this anahtar sözcüğü ile erişebiliriz. this anahtar sözcüğü static olmayan metodun çağrılmasında kullanılan referansı temsil etmektedir. Örneğin:

```
r.Foo();
```

çağrısındaki Foo içerisinde this anahtar sözcüğünü kullanırsak bu r referansıdır.

static olmayan bir metot içerisinde sınıfın m_a isimli bir veri elemanını m_a biçiminde kullanmakla this.m_a biçiminde kullanmak arasında hiçbir farklılık yoktur. İşin aslı biz elemanı m_a biçiminde kullansak bile aslında derleyici zaten ona this.m_a gibi bir kodla erişmektedir.

this referansının türü nedir? İşte biz this anahtar sözcüğünü hangi sınıf içerisinde kullanıyorsak this o türdendir. Sınıfın static metotlarında ve property elemanlarında this anahtar sözcüğü kullanılamaz. Çünkü zaten onlar bir referansla çağrılmamaktadır. Dolayısıyla derleyici onlara böyle bir argüman geçirmeemektedir.

Peki this anahtar sözcüğüne neden gereksinim duyulmaktadır?

Sınıfın bir veri elemanıyla aynı isimli bir yerel değişken ya da parametre değişkeni olduğu durumda sınıfın veri elemanına erişmek için this kullanılabilir. Örneğin:

```
    class Sample
    {
        private int a;

        public Sample(int a)
        {
            this.a = a;           // this.a sınıfın veri elemanı olan a
            //...
        }
        //...
    }
```

Bazı programcılar bir çakışma olmasa bile yine de this anahtar sözcüğünü kullanmaktadır. Böylece kodu inceleyen kişiler ilgili değişkeninin sınıfın bir veri elemanı olduğunu anlayabilmektedir. Bazı programcılar da bu vurgulamayı this ile değil sınıfın veri elemanlarını belli bir önekle isimlendirerek yaparlar. Biz de kursumuzda sınıfın veri elemanlarını m_xxx öneki ile isimlendiriyoruz. Böylece kodu inceleyen kişi onun bir

veri elemanı olduğunu hemen anlayabiliyor.

Sınıfın static olmayan bir metodu sınıfın başka bir static olmayan metodunu doğrudan Foo() biçiminde çağırmasıyla this.Foo() biçiminde çağırması arasında bir farklılık yoktur. Fakat bazı programcılar bu durumda da özellikle this anahtar sözcüğünü kullanmaktadır. Örneğin:

```
class Sample
{
    //...
    public void Foo()
    {
        //...
        this.Bar();     // Bar() çağrısı ile eşdeğer
        //...
    }

    public void Bar()
    {
        //...
    }
    //...
}
```

Bunun sebebi çağrılan metodun static olmayan bir metot olduğunu vurgulamaktır. this.Bar() birr çağrıda kodu inceleyen kişi Bar metodunun static olmayan bir metot olduğunu hemen anlar.

this anahtar sözcüğü sınıfın static olmayan property elemanlarının get ve set bölümlerinde de benzer biçimde kullanılmaktadır.

Delegeler (Delegates)

Aslında metotlar ardışıl makine komutlarından oluşmaktadır. (Tabii aslında .NET'te metotlar ara kodlardan oluşur. Ancak onlar yine gerçek makine komutlarına dönüştürülmektedir.) Bir metot onun bellekteki adresi bilinerek çağrılabilir. C#'ta bir metodun yalnızca ismi (yani (...) operatörü olmadan) o metodun bellekteki başlangıç adresi anlamına gelmektedir. Örneğin:

```
Sample.Foo();
```

ifadesi "Sample sınıfının static Foo metodunu çağır" anlamına gelirken,

```
Sample.Foo
```

ifadesi "Sample sınıfının static Foo metodunun başlangıç adresi anlamına" gelir. C#'taki (...) operatörü "operandı olan adreste bulunan fonksiyonun çağrılması işlemini yapar.

C#'ta metotları tutan istendiğinde tuttuğu metotları bizim çağıran özel sınıflara delege denilmektedir. Java'da delege yoktur. C++'ta zaten fonksiyon göstericileri delege yerine geçmektedir.

Delege bildirimlerinin genel biçimi şöyledir:

```
delegate <geri dönüş değerinin türü> <delege ismi> ([parametre bildirimi]);
```

Örneğin:

```
delegate int Proc(int a, int b);
delegate void Exec();
```

Delegeler aslında birer sınıftır. Dolayısıyla her ne kadar bildirimleri metotları çağrıştırsa da delege bildirildiğinde aslında bir sınıf bildirilmiş olmaktadır. Bu nedenle delege bildirimleri de normal olarak isim

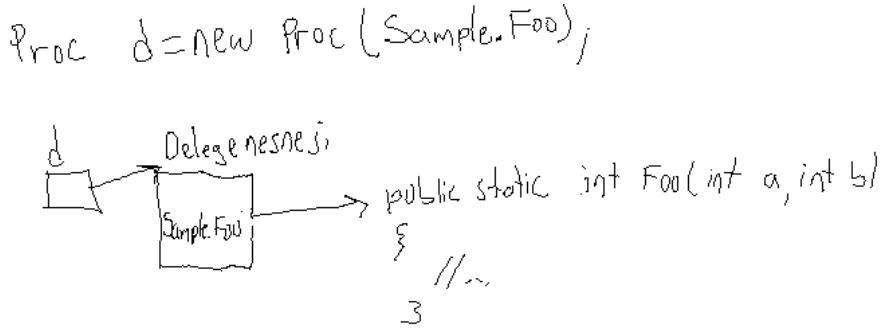
alanlarının içerisinde bulunmalıdır. Delegeler kategori olarak referans türlerine ilişkindir. Yani bir delege türünden değişken bildirdiğimizde o bir adres tutmaktadır.

```
Proc d;          // d bir referans;
Exec e;          // e bir referans
```

Bir metodun tutulması için onun başlangıç adresi yeterlidir. Çünkü metotların kodları bellekte ardışıl bir biçimde bulunur. Başlangıç adresi bilinen bir metot çağrılabilir.

Bir delege her metodu tutamaz. Geri dönüş değeri ve parametre türleri bildirimdeki gibi olan metotları tutabilir. Örneğin yukarıdaki Proc delegesi ismi ne olursa olsun geri dönüş değeri int, parametresi int, int olan metotları tutabilir. Exec delegesi geri dönüş değeri void parametresi olmayan metotları tutabilir.

Delege nesneleri yine new operatörü ile yaratılır. Derleyici delege bildirimleri için oluşturdukları sınıfa tek bir başlangıç metodu eklemektedir. O da delegenin tutacağı metodun adresini alır. Bunun dışında delege sınıfları default başlangıç metoduna sahip değildir. Örneğin:



Görüldüğü gibi delege nesnesi içerisinde delege nesnesinin tuttuğu metodun başlangıç adresi bulunmaktadır.

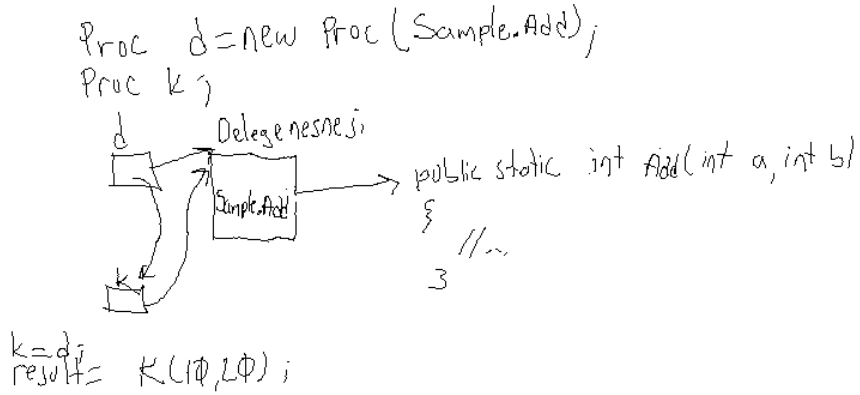
d bir delege referansı olmak üzere biz bu referansı metot çağırma operatörüyle d(...) biçiminde kullanırsak bu ifade "delegenin gösterdiği yerdeki delege nesnesinin içerisinde tutulan metodu çağır" anlamına gelir. Eğer metot parametreliyse argüman da girmek gerekir. Benzer biçimde d(...) ifadesinden elde edilen değer çağrılan metodun geri dönüş değeridir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Proc d;
            int result;

            d = new Proc(Sample.Add);
            result = d(10, 20);
            Console.WriteLine(result);

            d = new Proc(Sample.Multiply);
            result = d(10, 20);
            Console.WriteLine(result);
        }
    }

    class Sample
    {
        public static void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
```

```
        public static int Add(int a, int b)
        {
            return a + b;
        }

        public static int Multiply(int a, int b)
        {
            return a * b;
        }
        //...
    }

    delegate int Proc(int a, int b);
}
```

Aynı türden iki delege referansı birbirlerine atanabilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Proc d = new Proc(Sample.Add);
            Proc k;
            int result;

            k = d;

            result = k(10, 20);     // Sample.Add çağrılır
            Console.WriteLine(result);
        }
    }

    class Sample
    {
        public static int Add(int x, int y)
        {
            return x + y;
        }
        //...
    }

    delegate int Proc(int a, int b);
}
```



Örneğin:

```
using System;

namespace CSD
{
```

```
    class App
    {
        public static void Main()
        {
            Proc x = new Proc(Sample.Foo);
            Proc y = new Proc(Sample.Bar);
            Proc z = new Proc(Sample.Tar);

            DoSomething(x);
            DoSomething(y);
            DoSomething(z);
        }

        public static void DoSomething(Proc d)
        {
            d();
        }
    }

    class Sample
    {
        public static void Foo()
        {
            Console.WriteLine("Foo");
        }

        public static void Bar()
        {
            Console.WriteLine("Bar");
        }

        public static void Tar()
        {
            Console.WriteLine("Tar");
        }
        //...
    }

    delegate void Proc();
}
```

Aşağıdaki örnekte metot int türden bir diziyi ve bir delegeyi parametre olarak almıştır. Dizinin her elemanı için ilgili delege metodunu çağırmıştır:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int[] a = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

            DoForEach(a, new Proc(Sample.Disp));
        }

        public static void DoForEach(int[] a, Proc d)
        {
            foreach (int x in a)
                d(x);
        }
    }

    class Sample
    {
        public static void Disp(int a)
        {
            Console.WriteLine(a);
```

```
        }
        //...
    }

    delegate void Proc(int a);
}
```

Delege türünden diziler de bildirilebilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Proc[] procs = new Proc[] { new Proc(Sample.Foo), new Proc(Sample.Bar), new
Proc(Sample.Tar) };

            foreach (Proc proc in procs)
                proc();

            // Aynı işlem şöyle de yapılabilirdi
            for (int i = 0; i < procs.Length; ++i)
                procs[i]();
        }
    }

    class Sample
    {
        public static void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }

        public static void Bar()
        {
            Console.WriteLine("Sample.Bar");
        }

        public static void Tar()
        {
            Console.WriteLine("Sample.Tar");
        }
    }

    delegate void Proc();
}
```

Delegeler static olmayan metotları da tutabilirler. Tabii static olmayan metotlar referanslarla çağrıldığına göre bizim de delegeye onun çağrılacağı referansı da vermemiz gerekir. İşte r bir referans Foo da ilgili sınıfın static olmayan bir metodu olmak üzere r.Foo ifadesi ile static olmayan metodun adresi delegeye referansla birlikte verilir. Örneğin:

```
Sample s = new Sample();
Proc d = new Proc(s.Foo);
```

Şimdi d(...) biçiminde delege çağırması yapıldığında aslında Foo metodu s referansı ile çağrılacaktır. Örneğin:

```
using System;

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

```
        {
            Sample s = new Sample(10);

            Proc d = new Proc(s.Disp);

            d();        // 10
        }
    }

    class Sample
    {
        private int m_a;

        public Sample(int a)
        {
            m_a = a;
        }

        public void Disp()
        {
            Console.WriteLine(m_a);
        }
        //...
    }

    delegate void Proc();
}
```

Örneğin:

```
using System;

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

            for (int i = 0; i < 10; ++i)
                Console.Write("{0} ", d(10));
            Console.WriteLine();
        }
    }

    delegate int Proc(int val);
}
```

Bir delege nesnesi static olan ya da olmayan bir metodun ismi verilerek yaratıldığında eğer o metot overload edilmişse delegenin parametrik yapısı ve geri dönüş değerine uygun olan overload edilmiş metot delegeye yerleştirilir. Yukarıdaki örnekte Random sınıfının üç farklı Next metodu vardır. Ancak Proc delegesinin geri dönüş değeri ve parametresi int türden olduğu için geri dönüş değeri ve parametresi int türden olan Next metody delege nesnesine yerleştirilecektir.

Delegelere Neden Gereksinim Duyulmaktadır?

Bazen birtakım sınıflar bazı olayları kendileri izlerler. O olaylar gerçekleştiğinde bizim bir metodumuzu çağırarak akışı bize verirler. İşte bu tür işlemlerde delegeler kullanılmaktadır. Yani "bir olay gerçekleştiğinde benim şu metodumu çağır" türü işlemler C#'ta delegeler yardımıyla yapılır. Aslında bu tür olaylar sanal metotlar yoluyla da yapılabilir. Şöyle ki: Sınıf bizden bir taban sınıf referansı ister. Olay gerçekleştiğinde o taban sınıfın sanal metodunu çağırır. Biz de ona o sınıftan türetme yapıp türemiş sınıf referansını veririz. Türemiş sınıfta da o metodu override ederiz. Böylece yazdığımız metot çağrılır. Ancak bu yöntem pek esnek değildir göreli olarak yavaştır. Java'da delegeler olmadığı için yalnızca bu yöntem kullanılmaktadır.

Delegelerin nasıl kullanıldığına ilişkin tipik bir örnek System.Threading isim alanındaki Timer sınıfıyla verilebilir. Bu sınıfın başlangıç metodu bizden bir delege yoluyla metot ister. Belli periyotlarda o metodu sürekli çağırır. Biz de periyodik işlemler yapabiliriz. Sınıfın başlangıç metodu şöyledir:

```
public Timer(TimerCallback callback, Object state, int dueTime, int period)
```

Metodun birinci parametresi TimerCallback isimli bir delege türündendir. Bu delege şöyle bildirilmiştir:

```
public delegate void TimerCallback(Object state)
```

İkinci paremetre delege metodu her çağrıldığında metoda geçirilecek ekstra değerdir. Üçüncü parametre bu işleme ne kadar milisaniye sonra başlanacağını belirtir. Son parametre de milisaniye cinsinden periyottur. Örneğin:

```
using System;
using System.Threading;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Timer timer = new Timer(new TimerCallback(Sample.Print), ".", 0, 1000);

            Console.ReadLine();
        }
    }

    class Sample
    {
        public static void Print(object o)
        {
            string s = (string)o;
            Console.Write(s);
        }
    }
}
```

Örneğin:

```
using System;
using System.Threading;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Timer timer = new Timer(new TimerCallback(Sample.Print), null, 0, 1000);

            Console.CursorVisible = false;
            Console.ReadLine();
        }
    }

    class Sample
    {
        public static void Print(object o)
        {
            string str;
            DateTime dt = DateTime.Now;
```

```
            str = string.Format("{0}:{1}:{2}", dt.Hour, dt.Minute, dt.Second);
            Console.SetCursorPosition(70, 1);
            Console.Write(str);
        }
    }
}
```

Örneğin bir GUI programında düğmeler Button isimli bir sınıfla temsil edilmiştir. Biz bir Button nesnesi yarattığımızda bir düğme yaratmış oluruz. Tüm pencereler gibi düğmeler de üzerine tıklanıp tıklanmadığını kendileri tespit ederler. Button sınıfı bizden bir delege yoluyla metot alıp üzerine tıklandığında o metodu çağırabilmektedir. İşte .NET'in GUI programlama modelinde mesaj işlemleri hep arka planda delegeler kullanılarak yapılmaktadır.

Delegelerle İlgili İşlemler

Bir delege nesnesi işin başında bir metot verilerek yaratılır. Fakat daha sonra dele nesnesinin birden fazla metodu tutması da sağlanabilir. Aynı türden iki delege referansı + operatörüyle toplanabilmektedir. Bu durumda yeni bir delege nesnes, yaratılır. Yeni delege nesnesinin metot listesi iki delege nesnesinin meetot listesinin birleşiminde oluşur. Artık o delege referansıyla delege metotlarını çağırırsak bunlar sırasıyla çağrılacaktır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Proc d1 = new Proc(Sample.Foo);
            Proc d2 = new Proc(Sample.Bar);
            Proc d3;

            d3 = d1 + d2;
            d3();
        }
    }

    class Sample
    {
        public static void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }

        public static void Bar()
        {
            Console.WriteLine("Sample.Bar");
        }
    }

    delegate void Proc();
}
```

Eğer delege birden fazla metodu tutuyorsa ve bu metotların geri dönüş değerleri varsa biz bu delege yoluyla delege metotlarını metotlarını çağırdığımızda son metodun geri dönüş değerini elde ederiz. Örneğin delege nesnesinin içerisinde Foo ve Bar metotları olsun. Bunların da geri dönüş değerleri int türden olsun:

```
val = d();          // Tar'ın geri dönüş değeri elde edilir.
```

Örneğin:

```
using System;

namespace CSD
```

```
{
    class App
    {
        public static void Main()
        {
            Proc d1 = new Proc(Sample.Foo);
            Proc d2 = new Proc(Sample.Bar);
            Proc d3;
            int result;

            d3 = d1 + d2;
            result = d3();
            Console.WriteLine(result);
        }
    }

    class Sample
    {
        public static int Foo()
        {
            Console.WriteLine("Sample.Foo");

            return 100;
        }

        public static int Bar()
        {
            Console.WriteLine("Sample.Bar");

            return 200;
        }
    }

    delegate int Proc();

}
```

Eğer operadların biri null ise toplama işleminin sonucunda yeni bir delege nesnesi oluşturulmaz. Toplama işleminde null olmayan referans elde edilir. Örneğin sembolik olarak:

```
d1 ---> Foo
d2 --> null
d3 = d1 + d2;

d3 == d1
```

Eğer her iki referans da null ise toplamadan elde edilen sonuç da null referans olur.

Aynı türden iki delege referansı - operatörüyle çıkartma işlemine sokulabilir. Bu durumda yeni bir delege nesnesi yaratılır. Bu delege nesnesinin metot listesinde soldaki delegenin metot listesinden sağdaki delegenin metot listesinin çıkartılmasıyla kalan metot listesi olacaktır. Örneğin sembolik olarak:

```
d1 ---> Foo, Bar
d2 ---> Bar
d3 = d1 - d2;

d3---> Foo
```

Fakat sağdaki delegenin metot listesinde soldaki delegenin metot listesi yoksa yeni bir delege nesnesi yaratılmaz. Çıkartma işleminden soldaki delege referansının aynısı elde edilir. Sembolik olarak örneğin:

```
d1 ---> Foo, Bar
d2 ---> Tar
d3 = d1 - d2;
```

```
d3 == d1
```

Ayrıca liste içerisinde sıra önemlidir. Yani metotların aynı dizilimde bulunması gerekir. Sembolik olarak örneğin:

```
d1 ---> Foo, Bar, Tar
d2 ---> Foo, Tar
d3 = d1 - d2;

d3 == d1
```

Çünkü burada d1'de Foo ve Tar yan yana değildir.

Eğer çıkartma işleminde sağdaki delegenin metot listesi soldaki delegenin metot listesinin birden fazla yerinde varsa listede sonda olan çıkartılır. Sembolik olarak örneğin:

```
d1 ---> Foo, Bar, Foo
d2 ---> Foo
d3 = d1 - d2;

d3 --> Foo, Bar
```

Çıkartma işleminde soldaki operand null ise ya da her iki operand null ise sonuç null elde edilir. Sağdaki operand null ise çıkartma işleminin sonucu olarak soldaki referansın aynısı elde edilir. Eğer çıkartma sonucunda hiçbir metot kalmamışsa yine null referans elde edilir.

Toplama ve çıkartma dışında delegeler başka işlemlere sokulamazlar.

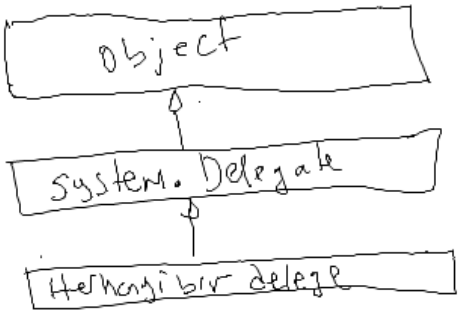
Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Proc d = null;

            d += new Proc(Sample.Foo);      // eşdeğeri d = d + new Proc(Sample.Foo)
            d += new Proc(Sample.Bar);      // eşdeğeri d = d + new Proc(Sample.Bar)

            d();
        }
    }

    class Sample
    {
        public static void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }

        public static void Bar()
        {
            Console.WriteLine("Sample.Bar");
        }
    }

    delegate void Proc();
```

```
}
```

Delegelerin Türetme Durumları

Delegelerden türetme yapılamaz. Ancak tüm delegelerin System isim alanı içerisindeki Delegate isimli bir sınıftan türetildiği varsayılmaktadır.



Sınıfların ve Yapıların Event Elemanları

Sınıfların ve yapıların delege türünden veri elemanlarını dışarıya kısıtlamak için event elemanlar dile sokulmuştur. Örneğin bir sınıfın D isimli public bir delege veri elemanı olsun ya da sınıfın m_d isimli bir private bir delege veri elemanı olduğunu ve onun D isimli bir property ile dışarıya açıldığını düşünelim. Biz bu D elemanı ile her şeyi yapabiliriz. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s = new Sample();
            s.P = new Proc(App.Foo);        // geçerli
            s.P();                          // geçerli
        }

        public static void Foo()
        {
            Console.WriteLine("Foo");
        }
    }

    class Sample
    {
        private Proc m_proc;

        public Proc P
        {
            get { return m_proc; }
            set { m_proc = value; }
        }
        //...
    }

    delegate void Proc();

}
```

Delege türlerinden veri elemanlarına güvenli bir biçimde erişebilmek için event elemanlar kullanılmaktadır. Event elemanları bir çeşit özel delege propery'si gibi düşünebilirisniz. Bir event elemanın bir delege property'sinden farkı veri elemandan farkı dışarıdan yalnızca ona += ve -= operatörleriyle erişilmesidir. Bu iki operatör dışında event elemanlar başka operatörlerle işleme sokulamazlar. Onlara atama dışarıdan atamalar

yapamayız, onların tuttulkları metotları dışarıdan çağıramayız. Örneğin E Sample sınıfının bir event eleman olsun:

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

s.E = new Proc(Sample.Foo);      // error!
s.E();                           // error!

s.E += new Proc(Sample.Foo);     // geçerli
s.E -= new Proc(Sample.Foo);     // geçerli
```

Eevent elemanların genel bildirim biçimi şöyledir:

```
public event <delege türü> <isim>
{
	add
	{
		//...
	}

	remove
	{
		//...
	}
}
```

Bir event elemanın add ve remove isimli iki bölümü vardır. (Bu iki bölüm de bulunmak zorundadır. Yani read-only ya da write-only event kavramı yoktur.) Genellikle sınıfta private bir delege veri elemanı tutulur. add bölümünde ona ekleme yapılır, remove bölümünde de çıkartma yapılır. Yani event eleman aslında delege property'si gibidir.

value anahtar sözcüğü hem add bölümünde hem de remove bölümünde kullanılabilir. value anahtar sözcüğü += ve -= operatörünün sağındaki delege referansını temsil eder. event eleman += operatörüyle kullanıldığında onun add bölümü, -= operatörüyle kullanıldığında remove bölümü çalıştırılır.

Anahtar Notlar: Visual Studio IDE'sinde event elemanlar intellisense'te şimşek çakma sembolüyle temsil edeilmektedir.

Bir event elemanın metotlarını dışarıdan biz çağıramayız. Bunu ancak o sınıf belli olay olduğunda kendisi çağırır. Zaten bunun için elemana "event" denilmiştir. Tabi biz test amacıyla sınıfa onu çağıracak bir metot yerleştirip event metotlarının dolaylı olarak çağrılmasını sağlayabiliriz. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            EventTest et = new EventTest();

            et.E += new Proc(Sample.Foo);
            et.Fire();
        }
    }

    class EventTest
    {
        private Proc m_e;

        public event Proc E
        {
            add
            {
```

```
            m_e = m_e + value;          // m_e += value;
        }

        remove
        {
            m_e = m_e - value;          // m_e -= value;
        }
    }

    public void Fire()
    {
        m_e();
    }
}

class Sample
{
    public static void Foo()
    {
        Console.WriteLine("Foo");
    }
}

delegate void Proc();
}
```

Pekiyi biz bir sınıfın bir elemanının event eleman olduğunu gördüğümüzde bundan ne anlamalıyız? (Örneğin Button sınıfının Click isimli bir event elemanı olsun):

1) Event eleman aslında bir delege property'si gibidir. Yani event eleman bir delege türündendir.

2) Biz bu event elemanı yalnızca += ve -= operatörleriyle kullanırız. Yani biz ona yalnızca delege ekleyip delege çıkartırız.

3) Eklediğimiz delege metotlarını biz çağıramayız. Delege metotlarını belli bir olay olduğunda o sınıfın kendisi çağıracaktır.

Uygulamada genellikle sınıfların delege türünden veri elemanları o sınıfların private bölümlerine yerleştirilip onlar dışarıya event eleman yoluyla açılmaktadır. Çünkü sınıfın event elemanının metotlarının dışarıdan çağrılması genellikle istenmez. Ayrıca dışarıdan sınıfın delege elemanına ekleme çıkartma yaparken başka birtakım işlemlerin de yapılması gerekebilir. Bu işlemler gizlice add ve remove bölümlerinde yapılabilmektedir.

Event Elemanların Kolay Yazımı (Erişimcisiz Event Elemanlar)

Event elemanın kolay yazılabilmesi için bir dile kısa yol eklenmiştir. Bir event eleman event anahtar sözcüğü kullanılarak sanki delege türünden veri elemanı gibi bildirilebilir. Bu durumda derleyici otomatik olarak sınıfın private bölümüne bir delege veri elemanı yerleştirir. Sonra bunun için bir event eleman yazar. event elemanın add bölümünde delegeye ekleme, remove bölümünde delegeden çıkartma yapar. Sınıf içerisinde bu event eleman kullanıldığında derleyici bildirmiş olduğu private delege veri elemanı kullanılmış gibi işlem yapmaktadır. Örneğin:

```
class Sample
{
       public event Proc E;        // event bildiriminin kolay yolu
       //
       public void Fire()
       {
              E();
       }
       //...
}
```

Bu işlemin eşdeğeri:

```
class Sample
{
	private Proc m_compilerGeneratedName;

	public event Proc E
	{
		add { m_compilerGeneratedName += value; }
		remove { m_compilerGeneratedName -= value;
	}
	//...
	public void Fire()
	{
		m_compilerGeneratedName();
	}
	//...
}
```

Sınıf içerisinde artık event eleman kullanıldığında bu eleman derleyicinin private bölüme yerleştirdiği delege veri elemanını temsil etmektedir. Yani sınıf bildirimi içerisinde biz evet elemanı sanki delege veri elemanıymış gibi kullanabiliriz. Tabii o dışarıdan yine yalnızca += ve -= operatörleriyle kullanılabilmektedir.

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            EventTest et = new EventTest();

            et.E += new Proc(Sample.Foo);
            et.Fire();
        }
    }

    class EventTest
    {
        public event Proc E;

        public void Fire()
        {
            E();
        }
    }

    class Sample
    {
        public static void Foo()
        {
            Console.WriteLine("Foo");
        }
    }

    delegate void Proc();
}
```

Metot İsimlerinden Delegelere Otomatik Dönüştürme

Normal olarak bir delege referansına bir delege nesnesi yaratılıp onun adresi atanır. Örneğin:

```
Proc d = new Proc(Sample.Foo);
```

Fakat ".NET Framework 2.0" ile birlikte yazımı kolaylaştırmak için metot isimlerinden (yani metot adreslerinden) delegelere otomatik dönüşüm tanımlanmıştır. Buna göre biz bir delege referansına doğrudan parametrik yapısı ve geri dönüş değeri o delegeyle uyumlu olan bir metodun ismini atayabiliriz. Örneğin:

```
Proc d = Sample.Foo;
```

Burada önemli nokta şudur: Bu işlemle d referanına metodun adresi atanmamaktadır. Derleyici böylesi bir atama işleminde yine bir delege nesnesi yaratıp, metodu onun içerisine yerleştirip o delege nesnesinin adresini referansa atamaktadır. Yani:

```
Proc d = Sample.Foo;
```

işlemi ile,

```
Proc d = new Proc(Sample.Foo);
```

işlemi tamamen eşdeğerdir.

+ ve - operatörünün bir operandı delege türünden referans ise diğeri doğrudan metot ismi olabilir. Örneğin:

```
d = d + Sample.Foo;
```

işlemi geçerlidir. Benzer biçimde aynı şey += ve -= operatörüyle de yapılabilir. Örneğin:

```
d += new Proc(Sample.Foo);
```

ile,

```
d += Sample.Foo;
```

aynı anlamdadır. Tabi event elemanlarda da aynı şey yapılabilir. Yani E TestEvent sınıfının bir event elemanı olmak üzere:

```
TestEvent te = new TestEvent();

te.E += new Proc(Sample.Foo);
```

ile,

```
TestEvent te = new TestEvent();

s.E += Sample.Foo;
```

aynı anlamdadır. Toplama ve çıkartma işleminde her iki operand da metot ismi olamaz. Örneğin:

```
d = Sample.Foo + Sample.Bar;           // error
```

Tabii aynı işlemler static olmayan metotlar için de benzer biçimde geçerlidir. Örneğin Foo r referansına ilişkin sınıfın static olmayan bir metodu olsun:

```
d = r.Foo;          // geçerli
```

İskelet Bir GUI Programının Oluşturulması

1) İskelet bir GUI programı için önce boş bir proje yaratılır. O projede aşağıdaki DLL'lere referans edilir:

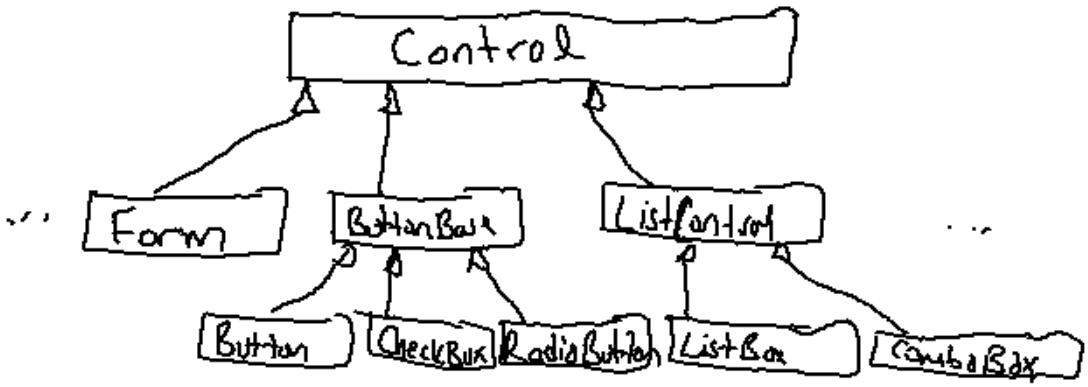
```
System.dll
System.Windows.Forms.dll
system.Drawing.dll
```

2) Daha sonra projeye bir bir dosya eklenir ve aşağıdaki program yazılır:

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

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Application.Run(new MainForm());
        }
    }

    class MainForm : Form
    {
        public MainForm()
        {
            //...
        }
        //...
    }
}
```

3) Proje seçeneklerine gelinir. Output Type: "Windows Application" yapılır.

Bir GUI programında ekranda gördüğümüz görsel öğelerin hepsi birer penceredir. Yani örneğin programın ana penceresi, düğmeler, edit alanları vs. hep birer penceredir. Tüm pencerelerin ortak birtakım özellikleri vardır. Bu ortak özellikler Control isimli sınıfta toplanmıştır. Control sınıfından sınıflar türetilmiştir. Örneğin:



Böylece biz bir görsel öğeyi pencereye yerleştirmek istersek, ilgili sınıf türünden bir nesne yaratırız. Sonra o nesnenin hangi pencerenin içerisinde olması gerektiğini belirleriz. Bir pencere başka bir pencerenin içerisindeyse ve onun dışına çıkamıyorsa böyle pencerelere alt pencere (child window) denilmektedir. Her alt pencerinin bir üset penceresi (parent window) vardır.

.NET Form kütüphanesinde programın ana penceresi Form sınıfıyla temsil edilmiştir. Fakat iskelet programda Form sınıfının içerisine elemanlar yerleştirileceği için Form sınıfının doğrudan kullanılması yerine iskelet ondan MainForm isimli bir sınıf türetilmiştir. Bir görsel öğeyi Form pencersine (yani ana pencereye) eklemek için yapacağımız şey o görsel öğenin ilişkin olduğu sınıf türünden bir nesne yaratıp onun Parent property'sine Form'un referansını atamaktır.

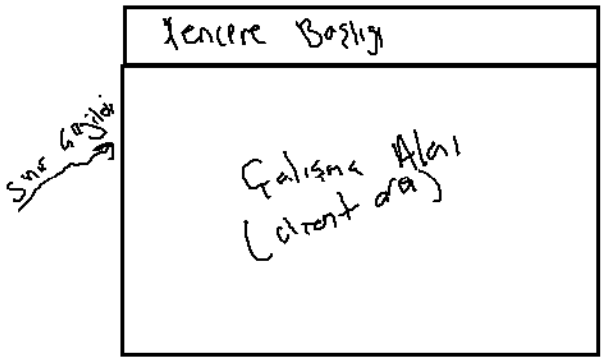
Form sınıfı ile onun içerisindeki görsel öğelere ilişkin sınıflar arasında içerme ilişkisi vardır. (Yani örneğin form açıldığında düğmeler, edit alanları ile vs. açılır. Kapandığında da bunların hepsi yok olur.) Bu nedenle bizim görsel öğelere ilişkin referansları Form sınıfının private bölümünde tutmamaız ve bunların yaratımlarını

Form sınıfının başlangıç metodu içerisinde yapmamız uygun olur. Örneğin:

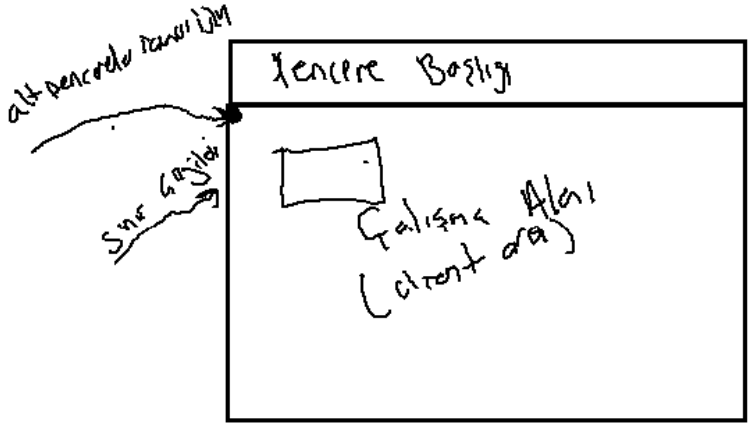
```
class MainForm : Form
{
    private Button m_buttonOk;
    private Button m_buttonCancel;
    //...

    public MainForm()
    {
        m_buttonOk = new Button();
        m_buttonCancel = new Button();
        //...
    }
    //...
}
```

Control sınıfının elemanları tüm pencere sınıflarında kullanılabilmektedir. Örneğin Control sınıfının string türünden Text isimli bir property elkemanı vardır. Bu eleman Form sınıfında pencere başlık yazısını, Button sınıfında düğmenin üzerindeki yazıyı, TextBox sınıfında edit alanı içerisindeki yazıyı belirtir. Yani her sınıfın bir Text property'si vardır. Ancak bu property o sınıflara özgü bir anlam taşımaktadır.

Windows'ta pencere başlığının aşağısındaki programcı tarafından kullanılabilen aktif çizim alanına çalışma alanı (client area) denilmektedir:



Control sınıfının Point türünden Location property'si pencerenin sol üst köşe konumunu belirlemek için kullanılır. Biz bu property'yi pencereyi konumlandırmak için kullanırız. Ana pencereler için (yani form için orijin noktası masaüstünün sol-üst köşesi, alt pencereler için ise orijin noktası onun üst penceresinin çalışma alanının sol-süst köşesidir:

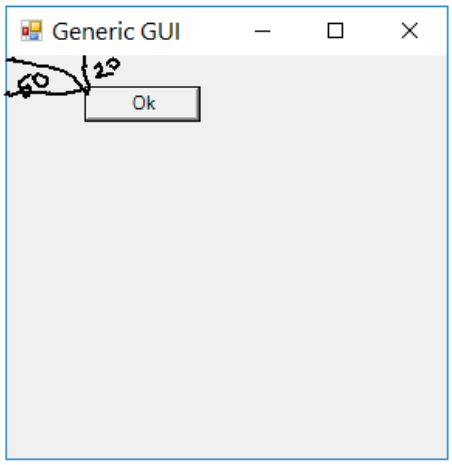


Kullanılan birim pixel'dir. Örneğin:

```
m_buttonOk = new Button();
m_buttonOk.Parent = this;
m_buttonOk.Text = "Ok";
```

```
m_buttonOk.Location = new Point(50, 20);
```

Burada üzerinde “Ok” yazısı yazan düğme çalışma alanının sol-üst köşesine göre 50 pixel sağda ve 20 pixel aşağıdadır:



Control sınıfındaki Size property’si Size isimli bir yapı türündendir. Pencereninin genişlik ve yüksekliğini ayarlamakta kullanılır. Kontrollerin (yani alt pencerelerin) yaratıldığında default bir genişlik-yükseklik değeri vardır. Fakat biz onları daha sonra değiştirebiliriz.

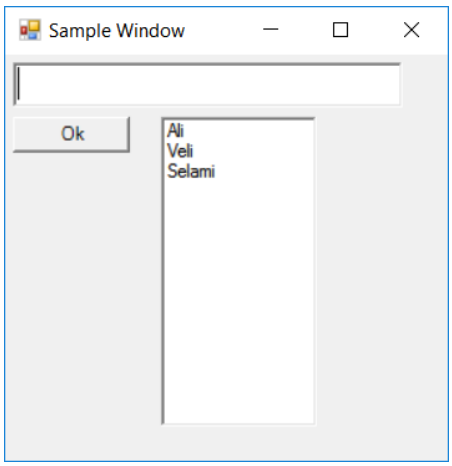
Control sınıfının Click isimli event elemanı EventHandler isimli bir delege türündendir. Bu delege aşağıdaki gibi bildirilmiştir:

```
delegate void EventHandler(object sender, EventArgs);
```

Bir kontrole fare ile tıklanıp elimizi kontrolden çektiğimizde Click isimli event elemanın (yani delegenin) tuttuğu metotlar o kontrol tarafından çağrılır. Bu durumda örneğin bir düğmeye tıkladığımızda bir kodun çalışmasını istiyorsak Button sınıfının Click event elemanına delege nesnesi girmemiz gerekir. Örneğin:

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

namespace CSD
{
    class App
    {
        public static void Main()
        {
            MainForm mf = new MainForm();
            Application.Run(mf);
        }
    }

    class MainForm : Form
    {
        private Button m_buttonOk;
        private TextBox m_textBoxName;
        private ListBox m_listBox;

        public MainForm()
        {
            Text = "Sample Window";

            m_textBoxName = new TextBox();
            m_textBoxName.Location = new Point(5, 5);
            m_textBoxName.Width = 250;
            m_textBoxName.Font = new Font("Times New Roman", 14);
            m_textBoxName.ForeColor = Color.Red;
```

```
            m_textBoxName.Parent = this;

            m_buttonOk = new Button();
            m_buttonOk.Text = "Ok";
            m_buttonOk.Location = new Point(5, 40);
            m_buttonOk.Click += new EventHandler(buttonOkClickHandler);
            m_buttonOk.Parent = this;

            m_listBox = new ListBox();
            m_listBox.Parent = this;
            m_listBox.Location = new Point(100, 40);
            m_listBox.Width = 100;
            m_listBox.Height = 200;
            m_listBox.DoubleClick += listBoxDoubleClickHandler;

            m_listBox.Items.Add("Ali");
            m_listBox.Items.Add("Veli");
            m_listBox.Items.Add("Selami");
        }

        private void listBoxDoubleClickHandler(object sender, EventArgs e)
        {
            MessageBox.Show(m_listBox.SelectedItem.ToString());
        }

        private void buttonOkClickHandler(object o, EventArgs e)
        {
            MessageBox.Show(m_textBoxName.Text);
        }
    }
}
```



Exception İşlemleri (Exception Handling)

İnglizce'de "exception" sözcüğü "istisna" anlamına gelmektedir. Exception bir terim olarak yazılımda "aniden ortaya çıkan problemli durumları" anlatmak için kullanılmaktadır. C#'ta ve diğer nesne yönelimli dillerin hemen hepsinde bir exception mekanizması vardır. Daha önceki konularda bir exception oluştuğunda programın çöktüğünü gördük. Fakat aslında bir exception oluştuğunda exception ele alınarak (handle edilerek) programın çökmesi engellenebilir.

Exception mekanizması sayesinde bir kod parçasındaki problemli durumlar tek bir yerden ele alınarak yönetilebilmektedir. Bu da hem programcının işini kolaylaştırmakta hem de kodun daha sade gözükmesine yol açmaktadır.

C#'ta exception işlemleri için dört anahtar sözcükten faydalanılır: try, catch throw ve finally.

try anahtar sözcüğünü bir blok izlemek zorundadır. Buna "try bloğu" denir. try bloğu tek başına bulundurulamaz. try bloğunu da bir ya da birden fazla catch bölümü ya da finally bölümü izlemek zorundadır. Yani try bloğundan sonra bir ya da birden fazla catch bölümü izeleyebilir, bir finally bölümü izleyebilir ya da

önce bir ya da birden falz catch bölümü sonra finally bölümü izleyebilir. catch bölümünün genel biçimi şöyledir:

```
catch (<tür>[isim])
{
      //...
}
```

catch anahtar sözcüğünü parantezler izler. Bu parantezlerin içerisinde catch parametre bildirimi yapılır. catch parametresi bir tane olmak zorundadır. Parametrenin yalnızca türü belirtilebilir ya da hem türü hem de ismi belirtilebilir. catch parametresi herhangi bir türden olamaz. System isim alanı içerisindeki Exception isimli sınıf türünden ya da o sınıftan türetilmiş bir sınıf türünden olmak zorundadır. try ile catch arasında ve catch bloklarının arasında hiçbir deyim bulunamaz. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                //...
            }
            catch (MyException me)
            {
                //...
            }
            catch (YourException ye)
            {
                //...
            }
        }
    }

    class MyException : Exception
    {
        //...
    }

    class YourException : Exception
    {
        //...
    }
}
```

Aynı parametre türüne ilişkin birden fazla catch bloğu da bulunamaz.

Programın akışı try bloğuna girdiğinde artık bir exception kontrolü uygulanır. try bloğu içerisindeyken bir exception oluştuğunda akış try bloğunun uygun uygun catch bloğuna aktarılır. Kontroller ve hata ele alımı catch bloklarında yapılmaktadır.

Exception'ı oluşturan asıl deyim throw deyimidir. throw deyiminin genel biçimi şöyledir:

```
throw [ifade];
```

throw anahtar sözcüğünün yanındaki ifade System isim alanı içerisindeki Exception sınıfı türünden ya da bu sınıftan türetilen bir sınıf türünden olmak zorundadır. Yani throw işlemi bir Exception sınıfı türünden ya da bu sınıftan türetilmiş bir sınıf türünden sınıf referansıyla yapılır. Programın akışı throw anahtar sözcüğünü gördüğünde akış bir goto işlemi gibi son girilen try bloğunun uygun parametreli catch bloğuna aktarılır. Artık akış bir daha geriye dönmez. O catch bloğu çalıştırılır. Sonra diğer catch blokları atlanır. Program catch bloklarının sonundan çalışmasına devam eder. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Console.WriteLine("main başladı");

            try
            {
                Foo(-2);
            }
            catch (MyException me)
            {
                Console.WriteLine("MyException yakalandı");
            }
            catch (YourException ye)
            {
                Console.WriteLine("YourException yakalandı");
            }

            Console.WriteLine("main bitti");
        }

        public static void Foo(int a)
        {
            Console.WriteLine("Foo başladı");

            if (a < 0)
            {
                MyException me = new MyException();
                throw me;
            }

            Console.WriteLine("Foo bitti");
        }
    }

    class MyException : Exception
    {
        //...
    }


    class YourException : Exception
    {
        //...
    }
}
```

Programın çıktısı şöyle olacaktır:

```
main başladı
Foo başladı
MyException yakalandı
main bitti
```

Eğer program try bloğuna girdikten sonra hiç exception oluşmazsa akış try bloğundan çıkar. Tüm catch blokları atlanır ve catch bloklarının sonundan çalışma devam eder. Yani catch blokları "exception oluşursa" işlem görmektedir.

Eğer akış try bloğuna girdikten sonra bir throw işlemi oluşur fakat exception hiçbir catch tarafından yakalanmazsa ya da o anda akış bakımından bir try bloğu içerisinde bulunulmuyorsa program çöker (ilgili thread sonlandırılır). Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Console.WriteLine("main başladı");

            try
            {
                Foo(-2);
            }
            catch (YourException ye)
            {
                Console.WriteLine("YourException yakalandı");
            }

            Console.WriteLine("main bitti");
        }

        public static void Foo(int a)
        {
            Console.WriteLine("Foo başladı");

            if (a < 0)
            {
                MyException me = new MyException();
                throw me;       // Dikkat bunu yakalayan bir catch yok!
            }

            Console.WriteLine("Foo bitti");
        }
    }

    class MyException : Exception
    {
        //...
    }

    class YourException : Exception
    {
        //...
    }
}
```

.NET'in sınıfı kütüphanesinde çeşitli metotlar problemli durumlarda çeşitli exception sınıflarıyla throw işlemi yapmaktadır. Biz de bu metotları çağırırken uygun catch bloklarını oluşturmalıyız. Yoksa exception oluştuğunda programımız çöker. Hangi metotların hangi sorunlar yüzünden hangi sınıflarla throw ettiği MSDN kütüphanesinde dokümante edilmiştir. Örneğin string sınıfının SubString metodu limit dışına çıkıldığında ArgumentOutOfRangeException isimli bir sınıfla throw eder:

```
using System;

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

            try
            {
                k = s.Substring(3, 7);
                Console.WriteLine(k);
```

```
            }
            catch (ArgumentOutOfRangeException e)
            {
                Console.WriteLine("Exception oluştu!");
            }

            Console.WriteLine("main bitti");
        }
    }
}
```

Örneğin:

```
using System;

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

            for (;;)
            {
                try
                {
                    Console.Write("Bir sayı giriniz:");
                    val = int.Parse(Console.ReadLine());
                    Console.WriteLine(val);
                    break;
                }
                catch (ArgumentNullException e)
                {
                    Console.WriteLine("Argüman null değerinde!");
                }
                catch (FormatException e)
                {
                    Console.WriteLine("Sayının formatı bozuk");
                }
                catch (OverflowException e)
                {
                    Console.WriteLine("Sayı çok büyük ya da çok küçük");
                }
            }
        }
    }
}
```

Örneğin Syste.IO isim ı içendekii Directory sınıfının GetFiles metodu eğer parametresi ile verilen dizin yoksa DirectoryNotFoundException sınıf ile throw etmektedir:

```
using System;
using System.IO;

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

            Console.Write("Lütfen bir yol ifadesi giriniz:");
            path = Console.ReadLine();
            try
            {
                files = Directory.GetFiles(path);
```

```
                foreach (string file in files)
                    Console.WriteLine(Path.GetFileName(file));
            }
            catch (DirectoryNotFoundException e)
            {
                Console.WriteLine("İlgili dizin bulunamadı!..");

            }
        }
    }
}
```

Türemiş sınıf türünden bir throw işlemi taban sınıf türünden bir catch ile yakalanabilir (türemişten tabana otomatik dönüştürme var olduğundan). Bu durumda örneğin biz tek bir Exception parametreli catch ile tüm exception'ları yakayalabiliriz. Tabi normal olarak hangi exception'ın fırlatıldığını anlayamayız. Örneğin:

```
using System;

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

            for (;;)
            {
                try
                {
                    Console.Write("Bir sayı giriniz:");
                    val = int.Parse(Console.ReadLine());
                    Console.WriteLine(val);
                    break;
                }
                catch (Exception e)
                {
                    Console.WriteLine("Hatalı giriş!");
                }
            }
        }
    }
}
```

Anahtar Notlar: r bir referan belirtmek üzere r referansın dinamik türüne ilişkin sınıf ya da yapının ismi r.GetType().Name ifadesi ile elde edilebilir.

Eğer taban sınıfla türemiş sınıf catch blokları birarada bulundurulacaksa taban sınıf catch bloğunun türemiş sınıf catch bloğundan daha aşağıda bulundurulması zorunludur. Çünkü catch blokları yukarıdan aşağıya doğru ele alınır. (Eğer taban sınıf catch bloğu daha yukarıda olursa zaten bu zaten tüm exception'ları yakalayacaktır). Böyle bir durumda eğer türemiş sınıf türüyle throw yapılmışsa bunu türemiş sınıfa ilişkin catch bloğu yakalar. Taban sınıf türüyle ya da diğer türlerden biriyle throw yapılmışsa bunu taban sınıfa ilişkin catch bloğu yakalar. Örneğin:

```
using System;

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

            for (;;)
            {
                try
```

```
                {
                    Console.Write("Bir sayı giriniz:");
                    val = int.Parse(Console.ReadLine());
                    Console.WriteLine(val);
                    break;
                }
                catch (FormatException e)
                {
                    Console.WriteLine("Sayının formatı bozuk!");
                }
                catch (Exception e)
                {
                    Console.WriteLine("Hatalı giriş: {0}", e.GetType().Name);
                }
            }
        }
    }
}
```

Akış bakımından iç içe try blokları söz konusu olabilir. Yani akış bir try bloğuna girdikten sonra başka bir try bloğuna da girebilir. Bu durumda iç bir try bloğunda throw oluşursa içten dışa doğru sırasıyla try bloklarının catch blokları taranır. Hangi catch bloğu uygunsa exception'ı o yakalar. Eğer sonuna kadar hiçbir catch bloğu bulunamazsa program çöker. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Console.WriteLine("Main başladı...");
            try
            {
                Sample.Foo(-10);
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception parametreli dış catch bloğu");
            }

            Console.WriteLine("Main bitti...");
        }
    }

    class Sample
    {
        public static void Foo(int a)
        {
            Console.WriteLine("Foo başladı...");
            try
            {
                Bar(a);
            }
            catch (MyException e)
            {
                Console.WriteLine("MyException parametreli dış catch bloğu");
            }
            Console.WriteLine("Foo bitti...");
        }

        public static void Bar(int a)
        {
            Console.WriteLine("Bar başladı...");

            if (a < 0)
                throw new YourException();
```

```
            Console.WriteLine("Bar bitti...");

        }
    }

    class MyException : Exception
    {
        //...
    }

    class YourException : Exception
    {
        //..
    }
}
```

Burada Sample sınıfının Bar metodu içerisinde oluşan YourException iç try bloğunun (son girilen try bloğunun) catch blokları tarafından yakalanamamaıştır. Bu durumda daha yukarıdaki try bloğunun catch bloklarına bakılacak ve exception dış try bloğunun catch bloğu tarafından yakalanacaktır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                Foo();
            }
            catch (FormatException e)
            {
                Console.WriteLine("Sayının formatı bozuk");
            }
        }

        public static void Foo()
        {
            try
            {
                int val;

                Console.Write("Bir Sayı giriniz:");
                val = int.Parse(Console.ReadLine());
                Console.WriteLine(val * val);
            }
            catch (OverflowException e)
            {
                Console.WriteLine("Sayı çok büyük ya da çok küçük");
            }
        }
    }
}
```

Burada Foo içerisinde FormatException oluşursa bunu dış try bloğunun catch bloğu yakalar. Fakat OverflowException oluşursa bunu iç try bloğunun catch bloğu yakalar. Tabii eğer exception iç try bloğunun catch blokları tarafından yakalanmışsa artık o exception ele alınmış demektir. Bunun dış try bloğuna bir etkisi olmaz.

finally Bloğu

finally bloğu parametresiz bir bloktur. Eğer yerleştirilecekse catch bloklarının en sonuna yerleştirilir. try bloğundan sonra catch bloğu olmadan finally bloğu olabilir. Yani üç durum söz konusudur:

- try - catch
- try - catch - finally
- try - finally

finally bloğu exception oluşsa da oluşmasa da çalıştırılır. Exception oluşmazsa try bloğundan sonra catch blokları atlanır ve finally bloğu çalıştırılır. Exception oluşursa önce uygun catch bloğu çalıştırılır. Sonra finally bloğu yine çalıştırılır. Örneğin:

```
using System;

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

            try
            {
                Console.Write("Bir sayı giriniz:");
                val = int.Parse(Console.ReadLine());
                Console.WriteLine(val * val);
            }
            catch (Exception e)
            {
                Console.WriteLine("geçersiz giriş!");
            }
            finally
            {
                Console.WriteLine("finally bloğu");
            }
        }
    }
}
```

Bir exception oluştuğunda bu exception catch bloğu tarafından yakalanmamış olsun. Bu durumda try bloğunun finally bloğu çalıştırılacaktır. Ancak exception yakalanmadığı için bu durum yine programın çökmesiyle sonuçlanacaktır.

Pekiyi finally bloğu yerine biz kodu catch bloklarının aşağısına yazsaydık değişen ne olurdu?

```
try
{
      //...
}
catch (Exception e)
{
      //...
}

//finally
{
      // finally'yi kaldırsak aynı şey olur mu?
}
```

İşte finally her zaman çalıştırılmaktadır. Yani try bloğu içerisinde break yapılsa, continue yapılsa, return yapılsa, goto yapılsa yine finally çalıştırılacaktır. Ayrıca iç try bloğunda exception oluştuğunda akış dış try bloğunun catch bloğu tarafından yakalanmadan önce onlar için finally blokları yine çalıştırılmaktadır. Örneğin:

```
using System;

namespace CSD
{
    class App
```

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

            for (;;)
            {
                try
                {
                    Console.Write("Bir sayı giriniz:");
                    val = int.Parse(Console.ReadLine());
                    if (val == 0)
                        break;
                    Console.WriteLine(val * val);

                }
                catch (Exception e)
                {
                    Console.WriteLine("geçersiz giriş!");
                }
                finally
                {
                    Console.WriteLine("finally bloğu");
                }
            }
        }
    }
}
```

Burada 0 girildiğinde for döngüsünden break ile çıkılmaktadır. Ancak yine finally bloğu çalıştırılacaktır. Döngü içerisinde continue ya da goto hatta return deyiminde de finally bloğunun çalıştırılmasına yol açar.

Örneğin:

```
using System;

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

            try
            {
                Foo();
            }
            catch (Exception e)
            {
                Console.WriteLine("exception oluştu");
            }
            finally
            {
                Console.WriteLine("dış finally bloğu");
            }
        }

        public static void Foo()
        {
            int val;

            try
            {
                Console.Write("Bir sayı giriniz:");
                val = int.Parse(Console.ReadLine());
                Console.WriteLine(val * val);
            }
            finally
            {
```

```
                Console.WriteLine("iç finally bloğu");
            }
        }
    }
}
```

Programcı exception oluştuğunda her zaman çalışmasını istediği birtakım boşaltım kodlarını finally bloğuna yerleştirebilir.

Parametresiz catch Bloğu

catch bloklarından biri de parametresiz catch bloğudur. Bu catch bloğu bulundurulacaksa tüm catch bloklarından sonra fakat finally bloğundan önce bulundurulur. Örneğin:

```
using System;

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

            try
            {
                Console.Write("Bir sayı giriniz:");
                val = int.Parse(Console.ReadLine());
                Console.WriteLine(val * val);
            }

            catch
            {
                Console.WriteLine("parametresiz catch bloğu");
            }
        }
    }
}
```

Parametresiz catch bloğu eğer yukarıdaki catch blokları yakalayamamışsa tüm exception'ları yakalar.

Pekiyi parametresiz catch bloğu ile Exception parametreli catch bloğu arasında ne fark vardır? İşte C++ gibi bazı dillerde herhangi bir türle throw edilebilmektedir. O dillerde yazılmış kodları C#'tan kullanırken bu exception'ları Exception parametreli catch bloğu ile yakalayamayız. Ancak onları parametresiz catch bloğu ile yakalayabiliriz. C# için iki durum arsında gerçekten bir fark yoktur. Ancak tabi parametresiz catch bloğunda fırlatılan exception nesnesini elde edemediğimize dikkat ediniz.

Exception Parametrelerinin Anlamı

Bir exception oluştuğunda onun nedne oluştuğu gibi, nerede oluştuğu gibi birtakım bilgileri elde etmek isteyebiliriz. Bunun için exception sınıfları kullanılmaktadır. Tipik olarak programcı önce bir exception nesnesini yaratır. Hata bilgileriyle bunun içini doldurur. Bununla throw eder. Exception'ı yakalayan kişi de bu nesnenin içerisinden bu bilgileri alıp isterse kullanabilir. Örneğin:

```
using System;

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

```
                Foo(-2);
            }
            catch (MyException e)
            {
                Console.WriteLine("Error ({0}):{1}", e.ErrCode, e.ErrText);
            }
        }

        public static void Foo(int a)
        {
            if (a < 0)
                throw new MyException(123, "Value cannot be negative!");

            Console.WriteLine("Ok");
        }
    }

    class MyException : Exception
    {
        private int m_errCode;
        private string m_errText;

        public MyException(int errCode, string errText)
        {
            m_errCode = errCode;
            m_errText = errText;
        }

        public int ErrCode
        {
            get { return m_errCode; }
            set { m_errCode = value; }
        }

        public string ErrText
        {
            get { return m_errText; }
            set { m_errText = value; }
        }

    }
}
```

System.Exception Sınıfı

Anımsanacağı gibi C#'ta bütün exception sınıfları System isim alanı içerisindeki Exception sınıfından türetilmek zorundadır. İşte bu Exception sınıfının bazı önemli elemanları vardır. Örneğin sınıfın virtual Message isimli property elemanı türemiş sınıflarda override edilmiştir ve ilgili exception'ın mesajını bize verir. Örneğin:

```
using System;

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

            try
            {
                Console.Write("Bir sayı giriniz:");
                val = int.Parse(Console.ReadLine());
                Console.WriteLine(val * val);
            }
            catch (Exception e)
            {
```

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

Burada örneğin FormatException oluşmuş olsun. Biz onu Exception parametreli catch bloğu ile yakalarız. Bu durumda e.Message yapıldığında aslında dinamik türe ilişkin FormatException sınıfının Message property'sinin get bölümü çalıştırılır.

Exception sınıfının StackTrace isimli string türünden property elemanı exception oluştuğunda hangi çağırma noktasından olunduğunu bize gösterir. Yorumalama aşağıdan yukarıya doğru yapılmalıdır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                Foo();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.StackTrace);
            }
        }

        public static void Foo()
        {
            Bar();
        }

        public static void Bar()
        {
            Tar();
        }

        public static void Tar()
        {
            int val;

            Console.Write("Bir sayı giriniz:");
            val = int.Parse(Console.ReadLine());
            Console.WriteLine(val * val);
        }
    }
}
```

Exception oluştuğunda ekran çıktısı şöyle olacaktır:

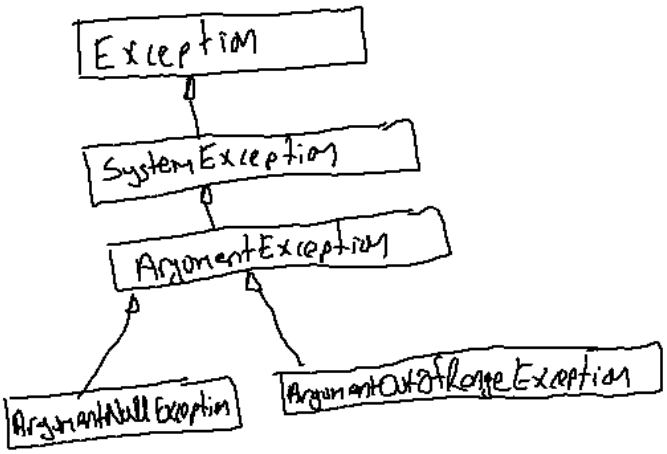
```
Bir sayı giriniz:ghjfhgfhgf
   konum: System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number, Numbe
rFormatInfo info, Boolean parseDecimal)
   konum: System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
   konum: System.Int32.Parse(String s)
   konum: CSD.App.Tar() e:\Dropbox\Kurslar\CSharp-Subat-2015\Src\Sample\Sample.cs içinde: satır 36
   konum: CSD.App.Bar() e:\Dropbox\Kurslar\CSharp-Subat-2015\Src\Sample\Sample.cs içinde: satır 28
   konum: CSD.App.Foo() e:\Dropbox\Kurslar\CSharp-Subat-2015\Src\Sample\Sample.cs içinde: satır 23
   konum: CSD.App.Main() e:\Dropbox\Kurslar\CSharp-Subat-2015\Src\Sample\Sample.cs içinde: satır 13
Press any key to continue . . .
```

Aslında exception ele alınmadığında program çökerken ekrana çıkan yazı StackTrace yazısıdır.

Önemli Exception sınıfları

Bu bölümde .NET'te çok karşılaşan bazı exception sınıflarından bahsedilecektir. Programcı da yeni exception sınıfı yazmak yerine zaten var olan bu exception sınıflarından biri kullanabilir.

Kütüphanede Exception sınıfından türetilmiş olan en önemli sınıf SystemException sınıfıdır. SystemException sınıfından türetilen ArgumentException genel olarak bir metodun parametresinin beğenilmemesi durumunda fırlatılır. Bundan da ArgumentNullException ve ArgumentOutOfRangeException sınıfları türetilmiştir.



Bir metodun parametresi null ise metot bunu kabul etmeyebilir. Bu durumda ArgumentNullException ile throw eder. Metodun parametresi belli sınırlar içerisinde olması gerekirken değilse metot ArgumentOutOfRangeException ile throw edebilir.

FormatException SystemException sınfından türetilen diğer bir exception sınıfıdır. Genel olarak bir metodun parametresi string ise ve onun belli bir formatı varsa, bu format yanlış girilmişse FormatException oluşur.

IndexOutRangeException yine SystemException sınıfından türetilmiştir. Bir dizinin pozitif ya da negatif bakımdan sınırları dışına erişim yapılmak istendiğinde CLR tarafından bu exception fırlatılır. Örneğin:

```
using System;

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

            try
            {
                Console.WriteLine(a[4]);
            }
            catch (IndexOutOfRangeException e)
            {
                Console.WriteLine("Dizi taşması oluştu!..");
            }
        }
    }
}
```

Aşağı doğru dönüştürme (downcast) yapılırken dönüştürülmek istenen referansın dinamik türü dönüştürülmek istenen türü içermiyorsa bu durumda CLR InvalidCastException oluşturur. Örneğin:

```
using System;

namespace CSD
{
```

```
    class App
    {
        public static void Main()
        {
            object o = 123;
            string s;

            try
            {
                s = (string)o;      // InnvalidCastException
                Console.WriteLine(s);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.GetType().Name);
            }
        }
    }
}
```

SystemException sınıfından türetilen diğer önemli bir exception sınıfı da NullReferenceException sınıfıdır. Bilindiği gibi bir referansın içerisinde null değeri varsa bu referans nokta operatörüyle kullanılamaz. İşte CLR bu durumda NullRefererenceException isimli exception'ı fırlatır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                string s = null;
                Console.WriteLine(s.Length);        // NullReferenceException
            }
            catch (NullReferenceException e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Arayüzler (Interfaces)

Bilindiği gibi C#'ta (ve Java'da böyle) çok türetme yoktur. Arayüzler çoklu türetmenin bazı avantajlarını kullanmamızı sağlayan türlerdir. Bir arayüz "tüm elemanları abstract olan ve veri elemanlarına sahip olmayan abstract sınıflara benzetilebilir. Arayüz bildiriminin genel biçimi şöyledir:

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

Arayüzler .NET'te geleneksel olarak I harfi ile başlatılarak isimlendirilmektedir. Bu durum onların hemen arayüz olarak fark edilmesini sağlar.

Bir arayüzün tüm elemanları default olarak public'tir. Bu nedenle arayüzlerde erişim belirleyicisi belirtilmez. (Belirtilirse error oluşur.) Arayüz metotları gövde içermez. Bildirimleri noktalı virgül ile kapatılmak zorundadır. Örneğin:

```
interface IX
{
    int Foo(int a, int b);
    void Bar();
}
```

Arayüz elemanları static olamaz. Arayüzler veri elemanlarına sahip olamazlar. Fakat property'lere sahip olabilirler.

Bilindiği gibi C#'ta bir sınıf yalnızca tek bir sınıftan türetilebilir. Sınıfın ya da yapının taban listesinde (yani ':' atomundan sonraki listede) bir arayüzün ismi geçirilirse bu duruma "ilgili sınıfın ya da yapının o arayüzü desteklemesi (implemente etmesi)" denilmektedir. Örneğin:

```
interface IX
{
    int Foo(int a, int b);
    void Bar();
}

class A
{
    //...
}

class B : A, IX
{
    //...
}
```

Burada B sınıfı A sınıfından türetilmiştir. Ancak IX arayüzünü de desteklemektedir. Örneğin:

```
interface IX
{
    int Foo(int a, int b);
    void Bar();
}

class Sample : IX
{
    //...
}
```

Burada Sample sınıfı object sınıfından türetilmiştir. Fakat IX arayüzünü e desteklemektedir.

Bir sınıf ya da yapı bir arayüzü destekliyorsa o sınıf ya da yapıda o arayüzün tüm metotları public düzeyde aynı geri dönüş değeriyle, aynı isimle ve aynı paramerik yapıyla gövdeli bir biçimde bildirilmek zarundadır. Örneğin:

```
interface IX
{
    int Foo(int a, int b);
    void Bar();
}

class Sample : IX
{
    public int Foo(int a, int b)
    {
        //...

        return 0;
    }

    public void Bar()
    {
        //...
    }
```

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

Yapılar türetmeye kapalıdır ancak arayüz desteğine açıktır. Yani bir yapı bir sınıftan türetilemez ancak çeşitli arayüzleri destekleyebilir. Örneğin:

```
interface IX
{
    int Foo(int a, int b);
    void Bar();
}

struct Test : IX
{
    public int Foo(int a, int b)
    {
        //...

        return 0;
    }

    public void Bar()
    {
        //...
    }
    //...
}
```

Bir sınıf ya da yapı birden fazla arayüzü destekleyebilir. Sınıflarda taban listede önce sınıf ismi belirtilmek zorundadır. Sonra arayüz isimleri herhangi bir sırada belirtilebilir. Örneğin:

```
interface IY
{
    void Bar();
}

class A
{
    //...
}


class B : A, IX, IY
{
    public void Foo()
    {
        //...
    }

    public void Bar()
    {
        //...
    }
    //...
}
```

Arayüzlerin taban listesinde hangi sırada yazılmış olduğunun bir önemi yoktur.

Bir arayüz türünden referanslar bildirilebilir. Ancak new operatörüyle nesneler yaratılamaz. Örneğin:

```
IX ix;               // geçerli
ix = new IX();      // error!
```

C#'ta sınıf ya da yapılardan onların desteklediği arayüzlere otomatik tür dönüştürmesi vardır. Yani bir sınıf referansı ya da yapı nesnesi o sınıfın ya da yapının desteklediği arayüz referansına atanabilir. Örneğin:

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

ix = s;     // geçerli Sample sınıfı IX arayüzünü destekliyor
```

Arayüzler çokbiçimli mekanizmaya sahiptir. Arayüz metotları sanal değildir. Bunlar override da edilmezler. Ancak yine de bunlar doğuştan çokbiçimli mekanizmaya dahildirler. Yani bir arayüz referansıyla arayüz metodunu çağırdığımızda arayüzün dinamik türüne ilişkin sınıf ya da yapının ilgili metodu çağrılır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            IX ix;
            Sample s = new Sample();

            ix = s;         // geçerli Sample sınıfı IX arayüzünü destekliyor
            ix.Foo();       // Sample.Foo çağrılır

        }
    }

    interface IX
    {
        void Foo();
    }

    class Sample : IX
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
        //...
    }
}
```

Örneğin bir metodun parametresi bir arayüz türünden olabilir. Bu durumda biz o metodu o arayüzü destekleyen herhangi bir sınıf ya da yapıyla çağırabiliriz. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s = new Sample();
            Mample m = new Mample();

            DoSomething(s);
            DoSomething(m);
        }

        public static void DoSomething(IX ix)
        {
            ix.Foo();
        }
    }

    interface IX
    {
        void Foo();
```

```
    }

    class Sample : IX
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
        //...
    }

    class Mample : IX
    {
        public void Foo()
        {
            Console.WriteLine("Mample.Foo");
        }
    }
}
```

Bir sınıf bir arayüzü destekliyorsa o sınıftan türetilmiş olan sınıflar da o arayüzü destekliyor durumdadır. Yani örneğin Sample sınıfı IX arayüzünü destekliyorsa Sample sınıfından türetilen Mample sınıfı da bu arayüzü destekliyor durumda olur. Biz de türemiş sınıf türünden referansı bu arayüz referansına atayabiliriz. Tabi bu arayüz referansıyla ilgili metot çağrıldığında taban sınıftaki desteklenen metot çağrılacaktır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            IX ix = new Mample();       // Sample IX'i desteklediği için Mample da destekliyor

            ix.Foo();                   // Sample.Foo çağrılır
        }
    }

    interface IX
    {
        void Foo();
    }

    class Sample : IX
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
        //...
    }

    class Mample : Sample
    {
        //...
    }
}
```

Türemiş sınıfta aynı isimli aynı parametrik yapıya ilişkin bir metot olsa bile (ki bu durumda uyarıyı kesmek için new anahtar sözcüğü kullanılmak zorundadır) yine arayüz yoluyla çağırma yapıldığında o arayüzü destekleyen taban sınıfın metodu çağrılır. Örneğin:

```
using System;

namespace CSD
{
```

```
    class App
    {
        public static void Main()
        {
            IX ix = new Mample();
            ix.Foo();       // Sample.Foo çağrılır
        }
    }

    interface IX
    {
        void Foo();
    }

    class Sample : IX
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
        //...
    }

    class Mample : Sample
    {
        public new void Foo()
        {
            Console.WriteLine("Mample.Foo");
        }
        //...
    }
}
```

Eğer istenirse türemiş sınıfın da yeniden aynı arayüzü desteklemesi sağlanabilir. Buna "taban sınıfta desteklenen arayüzün türemiş sınıfta yeniden desteklenmesi (interface reimplementation)" denilmektedir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            IX ix = new Mample();       // Sample IX'i desteklediği için Mample da destekliyor

            ix.Foo();                   // Mample.Foo çağrılır
        }
    }

    interface IX
    {
        void Foo();
    }

    class Sample : IX
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
        //...
    }

    class Mample : Sample, IX
    {
        public new void Foo()
        {
```

```
            Console.WriteLine("Mample.Foo");
        }
        //...
    }
}
```

Arayüzlere Neden Gereksinim Duyulmaktadır?

Arayüzler çoklu türetmenin olmadığı C#'a çoklu türetmenin bazı avantajlarını kazandırmaktadır. Bu sayede aralarında türetme ilişkisi olmayan farklı sınıflar aynı arayüz referansına atanabilir. Ve çokbiçimli mekanizmaya dahil edilebilir. Örneğin A ve B sınıfları tamamen bağımsız iki sınıf olsun. Yani bunlar biribirlerinden türetilmiş olmasınlar. Eğer arayüzler olmasaydı biz bu iki sınıfı ortak olarak yalnızca object türüne atayabilirdik. object sınıfının elemanlarını da biz değiştirememekteyiz. Halbuki biz bir türetme ilişkisi içerisinde olmayan A ve B sınıflarının aynı arayüzü desteklemesini sağlayabiliriz. Ve bu iki sınıf referansını da bu arayüz referansına atayabiliriz. Çoklu türetmenin en önemli avantajlarından biri farklı konulara ilişkin birden fazla çokbiçimli eylem oluşturabilmesidir. Örneğin:

```
class Sample : IDisposable, ICloneable, IComparable
{
      //...
}
```

Burada biz Sample sınıfı türünden referansı IDisposable, ICloneable ve IComparable türünden arayüz referanslarına atayabiliriz. Bu arayüzler tamamen farklı amaçlarla oluşturulmuş olabilir.

Arayüzlerin ikinci kullanım gerekçeleri de bunlar sayesinde yapıların da çok biçimli mekanizmaya dahil edilmesidir. Yani yapılar türetme kapalı oldukları halde arayüz desteğine açıktırlar. Böylece biz yapıları çokbiçimli mekaznizmaya dahil edebiliriz.

Arayüzlere İlişkin Ayrıntılar

Bir arayüz bir arayüzden türetilebilir. (Burada "destekleme" değil "türetme" terimi kullanılmaktadır.) Bu durumda türemiş arayüz sanki taban arayüzün elemanlarını içeriyormuş gibi bir etki söz konusu olur. Yani bir sınıf ya da yapı türemiş arayüzü destekliyorsa hem taban arayüzün hem de türemiş arayüzün elemanlarını bulundurmak zorundadır. Türemiş arayüz referansı yoluyla biz hem türemiş arayüzün elemanlarını hem de taban arayüzün elemanlarını kullanabiliriz. Ayrıca türemiş arayüz referansı taban arayüz referansına da doğrudan atanabilmektedir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            IX ix;
            IY iy;
            Sample s = new Sample();

            iy = s;
            ix = iy;

            iy.Foo();
            iy.Bar();
            ix.Foo();
        }
    }

    interface IX
    {
        void Foo();
```

```
    }

    interface IY : IX
    {
        void Bar();
    }

    class Sample : IY
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }

        public void Bar()
        {
            Console.WriteLine("Sample.Bar");
        }
        //...
    }
}
```

Sınıfın ya da yapının taban listesinde hem türemiş arayüz hem de taban arayüz belirtilebilir. Fakat bunun işlevsel bir etkisi yoktur. Yani sınıfın ya da yapının taban kısmında yalnızca türemiş arayüzün belirtilmesiyle hem türemiş hem de taban arayüzün belirtilmesi arasında işlevsel hiçbir fark yoktur. Ancak programcılar bazen okunabilirliği artırmak için taban listede hem türemiş hem de taban arayüzü belirtebilmektedir. Örneğin:

```
class Sample : IY, IX
{
    public void Foo()
    {
        Console.WriteLine("Sample.Foo");
    }

    public void Bar()
    {
        Console.WriteLine("Sample.Bar");
    }
    //...
}
```

Burada sınıfın taban listesinde yalnızca IY belirtilseydi de farkeden birşey olmazdı. Örneğin Microsoft dokümanlarında da bu durumla sık karşılaşılmaktadır:

```
public class ArrayList : IList, ICollection, IEnumerable, ICloneable
{
    //...
}
```

Burada aslında IList arayüzü ICollection arayüzünden, ICollection arayüzü de IEnumerable arayüzünden türetilmiş durumdadır. Yani yukarıdaki bildirimin eşdeğeri şöyle yazılabilirdi:

```
public class ArrayList : IList, ICloneable
{
    //...
}
```

Pekiyi iki arayüzün tesadüfen aynı isimli ve aynı parametrik yapıya sahip metodu varsa ve biz bu iki arayüzü de destekliyorsak ne olur? Bu durumda tek bir bildirim her iki arayüzün gereksinimini de karşılar. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
```

```
        public static void Main()
        {
            Sample s = new Sample();
            IX ix = s;
            IY iy = s;

            ix.Foo();
            iy.Foo();
        }
    }

    interface IX
    {
        void Foo();
        //...
    }

    interface IY
    {
        void Foo();
        //...
    }

    class Sample : IX, IY
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
        //...
    }
}
```

Arayüzler de property elemanlara sahip olabilir. Ancak property bildirimlerinin get ve set bölümler noktalı virgül ile kapatılmalıdır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s = new Sample();
            IX ix = s;

            ix.Count = 10;
            Console.WriteLine(ix.Count);
        }
    }

    interface IX
    {
        int Count
        {
            get;
            set;
        }
        //...
    }

    class Sample : IX
    {
        private int m_count;

        public int Count
        {
            get { return m_count; }
```

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

Eğer arayüzdeki property read-only ya da write-only ise biz onu desteklerken read-write olarak da destekleyebiliriz. Tabi arayüzdeki property read-write ise biz onu yalnızca read-write olarak desteklemek zorundayız.

Arayüz Elemanlarının Açıkça Desteklenmesi (Explicit Implementation)

Şimdiye kadar biz arayüz elemanlarını normal biçimde destekledik. Arayüz elemanları açıkça (explicit) da desteklenebilir. Açıkça destekleme sırasında erişim belirleyici anahtar sözcük yazılmaz (yazılırsa error oluşur). Arayüz elemanı arayüz ismi ve eleman ismiyle niteliklendirilerek belirtilir. Örneğin:

```
interface IX
{
    void Foo();
}

class Sample : IX
{

    void IX.Foo()
    {
        Console.WriteLine("Sample.IX");
    }
    //...
}
```

Açıkça desteklenmiş olan arayüz elemanları isim araması sırasında görülmez. Yani açıkça desteklenen metotlar ya da property'ler sınıfın ya da yapının içerisinde doğrudan ya da o sınıf türünden referanslar ya da yapı türünden değişkenlerle kullanılamazlar. Bunlar yalnızca çokbiçimli olarak arayüz referanslarıyla kullanılabilirler. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s = new Sample();
            IX ix;

            ix = s;

            //s.Foo();        // error!
            ix.Foo();       // geçerli
        }
    }

    interface IX
    {
        void Foo();
    }

    class Sample : IX
    {

        void IX.Foo()
        {
            Console.WriteLine("Sample.IX");
        }
```

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

Açıkça desteklenen eleman adeta sınıfın elemanı değilmiş gibidir. Bunlar yalnızca ilgili arayüz yoluyla çokbiçimli mekanizmayla kullanılabilirler. Bunun dışında bunlar aşağıdan sınıf içerisinden görülemezler.

Bir arayüz elemanı yalnızca normal, yalnızca açıkça ya da hem normal hem de açıkça desteklenebilir. Eğer arayüz elemanı yalnızca normal desteklemişse o hem o sınıf ya da yapı türünden değişkenlerle hem de arayüz referanslarıyla çokbiçimli olarak kullanılabilir. Eğer arayüz elemanları yalnızca açıkça desteklenmişse o elemanlar yalnızca arayüz yoluyla kullanılabilirler. Eğer arayüz elemanları hem normal hem de açıkça desteklenmişlerse aşağıdan çağırmalarda normal desteklenen, arayüz yoluyla çağırmada açıkça desteklenen çağrılır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s = new Sample();
            IX ix;

            ix = s;

            s.Foo();        // Normal olan çağrılır
            ix.Foo();       // Açıkça desteklenen çağrılacak
        }
    }

    interface IX
    {
        void Foo();
    }

    class Sample : IX
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo (Normal)");
        }

        void IX.Foo()
        {
            Console.WriteLine("Sample.Foo (Açıkça)");
        }
        //...
    }
}
```

Açıkça destekleme sayesinde iki farklı arayüzün aynı isimli elemanları varsa bunlar için ayrı metotlar yazılabilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s = new Sample();
            IX ix;
            IY iy;
```

```
            ix = s;
            iy = s;

            ix.Foo();
            iy.Foo();
        }
    }

    interface IX
    {
        void Foo();
        //...
    }

    interface IY
    {
        void Foo();
        //...
    }

    class Sample : IX, IY
    {
        void IX.Foo()
        {
            Console.WriteLine("Sample.Foo (IX)");
        }

        void IY.Foo()
        {
            Console.WriteLine("Sample.Foo (IX)");
        }
        //...
    }
}
```

Arayüzlerle İlgili Tür Dönüştürmeleri

Arayüzlerle ilgili dçnüştürmeler dört durum olarak ele alınabilir:

1) Bir sınıf referansının ya da yapı değişkeninin onun desteklediği bir arayüz referansına dönüştürülmesi.

2) Bir arayüz referansının herhangi bir sınıf türüne dönüştürülmesi

3) Bir arayüz referansının başka bir arayüz referansına dönüştürülmesi

4) Bir sınıf referansının onun desteklemediği bir arayüz referansına dönüştürülmesi

Bilindiği gibi sınıflardan ve yapılardan onların desteklediği arayüz türlerine otomatik (implicit) dönüştürme zaten vardır. Yani bir sınıf türünden referans o sınıfın desteklediği bir arayüz referansına atanabilir.

Bir arayüz referansı herhangi bir sınıf türüne tür dönüştürme operatörü ile dönüştürülmek istendiğinde (downcast) derleme aşamasından her zaman başarıyla geçilir. Ancak programın çalışma zamanı sırasında ayrıca bir haklılık kontrolü yapılmaktadır. Öyle ki dönüştürülmek istenen arayüz referansının dinamik türü dönüştürülmek istenen türü içermiyorsa exception oluşur (InvalidCastException). Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            IX ix = new Sample();       // Otomatik dönüştürme
```

```
            Sample s;

            s = (Sample)ix;         // Derleme aşamasındna geçilir.
                                    // Çalışma zamanı sırasında kontrol yapılır.
            s.Foo();
        }
    }

    interface IX
    {
        void Foo();
        //...
    }

    class Sample : IX
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
    }
}
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            IX ix = new Sample();       // Otomatik dönüştürme
            Random r;

            r = (Random)ix;         // Derleme aşamasındna geçilir. InvalidCastException oluşur
        }
    }

    interface IX
    {
        void Foo();
        //...
    }

    class Sample : IX
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
    }
}
```

Pekiyi eğer dönüştürülecek sınıf o arayüzü desteklemiyorsa neden denetim derleme aşamasında yapılamıyor? Çünkü sınıf o arayüzü desteklemiyor olsa bile, o arayüz referansında o sınıftan türetilmiş olan ve o arayüzü destekleyen bir nesnenin adresi olabilir. Bu durumda dönüştürme geçerli olacaktır. Tabii yapılar için ve sealed sınıflar için aynı durum söz konusu değildir. Bunlarda denetim gerçekten derleme aşamasında yapılır. Çünkü bunlardan zaten türetme yapılamamaktadır. Yani biz arayüzü bir yapıya ya da sealed bir sınıfa dönüştürmek istersek denetim yalnızca derleme aşamasında yapılmaktadır.

Bir arayüz referansı tür dönüştürme operatörüyle başka bir arayüz türüne dönüştürülmek istenebilir. Bu durumda da her zaman derleme aşamasından başarıyla geçilir. Kontrol yine programın çalışma zamanı sırasında yapılır. Çalışma zamanı sırasında dönüştürülecek referansın dinamik türünün dönüştürülecek arayüzü destekleyip desteklemediğine bakılmaktadır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            IX ix;
            IY iy;
            Sample s = new Sample();

            ix = s;         // geçerli, otomatik dönüştürme
            iy = (IY)ix;    // derleme aşamasından geçilir, exception oluşmaz

            iy.Bar();
        }
    }

    interface IX
    {
        void Foo();
    }

    interface IY
    {
        void Bar();
    }

    class Sample : IX, IY
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }

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

Fakat örneğin Sample sınıfı IY arayüzünü desteklemiyor olsun:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            IX ix;
            IY iy;
            Sample s = new Sample();

            ix = s;         // geçerli, otomatik dönüştürme
            iy = (IY)ix;    // derleme aşamasından geçilir, exception oluşur!

            iy.Bar();
        }
    }

    interface IX
    {
        void Foo();
    }
```

```
    interface IY
    {
        void Bar();
    }

    class Sample : IX
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
    }
}
```

Herhangi bir sınıf türünden referans o sınıfın desteklemediği bir arayüz türüne tür dönüştürme operatörü ile dönüştürülmek istenebilir. Bu durumda derleme aşamasından her zaman başarıyla geçilir. Fakat programın çalışma zamanı sırasında referansın dinamik türünün ilgili arayüzü destekleyip desteklemediğine bakılır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            object o = new Sample();
            IX ix;

            ix = (IX)o;     // derleme aşamasından geçilir, çalışma zamanında exception oluşmaz
            ix.Foo();       // Sample.Foo çağrılır
        }
    }

    interface IX
    {
        void Foo();
    }

    class Sample : IX
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
    }
}
```

Örneğin aynı arayüzüyü destekleyen değişik türden sınıf nesneleri bir ArrayList'e yerleştirilmiş olsun:

```
using System;
using System.Collections;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            ArrayList al = new ArrayList();

            al.Add(new Sample());
            al.Add(new Mampl());

            foreach (IX ix in al)        // geçerli, foreach dönüştürmeyi tür dönüştürme operatörüyle
yaapar
```

```
                ix.Foo();
        }
    }

    interface IX
    {
        void Foo();
    }

    class Sample : IX
    {
        public void Foo()
        {
            Console.WriteLine("Sample.Foo");
        }
    }

    class Mampl : IX
    {
        public void Foo()
        {
            Console.WriteLine("Mample.Foo");
        }
    }
}
```

Ancak bir yapı değişkeni o yapının desteklemediği bir arayüz türüne tür dönüştürme operatörüyle dönüştürülmeye çalışılırsa hata derleme aşamasında oluşur. Çünkü yapılar türetmeye kapalıdır. Aynı durum sealed sınıflar için de geçerlidir.

.NET'te Çok Kullanılan Bazı Arayüzler

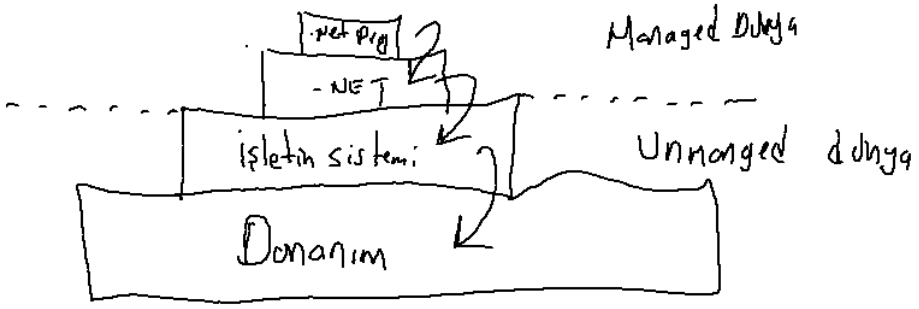
Bu bölümde .NET'te çok kullanılan birkaç arayüz hakkında bilgi verilecektir.

IDisposable Arayüzü

IDisposable arayüzü programlarda çok sık karşımıza çıkar. Bu arayüzün Dispose isimli tek bir metodu vardır:

```
interface IDisposable
{
      void Dispose();
}
```

IDisposable arayüzü yönetilmeyen (unmanaged) kaynakların boşaltılması için kullanılmaktadır. Bilindiği .NET işletim sisteminin üzerine kurulan bir ortamdır. Bizim C# programlarımız da .NET ortamında çalışmaktadır. .NET ortamında çalışan kodlara "maneged" kodlar, .NET ortamında çalışmayan kodlara da "unmanaged" kodlar denilmektedir.



Managed kodlar .NET tarafından izlenir. Bunların tahsis ettiği kaynaklara yönetilen (managed) kaynaklar

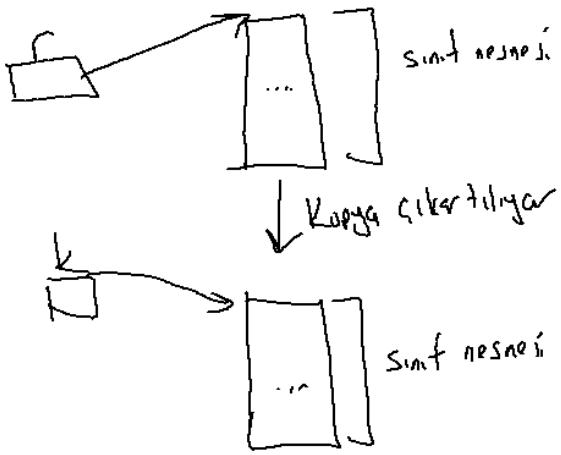
denilmektedir. Yönetilen kaynakların boşaltımı .NET tarafından otomatik olarak yapılabilmektedir. Ancak .NET dünyasından dışında doğrudan işletim sistemi üzerinde çalışan kodlar da vardır. (Örneğin C'de yazılmış program). Bunlara .NET dünyasında "yönetilmeyen (unmanaged)" kodlar denilmektedir. Ayrıca işletim sisteminin kendisi de .NET bakımından yönetilmeyen kodlara sahip bir sistemdir. İşte biz işletim sistemi düzeyinde birtakım tahisatlar yaptığımızda da bunlar .NET tarafından otomatik olarak görülmezler ve serbest bırakılamazlar. Bunları bizim kendimizin serbest bırakması gerekir. İşte IDisposable arayüzünün Dispose metodu bu yönetilmeyen (unmanaged) kaynakları serbest bırakmaktadır. Bu nedenle bir ne zaman IDisposable arayüzünü destekleyen bir sınıf ya da yapı kullansak işimiz bittiğinde Dispose metodunu çağırmalıyız.

Pekiyi IDisposable arayüzünü destekleyen bir sınıf ya da yapıyı kullandıktan sonra Dispose metodunu çağırmazsak ne olur? Çoğu kez bu sınıfların bitiş metotları (destructors) Dispose metodunu çağırarak boşaltımı yapmaktadır. Ancak burada bir gecikme söz konusu olabilir. En iyi teknik programcının Dispose metodunu kendisinin çağırmasıdır.

ICloneable Arayüzü

Bu arayüz bir nesnenin kopyasından çıkarmak için kullanılır. Bir referansın gösterdiği nesnenin aynısından bir tane daha oluşturmak istiyorsak bu arayüzün sunduğu Clone metodundan faydalanırız. ICloneable arayüzünün Clone isimli tek bir metodu vardır:

```
interface ICloneable
{
      object Clone();
}
```

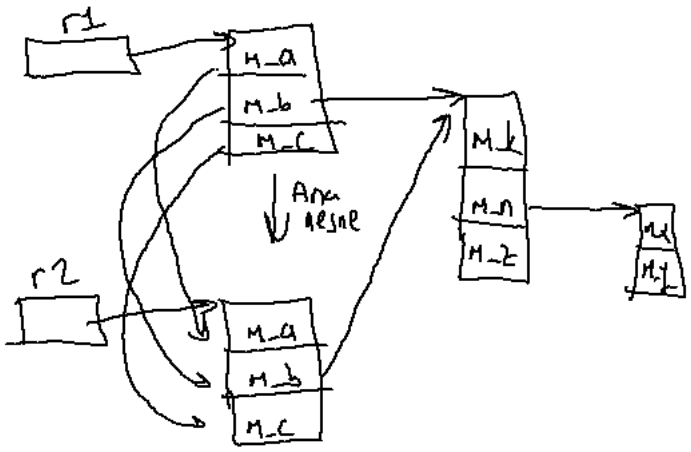


Bir nesnenin kopyasını oluşturmak demek, aynı türdne yeni bir nesne yaratıp onun veri elemanlarına bire bir diğerinin veri elemanlarını atamak demektir. Bir nesnenin kopyasını herhangi bir kişi oluşturamaz. Çünkü sınıfın private veri elemanlarına dışarıdan erişilememektedir. O halde ancak o sınıfı yazanlar nesnenin kopyasını oluşturabilirler. İşte ICloneable arayüznden gelen Clone metodu bunu yapmaktadır. Örneğin:

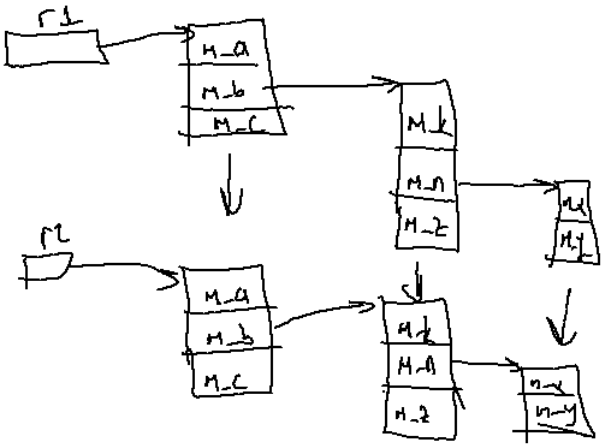
```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Person per1 = new Person("Kaan Aslan", 123);
            Person per2;

            per2 = (Person)per1.Clone();

            Console.WriteLine(per1);
```

```
            Console.WriteLine(per2);
        }
    }

    class Person : ICloneable
    {
        private string m_name;
        private int m_no;

        public Person()
        { }

        public Person(string name, int no)
        {
            m_name = name;
            m_no = no;
        }

        public override string ToString()
        {
            return string.Format("{0}, {1}", m_name, m_no);
        }

        public object Clone()
        {
            Person per = new Person();
            per.m_name = m_name;
            per.m_no = m_no;

            return per;
        }
    }
}
```

Klonlama sığ (shallow) ve derin (deep) olmak üzere ikiye ayrılmaktadır. Sığ klonlamada yalnızca ana nesnenin veri elemanları kopyalanır. Eğer sınıfın bir referans veri elemanı varsa onun gösterdiği nesnenin de kopyası çıkartılmaz. Örneğin:



Derin klonlamada ise özyinelemeli olarak tüm nesnelerin klonlarından çıkartılır. Örneğin:



ICloneable arayüzünü destekleyen bir sınıf ya da yapının derin mi, yoksa sığ mı kopyalama yapacağı dokümanlara bakılarak tespit edilmelidir. Örneğin ArrayList ICloneable arayüzünü destekler fakat sığ klonlama yapmaktadır. Yani bir ArrayList nesnesinin kopyasındna çıkardığımızda yalnızca ana nesnenin kopyasından çıkarmış oluruz. Oraya yerleştirdiğimiz elemanların da kopyasını oluşturmayız. Örneğin:

```
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);

            ArrayList al2 = (ArrayList)al.Clone();
            foreach (int x in al2)
                Console.Write("{0} ", x);
            Console.WriteLine();
        }
    }
}
```

Anımsanacağı gibi C#'ta tüm diziler system.Array isimli bir sınıftan türetilmiş durumdadır. İşte bun sınıf da ICloneable arayüzünü desteklemektedir. Yani biz herhangi bir türdne diziyi Clone metodu ile kopyalayabiliriz. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int[] a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            int[] b;

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

Dosya İşlemleri

İkincil belleklerde organize edilmiş alanlara dosya (file) denir. Dosyaların isimleri ve özellikleri vardır. Dosya işlemleri aslında işletim sistemi tarafından yapılır. NET'in bu konudaki sınıfları dolayları olarak işletim sisteminin API fonksiyonlarını çağırmaktadır.

Bir dosyanın yerini beliren yazısal ifadeye "yol ifadesi (path)" denilmektedir. Windows'ta dizin geçişleri '\' karakteri ile UNIX/Linux sistemlerinde '/' ile belirtilir. Windows sistemlerinde ayrıca bir de sürücü (drive) kavramı vardır. UNIX/Linux sistemlerinde sürücü kavramı yoktur. Windows sistemlerinde her sürücünün ayrı bir kökü ve dizin ağacı vardır. Sürücünün kök dizini onun en dış dizinidir.

Yol ifadeleri mutlak (absolute) ve göreli (relative) olmak üzere ikiye ayrılmaktadır. Eğer sürücü ifadesinden sonraki (yol ifadesinde sürücü de belirtilmeyebilir) ilk karakter '\' ise böyle yol ifadelerine mutlak, değilse göreli yol ifadeleri denilmektedir. Örneğin:

```
"c:\a\b\c.dat"   --> mutlak yol ifadesi
"\x\y\z.txt"     ---> mutlak yol ifadesi
"x\y\z.txt"      ---> göreli
"x.txt"          ---> göreli
```

Çalışmakta olan her programın (yani prosesin) bir çalışma dizini (current working directory) vardır. Programın çalışma dizini göreli yol ifadelerinin çözülmesi için orijin belirtir. Örneğin, programımızın çalışma dizini "c:\temp" olsun. Biz bu programda "x\y\z.dat" biçiminde bir yol ifadesi kullanırsak toplamda "c:\temp\x\y\z.dat" dosyasını belirtmiş oluruz. Prosesin çalışma dizini istenildiği zaman değiştirilebilir. Ancak işin başında .exe dosyanın bulunduğu dizindir.

Mutlak yol ifadeleri kök dizinden itibaren çözülür. Eğer yol ifadesinde sürücü belirtilmemişse prosesin çalışma dizinin bulunduğu sürücü o mutlak yol ifadesindeki sürücü olarak alınır. Örneğin prosesin çalışma dizini "d:\temp" olsun. "\a\b\c.dat" yol ifadesi d'nin kök dizininden itibaren yol belirtir.

Bir programın (yani prosesin) çalışma dizini System isim alanı içerisindeki Environment sınıfının static read/write CurrentDirectory property'si ile alınıp değiştirilebilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Console.WriteLine(Environment.CurrentDirectory);
            Environment.CurrentDirectory = @"c:\windows";
            Console.WriteLine(Environment.CurrentDirectory);
        }
    }
}
```

Windows'ta dosya ve dizin isimlerinin büyük harf küçük harf dutarlılığı yoktur. Windows dosyanın ismini bizim belirttiğimiz gibi saklar. Ancak işleme sokarken büyük harf küçük harf farkını dikkate almaz. Ancak UNIX/Linux sistemlerinde dosya ve dizin isimlerinin büyük harf küçük harf duyarlılığı vardır.

Yol iafadelerinde kullanabileceğimiz iki özel dizin ismi vardır. Bunlar "." ve ".." isimleridir. "." prosesin çalışma dizinini, ".." ise üst dizini belirtir.

File Sınıfı

File sınıfı bütün metotları static olan bir sınıftır. System.IO isim alanı içerisindedir. File sınıfı bir dosya üzerinde bütünsel işlem yapan metotlara sahiptir. Önemli elemanları şunlardır:

- Exists metodu bir dosya var mı yok mu diye bakar:

```
public static bool Exists(
        string path
)
```

Örneğin:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Console.WriteLine(File.Exists(@"c:\windows\notepad.exe") ? "Var" : "Yok");
        }
    }
}
```

Örneğin:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Console.WriteLine(Environment.CurrentDirectory);
            Console.WriteLine(File.Exists(@"notepad.exe") ? "Var" : "Yok"); // Yok

            Environment.CurrentDirectory = @"c:\windows";
            Console.WriteLine(File.Exists("notepad.exe") ? "Var" : "Yok");  // Var

            Environment.CurrentDirectory = @"c:\";
            Console.WriteLine(File.Exists(@"windows\notepad.exe") ? "Var" : "Yok");  // Var
        }
    }
}
```

- Sınıfın Copy metotları dosya kopyalamak için kullanılır.

```
public static void Copy(
        string sourceFileName,
        string destFileName
)
```

Eğer dosya varsa exception oluşur. Fakat 3 parametreli Copy ile varsa Copy'nin olan dosyayı ezmesi de sağlanabilir.

```
public static void Copy(
        string sourceFileName,
        string destFileName,
        bool overwrite
)
```

Örneğin:

```
using System;
using System.IO;
```

```
namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                File.Copy(@"c:\windows\notepad.exe", "notepad.exe");
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

- File sınıfının Delete Metodu ile dosyayı silebiliriz. Ancak dosya yoksa bu metot exception throw etmemektedir.

```
public static void Delete(
       string path
)
```

Örneğin:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                File.Delete("notepad.exe");
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

- File sınıfının ReadAllText isimli metodu bir text dosyadaki tüm yazıyı okur ve bize onu bir string olarak verir:

```
public static string ReadAllText(
        string path
)
```

Örneğin:

```
using System;
using System.IO;

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

```
            {
                string s;

                s = File.ReadAllText(@"..\..\Sample.cs");
                Console.WriteLine(s);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

- File sınıfının WriteAllText metodu bizden bir string ve bir de yol ifadesi alır. O yol ifadesine ilişkin dosyayı yaratarak o yazıyı onun içerisine yazar. Dosya zaten varsa exception oluşmaz. Örneğin:

```
public static void WriteAllText(
        string path,
        string contents
)
```

Örneğin:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                string s = "Ali veli selami";

                File.WriteAllText("x.txt", s);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

- Eğer dosyanın içerisinde yazı yoksa onu okumak için ReadAllBytes metodu kullanılır:

```
public static byte[] ReadAllBytes(
        string path
)
```

Tabi metot bize okunan bilgileri byte dizisi olarak verir. Benzer biçimde byte dizisi içerisindekileri dosyaya yazan WriteAllBytes metodu da vardır:

```
public static void WriteAllBytes(
        string path,
        byte[] bytes
)
```

- Sınıfın ReadAllLines isimli metodu text dosyayı satır satır okur, her bir satırı bir stirng dizsine yerleştirir:

```
public static string[] ReadAllLines(
        string path
```

```
)
```

Örneğin:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                foreach (string line in File.ReadLines(@"..\..\Sample.cs"))
                    Console.WriteLine(line);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Bu metodun WriteAllLines isimli ters işlemi yapan bir biçimi de vardır:

```
public static void WriteAllLines(
        string path,
        string[] contents
)
```

Örneğin:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            string[] lines = { "Ali", "Veli", "Selami" };
            try
            {
                File.WriteAllLines("test.txt", lines);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

FileInfo Sınıfı

FileInfo sınıfı da belirli bir dosya üzerinde temel işlemleri yapmak için kullanılmaktadır. Aslında FileInfo sınıfı ile File sınıfı işlevsellik bakımından birbirlerine çok benzer. File sınıfı tüm elemanları static olan bir sınıftır. Halbuki FileInfo sınıfının elemanları static olmayan elemanlardır. FileInfo sınıfı ile çalışırken önce başlangıç metodunda bir yol ifadesi verilir. Yol ifadesi ile verilen dosyanın bulunuyor olması gerekmez. Sonra sınıfın elemanları ile uygun işlemler yapılır. FileInfo sınfı File sınıfıyla benzer metotlara sahiptir. Örneğin:

```
using System;
```

```
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                FileInfo fi = new FileInfo(@"..\..\Sample.cs");

                Console.WriteLine(fi.Exists ? "Var" : "Yok");
                if (fi.Exists)
                    Console.WriteLine(fi.LastWriteTime.ToString());
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Directory Sınıfı

Directory sınıfı da tıpkı File sınıfı gibi static metotlardan oluşmaktadır. Nasıl File sınıfı bir dosya üzerinde işlem yapan metotlara sahipse Directory sınıfı da bir dizin üzerinde işlem yapan metotlara sahiptir. Directory sınıfının önemli elemanları şunlardır:

- Exists metodu yine bir dizinin olup olmadığını belirlemekte kullanılır.

- CreateDirectory metodu dizin yaratmakta kullanılır. Örneğin:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                Directory.CreateDirectory("test");
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

- Delete metotları dizini silmek için kullanılır. Örneğin:

```
using System;
using System.IO;

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

```
            {
                Directory.Delete("test");
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Ancak tek parametreli Delete metodu içi boş olan dizinleri siler. Eğer dizinin içi doluysa iki parametreli Delete metodu kullanılmalıdır:

```
public static void Delete(
        string path,
        bool recursive
)
```

İçi dolu dizinleri silmek için ikinci parametre true girilmelidir.

- Daha önceden de gördüğümüz gibi GetFiles metodu dizin içerisindeki tüm dosyaları, GetDirectories metodu dizin içerisindeki tüm dizinleri elde etmekte kullanılır.

- Sınıfın GetLogicalDrives isimli metodu o andaki sürücüleri kök ifadesi olarak bir string dizisi biçiminde bize verir. Örneğin:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            try
            {
                foreach (string drive in Directory.GetLogicalDrives())
                    Console.WriteLine(drive);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Bir sürücü hakkında ayrıntılı bilgiler DriveInfo sınıfı ile elde edilebilir. DriveInfo nesnesi sürücünün kök yol ifadesi verilerek yaratılır. Örneğin sınıfın DriveType isimli static olmayan property'si bize sürücünün türünü DriveType isimli enum türünden verir. Örneğin:

DirectoryInfo Sınıfı

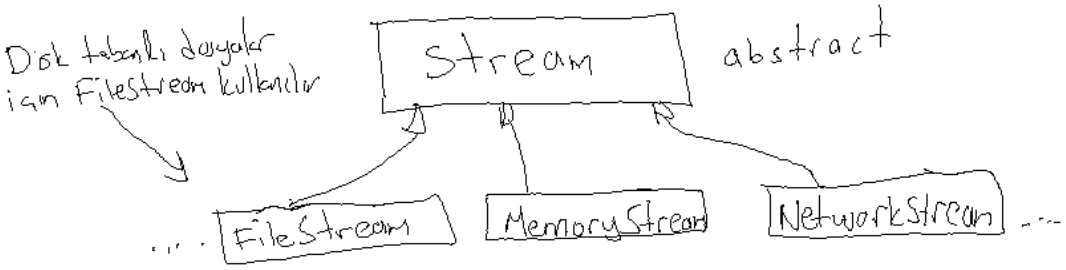
File sınıfı ile FileInfo sınıfı arasındaki ilişki Directory sınıfı ile DirectoryInfo sınıfı arasındaki ilişkiye benzerdir. DirectoryInfo sınıfı da static olamayan elemanlara sahiptir ve aslında Directory sınıfıyla benzer işlevsellik sunar.

Stream Tabanlı Dosya İşlemleri

Yukarıda görmüş olduğumuz File ve FileInfo sınıfları dosyalar üzerinde temel ve bütünsel işlemler yapan sınıflardır. Halbuki çoğu kez büyük bir dosyanın yalnızca bir kısmı üzerinde işlem yapmak isteyebiliriz. Bunun

için stream tabanlı işlemler yapan sınıflar kullanılır. Zaten dosya işlemleri denildiğinde akla bu tür işlemler gelmektedir.

Bir dosya üzerinde işlemler yapmak için önce dosyayı açmak gerekir. İşlemler yapıldıktan sonra dosya kapatılır. .NET'te dosya gibi işlem gören tüm kaynaklar Stream isimli bir sınıftan türetilen sınıflarla temsil edilmiştir:



Normal disk tabanalı dosyalar FileStream sınıfıyla temsil edilmektedir. FileStream sınıfı türünden bir nesne yaratıldığında ilgili dosya da açılmış olur. Dosyanın kapatılması Stream sınıfından gelen Close metoduyla yapılır.

FileStream sınıfının pek çok başlangıç metodu olsa da en çok kullanılanı aşağıdaki metotur.

```
public FileStream(
        string path,
        FileMode mode,
        FileAccess access
)
```

Metodun birinci parametresi açılacak dosyanın yol ifadesini belirtir. İkinci parametre FileMod isimli bir enum türündendir. Burada dosyanın açış modu belirtilir. FileMod elemanları ve anlamları şöyledir:

Mod	**Anlamı**
FileMode.Open	Olan bir dosya bu modda açılabilir. Dosya yoksa exception oluşur
FileMode.Create	Burada dosya yoksa yaratılır ve açılır. Dosya varsa sıfırlanır ve açılır.
FileMode.OpenOrCreate	Dosya varsa olanı açar, yoksa yaratarak açar
FileMode.CreateNew	Dosya varsa exception oluşu. Ancak yoksa yaratılır ve açılır
FileMode.Truncate	Dosya varsa sıfırlanır ve açılır, yoksa exception oluşur
FileMode.Append	Dosya varsa açılır, yoksa yaratılır ve açılır. Bu modda dosyaya her yazılan sona eklenir

Metodun son parametresi FileAccess isimli bir enum türündendir. Bu parametre dosyaya ne yapmak istediğimizi belirtir. Bu enum türünün üç elemanı vardır:

```
FileAccess.Read
FileAccess.Write
FileAccess.ReadWrite
```

Dosya açma işleminni tipik kalıbı şöyledir:

```
using System;
using System.IO;

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

```
            FileStream fs = null;

            try
            {
                fs = new FileStream("test.dat", FileMode.Create, FileAccess.Write);
                //...
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                if (fs != null)
                    fs.Close();
            }
        }
    }
}
```

Dosyaların uzantılarının işletim sistemi için bir önemi yoktur. Uzantı ne olursa olsun dosyaların içerisinde byte yığınları vardır. Biz de temelde dosyalardan byte okuyup byte yazarız.

Dosya Gösterici Kavramı

Dosya içerisindeki her bir byte'a ilk byte sıfır olmak üzere bir offset numarası karşı düşürülmüştür. Yani dosya içerisindeki her bir byte'ın bir numarası vardır. Dosya göstericisi o anda dosyanın hangi offset'inden okuma ya da yazma yapılacağını anlatan bir offset belirtir. Yani dosya göstericisi bir çeşit imleç görevi görmektedir. Örneğin:

Bu örnekte dosya göstericisinin 2'nci offset'i gösterdiğini düşünelim. Biz artık 2 byte'lık bir okuma yaparsak 2 ve 3 numaralı offset'teki byte'ları okuruz. Okuma ve yazma metotları okunan ya da yazılan miktar kadar dosya göstericisini otomatik ilerletmektedir. Dosya açıldığında dosya göstericisi başlangıçta 0'ıncı offset'tedir.

Dosya Göstericisinin EOF Durumu

Dosya göstericisinin dosyanın son byte'ından sonraki byte'ı göstermesi durumuna EOF durumu denir. EOF durumundan okuma yapılamaz. Fakat yazma yapılabilir. Bu durum dosyaya ekleme anlamına gelir. Dosyaya ekleme yapmanın başka bir yolu yoktur. Dosya FileMod.Append moduyla açıldığında her Write işlemi otomatik olarak dosya göstericisini EOF durumuna çekerek yazma yapmaktadır.

Dosyadan Okuma Yazma İşlemi

Dosyadan okuma ve dosyaya yazma yapmak için Stream sınıfından gelen abstract Read ve Write metotları kullanılmaktadır. Read ve Write metotları FileStream sınıfında override edilmiştir:

```
public abstract int Read(
        byte[] buffer,
        int index,
        int count
)

public abstract void Write(
        byte[] buffer,
```

```
        int index,
        int count
)
```

Metotların birinci parametreleri okunacak ya da yazılacak bilginin bulunduğu byte dizisini belirtir. Read metodu okuduklarını bu diziye yerleştirir. Write metodu ise bu dizinin içindekileri dosya göstericisinin gösterdiği yerden itibaren dosyaya yazar. İkinci parametreler dizide birer index belirtir. Dizidki o index'ten itibaren işlem yapılır. Üçüncü parametreler kaç byte yazılıp okunacağını belirtir. Read metodunda dosya göstericisinin gösterdiği yerden itibaren dosyada kalandan daha fazla byte okunmak istenirse Read okuyabildiği kadar byte'ı okur, okuyabildiği byte sayısı ile geri döner. Örneğin:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream fs = null;
            byte[] buf = new byte[30];

            try
            {
                fs = new FileStream(@"..\..\Sample.cs", FileMode.Open, FileAccess.ReadWrite);
                fs.Read(buf, 0, 30);
                Console.WriteLine(BitConverter.ToString(buf));

            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                if (fs != null)
                    fs.Close();
            }
        }
    }
}
```

Görüldüğü gibi Read ve Write metotları byte dizisiyle çalışmaktadır. Eğer biz istediğimiz bir bilgiyi bu metotlarla yazıp okuyacaksak onları byte dizisine dönüştürüp geri almamaız gerekir. İşte bunun için BitConverter sınıfı kullanılmaktadır.

Değerleri Byte Dizisine Dönüştürüp Geri Almak

BitConverter sınıfı static metotlardan oluşan bir sınıftır. Sınıfın bir grup her türden parametre alan overload edilmiş GetBytes metodu vardır. Bunlar ilgili değeri argüman olarak alıp onu byte'larına ayrıştırıp bize byte dizisi olarak verirler. Örneğin:

```
public static byte[] GetBytes(
        int value
)
```

Benzer biçimde sınıfın bir grup ToXXX isimli static metotları vardır. Bunlar da ters işlem yaparlar. Yani bizden byte dizisi alıp, onu birleştirerek XXX türünden değer olarak verirler. Örneğin ToInt32 metodunun parametrik yapısı şöyledir:

```
public static int ToInt32(
        byte[] value,
        int startIndex
```

```
)
```

Metodun birinci parametresi byte türünden diziyi alır, ikinci parametre de dönüştürmenin hangi index'ten itibaren yapılacağını belirtir.

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int a = 12345678, b;
            byte[] buf;

            buf = BitConverter.GetBytes(a);
            b = BitConverter.ToInt32(buf, 0);
            Console.WriteLine(b);
        }
    }
}
```

Örneğin biz 0'dan 100'e kadar sayıları dosyaya şöyle yazabiliriz:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream fs = null;

            try
            {
                fs = new FileStream("test.dat", FileMode.Create, FileAccess.Write);

                for (int i = 0; i < 100; ++i)
                {
                    byte[] buf = BitConverter.GetBytes(i);
                    fs.Write(buf, 0, buf.Length);
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                if (fs != null)
                    fs.Close();
            }
        }
    }
}
```

Bunları da dosyadan şöyle okuyabiliriz:

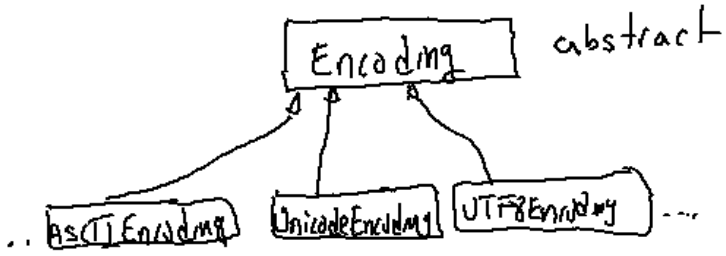
```
using System;
using System.IO;

namespace CSD
```

```
{
    class App
    {
        public static void Main()
        {
            FileStream fs = null;

            try
            {
                fs = new FileStream("test.dat", FileMode.Open, FileAccess.Read);
                byte[] buf = new byte[4];
                int val;

                for (int i = 0; i < 100; ++i)
                {
                    fs.Read(buf, 0, 4);
                    val = BitConverter.ToInt32(buf, 0);
                    Console.Write("{0} ", val);
                }
                Console.WriteLine();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                if (fs != null)
                    fs.Close();
            }
        }
    }
}
```

Yukarıdaki örnekte biz int türden bilgileri byte'larına ayrıştırıp Write metodu ile dosyaya yazdık. Ve sonra onları Read metodu ile okuyup yeniden int türüne dönüştürdük. Burada int değerlerin yazısal olarak değil byte'sal olarak dosyaya yazıldığına dikkat ediniz. Bu dosya bir editörle açıldığında anlamlı bir görüntü oluşmayacaktır. Eğer biz sayırları yazısal olarak dosyaya yazsaydık o zaman anlamlı bir görüntü elde ederdik.

Yazıların byte Dizilerine Dönüştürülmesi

Yazıların byte dizilerine dönüştürülmesi ve geri alınması BitConverter sınıfıyla yapılmamaktadır. Çünkü yazılar söz konusu olduğunda "Encoding" konusu da devreye girmektedir. Bu yüzden yazıların dönüştürülmesi Encoding sınıflarıyla yapılır. Her karakter kodlaması için Encoding isimli abstract sınıftan türetilmiş bir encoding sınıfı bulunur.



Bu durumda örneğin biz bir ASCII yazıyı byte dizisine dönüştürüp alacaksak ASCIIEncoding, UTF8 bir yazıyı byte dizisine dönüştürüp alacaksak UTF8Encoding sınıfını kullanmalıyız. Bu sınıfların GetBytes isimli static olmayan metotları yazıyı byte dizisine dönüştürür. GetString isimli static metotları ise byte dizisini yazıya dönüştürür. Örneğin:

```
using System;
using System.Text;

namespace CSD
```

```
{
    class App
    {
        public static void Main()
        {
            string msg = "Bugün hava çok güzel";
            byte[] buf;
            string str;

            UTF8Encoding ue = new UTF8Encoding();
            buf = ue.GetBytes(msg);
            str = ue.GetString(buf);
            Console.WriteLine(str);
        }
    }
}
```

Anahtar Notlar: UTF8 kodlaması UNICODE tablonun sıkıştırılmış biçimini oluşturmakta kullanılır. UTF8'de UNICODE bir yazıdaki İngilizce karakterler 1 byte ile diğer karakterler birden çok byte ile kodlanırlar. Böylece UTF8 kodlaması ekonomik bir kodlama halini alır. Bu nedenle en çok karşılaşılan karakter kodlamalarından biridir. Hatta C# standartlarına göre C#'ın kaynak dosyası da UTF8 kodlanmış olmak zorudandır.

Ayrıca taban Encoding sınıfının ASCII, UTF, Unicode gibi property elemanları bize ilgili encoding nesnesini yaratıp (singleton kalıbı ile) vermektedir. Yani örneğin:

```
Encoding e = Encoding.ASCII;
```

ile,

```
Encoding e = new ASCIIEncoding();
```

benzer etkiye sahiptir. Dolayısıyla birisi bizden Encoding nesnesi istediğinde biz ona Encoding.UTF8 gibi bir ifadeyle bu nesneyi verebiliriz.

Şimdi bir yazıyı Encoding sınıflarıyla byte dizisine dönüştürüp dosyaya yazalım. Örneğin:

```
using System;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream fs = null;

            try
            {
                fs = new FileStream("test.txt", FileMode.Create, FileAccess.Write);
                string s = "Bugün hava çok güzel";

                byte[] buf = Encoding.UTF8.GetBytes(s);
                fs.Write(buf, 0, buf.Length);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                if (fs != null)
                    fs.Close();
            }
        }
```

```
    }
}
```

Okuma işlemi de benzer biçimde yapılabilir:

```
using System;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream fs = null;
            byte[] buf = new byte[1000];
            int count;
            string s;

            try
            {
                fs = new FileStream("test.txt", FileMode.Open, FileAccess.Read);

                count = fs.Read(buf, 0, 1000);      /* dosyada olan kadarını okur */
                s = Encoding.UTF8.GetString(buf, 0, count);
                Console.WriteLine(s);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                if (fs != null)
                    fs.Close();
            }
        }
    }
}
```

Dosya Göstericisinin Konumlandırılması

Dosya göstericisini konumlandırmak için Stream sınıfındaki abstract Seek metodu kullanılır. Bu metot türemiş sınıflarda override edilmiştir:

```
public abstract long Seek(
      long offset,
      SeekOrigin origin
)
```

Metodun birinci parametresi konumlandırılacak offset'i belirtir. İkinci parametresi konumlandırma orijinini belirtmektedir. SeekOrigin isimli enum türünün üç elemanı vardır:

SeekOrigin.Begin: Bu durumda konumlandırma dosyanın başından itibaren yapılır. Offset >= 0 olmak zorundadır. Örneğin:

```
fs.Seek(3, SeekOrigin.Begin);
```

Burada konumlandırma dosyanın başından itibaren 3'üncü offset'e yapılmaktadır.

SeekOrigin.Current: Bu durumda konumlandırma dosya göstericisinin gösterdiği yerden itibaren yapılır. Pozitif ileri, negatif geri anlamına gelir. Offset pozitif ya da negatif ya da sıfır olabilir. Örneğin:

```
fs.Seek(-1, SeekOrigin.Current);
```

Burada dosya göstericisi eski gösterdiği yerin 1 gerisine konumlandırılmaktadır.

SeekOrigin.End: Bu durumda konumlandırma EOF durumundan itibaren yapılır. Offset <= 0 olmak zorundadır.

Örneğin bir dosyadaki yazının sonuna birşeyler eklemek isteyelim. Bunun için önce dosya göstericisini EOF durumuna çekmemiz gerekir. Bu işlem şöyle yapılabilir:

```
fs.Seek(0, SeekOrigin.End);
```

Örneğin:

```
using System;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream fs = null;

            try
            {
                fs = new FileStream("test.txt", FileMode.Open, FileAccess.Write);
                string s = ". Evet evet çok güzel";

                fs.Seek(0, SeekOrigin.End);

                byte[] buf = Encoding.UTF8.GetBytes(s);
                fs.Write(buf, 0, buf.Length);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                if (fs != null)
                    fs.Close();
            }
        }
    }
}
```

Stream sınıfının Position isimli long türden property elemanı da dosya göstericisini konumlandırmakta kullanılabilir. Tabi bu proerty konumlandırmayı her zaman baştan itibaren yapar. Yani örneğin:

```
fs.Seek(10, SeekOrigin.Begin);
```

çağrısı ile,

```
fs.Position = 10;
```

aynı anlamdadır.

Yazma ve Okuma İşlemini Kolaylaştıran Adaptör Sınıflar

Her türlü bilgiyi dosyaya yazmak için önce byte dizisine dönüştürmek gerekir. Benzer biçimde biz her şeyi byte

dizisi olarak okuyup onu yeniden dönüştürmemiz gerekir. İşte bunu yapan çeşitli adaptör sınıflar vardır.

BinaryWriter Sınıfı

BinaryWriter sınıfı başlangıç metodu yoluyla bizden bir Stream referansı alır. Sınıfın her türden parametre alan Write metotları vardır. Bunlar ilgili değeri byte dizisine dönüştürüp o Stream'in Write metoduyla dosyaya yazarlar. Sınıfın Close isimli metodu alınan Stream'i de Close etmektedir. Tipi kullanım şöyle olabilir:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream fs = null;
            BinaryWriter bw = null;

            try
            {
                fs = new FileStream("test.dat", FileMode.Create, FileAccess.Write);
                bw = new BinaryWriter(fs);

                for (int i = 0; i < 100; ++i)
                    bw.Write(i);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                if (bw != null)
                    bw.Close();
            }
        }
    }
}
```

BinaryReader Sınıfı

BinaryReader sınıfı da çok benzer kullanılmaktadır. Bu sınıf da bizden bir Stream parametresi alır. Sınıfın ReadXXX biçiminde metotları vardır. O metotlar önce bizim verdiğimiz stream'den okumayı yaparlar. Sonra onu BitConverter yoluyla XXX türüne dönüştürüp bize verirler. Örneğin ReadInt32 metodunun parametrik yapısı şöyledir:

```
public virtual int ReadInt32()
```

Örneğin:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream fs = null;
            BinaryReader br = null;
            int val;
```

```
            try
            {
                fs = new FileStream("test.dat", FileMode.Open, FileAccess.Read);
                br = new BinaryReader(fs);

                for (int i = 0; i < 100; ++i)
                {
                    val = br.ReadInt32();
                    Console.Write("{0} ", val);
                }
                Console.WriteLine();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                if (br != null)
                    br.Close();
            }
        }
    }
}
```

StreamReader ve StreamWriter Sınıfları

StreamReader ve StreamWriter sınıfları da yazıyla işlem yapan adaptör sınıflardır. StreamReader ve StreamWriter sınıfları bir Stream referansını ve bir Encoding nesnesini parametre olarak bizden alırlar. Sonra yazıları o Encoding'i ve Stream referansını kullanarak dosyaya yazdırırlar ve dosyadan okurlar. StreamWriter sınıfının Write ve WriteLine metotları yazdırma işlemini yapmaktadır. Bu metotlarla normal sayıları dosyaya yazdırabiliriz. Ancak bu durumda sayılar yazıya dönüştürülerek dosyaya yazdırılır. Yani editörle açtığımızda anlamlı şeyler görürürz.

Örneğin:

```
using System;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            FileStream fs = null;
            StreamWriter sw = null;
            int val;

            try
            {
                fs = new FileStream("test.text", FileMode.Create, FileAccess.Write);
                sw = new StreamWriter(fs, Encoding.UTF8);

                sw.WriteLine("Bu bir deneme");

                for (int i = 0; i < 10; ++i)
                {
                    sw.WriteLine(i);

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

```
            }
            finally
            {
                if (sw != null)
                    sw.Close();
            }
        }
    }
}
```

StreamWriter sınıfının doğrudan yol ifadesi alan bir versiyonu da vardır. Bu kendi içerisinde FileStream sınıfı ile dosyayı açar. Sonra StreamWriter nesnesini bununla oluşturur. Yani kullanımı daha kolaydır. Örneğin:

```
using System;
using System.IO;
using System.Text;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            StreamWriter sw = null;

            try
            {
                sw = new StreamWriter("test.txt", false, Encoding.UTF8);
                sw.WriteLine("Bu bir deneme");

                for (int i = 0; i < 10; ++i)
                {
                    sw.WriteLine(i);

                }
                Console.WriteLine();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                if (sw != null)
                    sw.Close();
            }
        }
    }
}
```

StreamReader sınıfı da benzer olarak text bir dosyadan okuma yapan bir sınıftır. Sınıfın kullanımı diğerlerine çok benzerdir. Sınıfın Stream nesnesi alan ve doğrudan dosyanın yol ifadesini alan başlangıç metotları vardır. Nesne yaratıldıktan sonra yine sınıfın ReadLine metodu ile satır satır okuma yapılabilir ya da ReadToEnd metodu ile dosyanın hepsi tek hamlede okunabilir.

Statik Sınıflar

Statik sınıf kavramı C#'a Framework 2.0 ile sokulmuştur. Sınıfı statik yapmak için sınıf bildiriminin başına static anahtar sözcüğü getirilir. Örneğin:

```
static class Sample
{
      //...
}
```

static bir sınıfın bütün elemanları static olmak zorundadır. Statik sınıfların içerisine static olmayan elemanlar yerleştirilemez. Statik sınıflar okunabilirliği artırmak için dile sokulmuştur. Örneğin Math sınıfı, File sınıfı, BitConverter sınıfı static sınıflardır.

Static sınıflar türünden referanslar bildirilemez. Zaten static olmayan elemanı bulunmayan bir sınıf için referans bildirmenin bir anlamı da yoktur. static sınıflar türünden nesneler de new operatörüyle yatarılamazlar. Statik sınıflar başka sınıflardan türetilemezler ve statik sınıflardan türetme yapılamaz. Yapılar statik yapılamaz. Yalnızca sınıflar statik yapılabilir.

Sealed Sınıflar

Sealed sınıflardan türetme yapılamaz. Yani sealed sınıflar türetmeye kapalıdır. Örneğin Console sınıfı sealed bir sınıftır. Sınıfı yazan kişi mantıksal bakımdan türetme uygun değilse sınıfı sealed yaparak bunu sınıfı kullanacaklara bildirmiş olur. Örneğin:

```
sealed class Sample
{
    //...
}

class Mample : Sample   // error!
{
    //...
}
```

Yapılar türetmeye kapalıdır. Yani onların adeta default olarak sealed olduğu düşünülebilir.

Sealed Override Metotlar ve Property'ler

sealed belirleyicisi yalnızca override belirleyici ile birlikte kullanılabilir. sealed override metotlar ve property'ler ilgili sınıftan türetme yapılsa bile artık daha fazla override edilemezler. Örneğin:

```
using System;

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

    class A
    {
        public virtual void Foo()
        {
            //...
        }
        //...
    }

    class B : A
    {
        public sealed override void Foo()
        {
            //...
        }
        //...
    }

    class C : B
    {
        public override void Foo()      // error!
```

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

Bir metodu ya da proprty'yi artık onun daha fazla override edilmesini istemiyorsak sealed override yapabiliriz.

Partial Sınıflar

Normal olarak aynı isim alanında aynı isimli sınıf birden fazla kez bildirilemez. Ancak Framework 2.0 ile birlikte partial anahtar sözcüğü kullanılarak bir sınıfın birden fazla parça ile bildirilmesi mümkün hale getirilmiştir. Partial sınıflarda sınıfın her parçasının bildiriminde partial anahtar sözcüğünün kullanılması zorunludur. Örneğin:

```
partial class Sample
{
    private int m_a;

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

//...

partial class Sample
{
    public void Bar()
    {
        //...
    }
}
```

Derleyici sınıfın tüm partial bildiirmlerini derleme aşamasında birleştirir ve onu tek bir sınıf olarak ele alır. Tabii yalnızca aynı isim alanın içerisindeki aynı isimli partial sınıflar birleştirilmektedir. Örneğin:

```
namespace A
{
    partial class Sample
    {
        private int m_a;

        public void Foo()
        {
            //...
        }
        //...
    }
    //...
}
namespace B
{
    partial class Sample
    {
        public void Bar()
        {
            //...
        }
    }
    //...
}
```

Buradaki iki Sample aslında farklı sınıflardır ve birleştirilmesi söz konusu değildir. Yukarıdaki örnekten de görüldüğü gibi sınıf tek parça olarak bildirildiği halde yine partial anahtar sözcüğü kullanılabilir. Yani partial

anahtar sözcüğünün kullanılmış olması sınıfın birden fazla parçadan oluşmasını zorunlu hale getirmez.

Partial sınıfların bir kısmı bir kaynak dosyada diğeri başka bir dosyada olabilir.

Partial sınıflar özellikle kod üreten IDE'lerle çalışırken karışıklık oluşmasın diye dile sokulmuştur. Örneğin böylece aynı sınıf farklı kaynak dosyalarda partial olarak bildirilebilmektedir. Bunlardan birine IDE ekleme yaparken diğerine programcı ekleme yapabilir. Böylece programcının yanlışlıkla IDE tarafından eklenen kodu bozması engellenmiş olur.

Operatör Metotları

Operatör metotları sınıf türünden referanslar üzerinde ya da yapı nesneleri üzerinde işlem yapılmasını sağlayan özel metotlardır. Aslında operatör metotları yalnızca okunabilirlik sağlar. Yoksa dile ek bir işlevsellik kazandırmamaktadır. Java'da operatör metotları yoktur. Ancak C++'ta vardır. Operatör metotları olmsaydı biz aynı şeyleri sıradan metotlarla da yaptırabilirdik. Örneğin iki karmaşık sayıyı toplayarak bize yeni bir karmaşık sayı veren bir metodu Complex isimli bir sınıfın static metodu olarak yazabiliriz:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Complex x = new Complex(3, 2);
            Complex y = new Complex(2, 5);
            Complex result;

            result = Complex.Add(x, y);     // result = x + y
            Console.WriteLine(result);
        }
    }

    class Complex
    {
        private double m_real;
        private double m_imag;

        public Complex()
        {
        }

        public Complex(double real, double imag)
        {
            m_real = real;
            m_imag = imag;
        }

        public override string ToString()
        {
            return string.Format("{0}+{1}i", m_real, m_imag);
        }

        public static Complex Add(Complex x, Complex y)
        {
            Complex result = new Complex();

            result.m_real = x.m_real + y.m_real;
            result.m_imag = x.m_imag + y.m_imag;

            return result;
        }
    }
}
```

Burada iki Complex nesnesini toplamak için Complex sınıfının Add isimli static metodu çağrılmıştır:

```
result = Complex.Add(x, y);
```

İşte operatör metotları bu işlemi daha okunabilir yapmak için kullanılır:

```
result = x + y;
```

Başka bir deyişle x + y ifadesini gören derleyici ismine operatör metodu denilen metodu çağırır. x ve y değerlerini bu metoda argüman yollar. Metodun geri dönüş değerini de işlemin sonucu gibi elde eder.

Operatör metotları public ve static olmak zorundadır. Operatör metotlarının geri dönüş değerleri herhangi bir türden olabilir. Operatör metotlarının isimleri operator anahtar sözcüğü ile operatör sembolünden oluşur. Operatör sembolü iki operandlı bir operatöre ilişkinse operatör metodunun parametresi iki tane, tek operandlı bir operatöre ilişkinse bir tane olmak zorudadır. Ayrıca operatör metotlarının parametrelerinden en az biri operatör metodunun yerleştirildiği sınıf ya da yapı türünden olmak zorundadır.

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Complex x = new Complex(3, 2);
            Complex y = new Complex(2, 5);
            Complex result;

            result = x + y;
            Console.WriteLine(result);
        }
    }

    class Complex
    {
        private double m_real;
        private double m_imag;

        public Complex()
        {
        }

        public Complex(double real, double imag)
        {
            m_real = real;
            m_imag = imag;
        }

        public override string ToString()
        {
            return string.Format("{0}+{1}i", m_real, m_imag);
        }

        public static Complex operator +(Complex x, Complex y)
        {
            Complex result = new Complex();

            result.m_real = x.m_real + y.m_real;
            result.m_imag = x.m_imag + y.m_imag;

            return result;
        }
    }
```

```
}
```

Diğer operatörler için de metotları benzer biçimde yazabiliriz:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Complex x = new Complex(3, 2);
            Complex y = new Complex(2, 1);
            Complex result;

            result = x * y;
            Console.WriteLine(result);
        }
    }

    class Complex
    {
        private double m_real;
        private double m_imag;

        public Complex()
        {
        }

        public Complex(double real, double imag)
        {
            m_real = real;
            m_imag = imag;
        }

        public override string ToString()
        {
            return string.Format("{0}+{1}i", m_real, m_imag);
        }

        public static Complex operator +(Complex x, Complex y)
        {
            Complex result = new Complex();

            result.m_real = x.m_real + y.m_real;
            result.m_imag = x.m_imag + y.m_imag;

            return result;
        }

        public static Complex operator -(Complex x, Complex y)
        {
            Complex result = new Complex();

            result.m_real = x.m_real - y.m_real;
            result.m_imag = x.m_imag - y.m_imag;

            return result;
        }

        public static Complex operator *(Complex x, Complex y)
        {
            Complex result = new Complex();

            result.m_real = x.m_real * y.m_real - x.m_imag * y.m_imag;
            result.m_imag = x.m_real * y.m_imag + x.m_imag * y.m_real;

            return result;
```

```
        }
    }
}
```

Şüphesiz operatör metotları kombine edilebilir. Örneğin:

```
x = y + z + k;
```

Burada önce y + z işlemi operatör metoduna yaptırılır. Buradan elde edilecek sonuç bu kez k ile birlikte yeniden operatör metoduna sokulur.

Pekiyi derleyici bir operatör gördüğünde ne yapar? Bir operatörle karşılaşan derleyici önce operatörün operandlarının türlerine bakar. Eğer her iki operand da temel türlere ilişkinse operatör metodu devreye girmez. Küçük tür büyük türe dönüştürülerek işlem yapılır. Eğer metotlardan en az biri bir sınıf ya da yapı türündense derleyici bu sınıf ya da yapılar içerisinde bu işi yapabilecek operatör metodu araştırır. Bulursa operandları ona argüman yollar ve metodu çağırır. Metodun geri dönüş değerini de işlem sonucu olarak belirler. Örneğin:

```
A a;
B b;
//...
a <op> b
```

işlemi yapılmış olsun. Burada derleyici A ve B sınıflarındaki tüm operator <op> metotlarını aday metotlar olarak belirler. Bunlardan uygun olanı ve en uygun olanı daha önce açıklandığı gibi seçer.

***, /, + ve - Operatörlerine İlişkin Operatör Metotlarının Yazımı**

Bu operatör metotlarının iki parametresi olmalıdır. Geri dönüş değerleri herhangi bir türden olabilirse de aynı sınıf ya da yapı türünden olması kombine edilmeyi sağlar.

Karşılaştırma Operatörlerine İlişkin Operatör Metotlarının Yazılması

Bu operatörlerin de iki operandları bulunmalıdır. Bunların geri dönüş değerleri herhangi bir türden olabilirse de bool türden olması en anlamlı durumdur. Karşılaştırma operatör metotları çiftler haline yazılmak zorundadır. Yani çiftlerden biri yazılmışsa diğeri de yazılmak zorundadır. Çiftler şunlardır:

```
>  <
>= <=
== !=
```

Örneğin bir sayıyı tutan bir Number sınıfı için örnek verebiliriz:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Number a = new Number(3), b = new Number(7), c = new Number(5);

            if (b - a > c)
                Console.WriteLine("evet");
            else
                Console.WriteLine("hayır");
        }
    }

    class Number
    {
```

```
private double m_val;

public Number()
{ }

public Number(double val)
{
    m_val = val;
}

public override string ToString()
{
    return m_val.ToString();
}

public static Number operator +(Number x, Number y)
{
    Number result = new Number();

    result.m_val = x.m_val + y.m_val;

    return result;
}

public static Number operator -(Number x, Number y)
{
    Number result = new Number();

    result.m_val = x.m_val - y.m_val;

     return result;
}

public static Number operator *(Number x, Number y)
{
    Number result = new Number();

    result.m_val = x.m_val * y.m_val;

    return result;
}

public static Number operator /(Number x, Number y)
{
    Number result = new Number();

    result.m_val = x.m_val / y.m_val;

    return result;
}

public static bool operator >(Number x, Number y)
{
    return x.m_val > y.m_val;
}

public static bool operator <(Number x, Number y)
{
    return x.m_val < y.m_val;
}

public static bool operator >=(Number x, Number y)
{
    return x.m_val >= y.m_val;
}

public static bool operator <=(Number x, Number y)
{
    return x.m_val <= y.m_val;
}
```

```
        public static bool operator ==(Number x, Number y)
        {
            return x.m_val == y.m_val;
        }

        public static bool operator !=(Number x, Number y)
        {
            return x.m_val != y.m_val;
        }
    }
}
```

++ ve -- Operatörlerine İlişkin Operatör Metotlarının Yazımı

Bu operatörler tek operandlı olduğu için bunlara ilişkin operatör metotları da tek parametreli olmak zorundadır. Bu operatörlerin önek ve sonek kullanımlarının aynı etkiyi yaratması için operatör metotlarının yazımında şunlara dikkat edilmelidir: Operatör metodu aldığı parametredeki nesneyi artırıp azaltmamalıdır. Yeni bir artırılmış ya da azaltılmış nesne oluşturup onunla geri dönmelidir. Örneğin Number sınıfı için aşağıdaki gibi ++ ve -- operatör metotları yazılabilir:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Number a, b;

            a = new Number(10);
            b = ++a;
            Console.WriteLine("a = {0}, b = {1}", a, b);

            a = new Number(10);
            b = a++;
            Console.WriteLine("a = {0}, b = {1}", a, b);
        }
    }

    class Number
    {
        private double m_val;

        public Number()
        { }

        public Number(double val)
        {
            m_val = val;
        }

        public override string ToString()
        {
            return m_val.ToString();
        }

        public static Number operator ++(Number n)
        {
            Number result = new Number();

            result.m_val = n.m_val + 1;

            return result;
        }

        public static Number operator --(Number n)
```

```
        {
            Number result = new Number();

            result.m_val = n.m_val - 1;

            return result;
        }
    }
}
```

Tür Dönüştürme Operatör Metotlarının Yazımı

Bilindiği gibi tür dönüştürme işlemi otomatik (implicit) ya da tür dönüştürme operatörüyle (explicit) yapılabilmektedir. Tür dönüştürme operatör metotları tür dönüştürmesi yapılacağı zaman devreye girer. Dönüştürme yapılacağı zaman derleyici tarafından bu metotlar çağrılmaktadır. Tür dönüştürme operatör metotlarından geri döndürülen değerler dönüştürülmüş sonuç değer olarakelde elde edilmektedir. Tür dönüştürme operatör metotlarının geri dönüş değeri belirtilmez. Bunların isimleri operator anahtar sözcüğü ile tür belirten sözcükten oluşur. Bu tür belirten sözcük aynı azamanda metodun geri dönüş değeri gibi ele alınmaktadır. Tür dönüştürme operatör metotlarının genel biçimi şöyledir:

```
public static <implicit/explicit> operator <tür>(<parametre bildirimi>)
{
      //...
}
```

implicit belirleyicisi operatör metodunun hem otomatik dönüştürmelerde hem de tür dönüştürme operatörüyle yapılan dönüştürmelerde kullanılabileceğini belirtir. Halbuki explicit belirleyicisi metodun yalnızca tür dönüştürme sırasında devreye gireceğini belirtmektedir.

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
           Number n = new Number(12.4);
           double d;

           d = n;       // Burada operator metodunun geri dönüş değeri d'ye atanmaktadır
           Console.WriteLine(d);
        }
    }

    class Number
    {
        private double m_val;

        public Number()
        { }

        public Number(double val)
        {
            m_val = val;
        }

        public override string ToString()
        {
            return m_val.ToString();
        }

        public static implicit operator double(Number n)
        {
            return n.m_val;
```

```
        }
    }
}
```

Yukarıdaki örnekte operatör metodu explicit olsaydı biz onu yalnızca tür dönüştürme operatörüyle kullanabilirdik.

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
           Number n = new Number(12.4);
           double d;

           d = (double)n;       // Burada operator metodunun geri dönüş değeri d'ye atanmaktadır
           Console.WriteLine(d);
        }
    }

    class Number
    {
        private double m_val;

        public Number()
        { }

        public Number(double val)
        {
            m_val = val;
        }

        public override string ToString()
        {
            return m_val.ToString();
        }

        public static explicit operator double(Number n)
        {
            return n.m_val;
        }
    }
}
```

Bir sınıf ya da yapıda farklı türlere dönüştürme yapan birden fazla operatör metodu bulunabilir. Bu durumda implicit dönüştürme söz konusuysa hedef tür dikkate alınarak explicit dönüştürme söz konusuysa dönüştürülecek tür dikkate alınarak dönüştürme yapılır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Number n = new Number(12.4);
            int i;
            double d;

            i = n;
            Console.WriteLine(i);

            d = n;
            Console.WriteLine(d);
```

```
        }
    }

    class Number
    {
        private double m_val;

        public Number()
        { }

        public Number(double val)
        {
            m_val = val;
        }

        public override string ToString()
        {
            return m_val.ToString();
        }

        public static implicit operator int(Number n)
        {
            return (int)n.m_val;
        }

        public static implicit operator double(Number n)
        {
            return n.m_val;
        }
    }
}
```

Sınıfın ya da yapının aynı türe dönüştürme yapan birden fazla operatör metodu bulunamaz. (Bunlardan biri implicit diğeri explicit de olamaz.)

is Operatörü

is operatörü iki operandlı araek bir operatördür. Operatörün soldaki operandı bir sınıf türünden referans sağdaki operandı ise bir sınıf ya da yapı ismi olmak zorundadır. Bu operatör soldaki referansın dinamik türünün sağdaki türü kapsayıp kapsamadığına bakar. Eğer soldaki referansın dinamik türü sağdaki türü kapsıyorsa (yani o türden ya da ondan türetilmiş olan bir türdense) operatör true değilse false değeri üretir. Böylece elimizde taban sınıf türünden bir referans varsa onun asıl türünün belli bir sınıf türünden olup olmadığını anlayabiliriz ve down cast işlemi yapabiliriz. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            object o;
            string s = "ankara";

            o = s;

            if (o is string)
                Console.WriteLine("evet");
            else
                Console.WriteLine("hayır");
        }
    }
}
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            object[] objs = { 1.2, DateTime.Now, 123, "Ankara" };

            foreach (object o in objs)
            {
                if (o is double)
                {
                    double result = (double)o;
                    Console.WriteLine(result);
                }
                else if (o is DateTime)
                {
                    DateTime result = (DateTime)o;
                    Console.WriteLine(result);
                }
                else if (o is int)
                {
                    double result = (int)o;
                    Console.WriteLine(result);
                }
                else if (o is string)
                {
                    string result = (string)o;
                    Console.WriteLine(result);
                }
            }
        }
    }
}
```

Örneğin:

```
using System;

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

            try {
                Console.Write("Bir sayı giriniz:");
                n = int.Parse(Console.ReadLine());
            }
            catch (Exception e)
            {
                if (e is FormatException)
                    Console.WriteLine("Sayının formatı bozuk!");
                else if (e is OverflowException)
                    Console.WriteLine("Sayı çok büyük ya da çok küçük!");
                else
                    Console.WriteLine("null referans");
            }
        }
    }
}
```

as Operatörü

as operatörü is ile benzer amaçlarla kullanılır. as operatörü de iki operandlı araek bir operatördür. as operatörünün soldaki operandı bir sınıf türünden referans sağdaki operandı ise bir sınıf ya da yapı ismi olmak zorundadır. Bu operatör soldaki operandın dinamik türünün sağdaki türü içerip içermediğine bakar. Eğer içerme varsa downcast uygulayıp bize soldaki referansı sağdaki türe dönüştürerek verir. Eğer içerme yoksa as operatörü null referans vermektedir. as operatörünün ürettiği değer her zaman sağ taraftaki operandın türü türündendir. Örneğin:

```
object o;
string result;
//...
result = o as string;
if (result != null)
{
      //...
}
```

Başka bir deyişle:

```
if (o is string)
{
      string s = (string)o;
      //...
}
```

Bu işlemin as karşılığı şöyledir:

```
s = o as string;
if (s != null)
{
      //...
}
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            object o = "ankara";
            string result;

            result = o as string;

            if (result != null)
                Console.WriteLine(result);
        }
    }
}
```

İndeskleyiciler

İndeksleyiciler bir sınıf türünden referansın ya da bir yapı değişkeninin köşeli parantez operatörüyle kullanımına olanak sağlayan elemanlardır. Bunların yazılmaları property'lere benzemektedir. Genel biçimleri şöyledir:

```
[erişim belirleyicisi] <tür> this<[<parametre bildirimi>]>
{
      get
      {
            //...
      }
      set
      {
            //...
      }
}
```

Örneğin:

```
public int this[int index]
{
      get
      {
            //...
      }
      set
      {
            //...
      }
}
```

İndeksleyici köşeli parantez ile değerin elde edileceği ifadede kullanıldığında indeksleyicinin get bölümü, indeksleyici köeşeli parantez ile içersine değer yerleştirileceği bir ifadede kullanıldığında set bölümü çalıştırılır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s = new Sample();
            int val;

            val = s["123"];
            Console.WriteLine(val);
        }
    }

    class Sample
    {
        public int this[string s]
        {
            get
            {
                return int.Parse(s);
            }
        }
        //...
    }
}
```

İndeksleyiciler static olamaz. Sentakstaki this anahtar sözcüğü aynı biçimde bulundurulmak zorundadır.

İndeksleyiciler genel olarak dizisel (liste) biçiminde çalışan sınıflarda kullanılmaktadır. Örneğin string sınıfının, ArrayList sınıfının indeksleyicileri vardır. İndksleyicilerin de set bölümlerinde value anahtar sözcüğü kullanılabilir. Burada kullanılan value anahtar sözcüğü indeksleyiciye atanacak değeri belirtmektedir.

Örneğin :

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            MyIntArray mia = new MyIntArray(5);

            for (int i = 0; i < 5; ++i)
                mia[i] = i * i;

            for (int i = 0; i < 5; ++i)
                Console.Write("{0} ", mia[i]);
            Console.WriteLine();
        }
    }

    class MyIntArray
    {
        private int[] m_array;

        public MyIntArray(int n)
        {
            m_array = new int[n];
        }

        public int this[int index]
        {
            get
            {
                return m_array[index];
            }
            set
            {
                m_array[index] = value;
            }
        }
        //...
    }
}
```

İndeksleyiciler çok boyutlu olabilir. Bu durumda köşeli parantezin içerisine birden fazla parametre yerleştirilir. Örneğin:

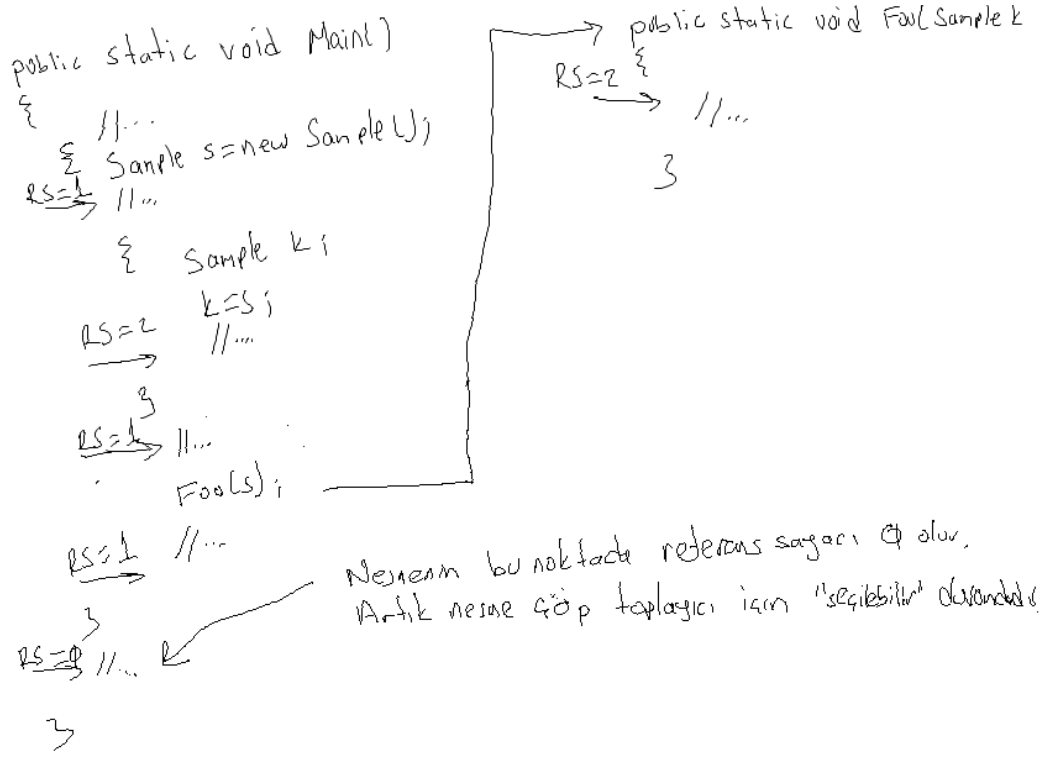
```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            MyIntArray mia = new MyIntArray(3, 3);

            mia[2, 1] = 100;
            Console.WriteLine(mia[2, 1]);
        }
    }

    class MyIntArray
```

```
    {
        private int[,] m_array;

        public MyIntArray(int n, int m)
        {
            m_array = new int[n, m];
        }

        public int this[int row, int col]
        {
            get
            {
                return m_array[row, col];
            }
            set
            {
                m_array[row, col] = value;
            }
        }
        //...
    }
}
```

Çöp Toplayıcı Mekanizma (Garbage Collection Mechanism)

Bilindiği gibi stack'te yer kaplayan yerel değişkenler ve parametre değişkenleri akış bloğa girdiğinde çok hızlı bir biçimde yaratılırlar ve akış bloktan çıktığında yine çok hızlı bit bir biçimde otomatik olarak yok edilirler. Pekiyi new ile heap'te tahsis edilen nesneler ne zaman yok edilmektedir? Bunları yok etmek için C++'ta delete isimli bir operatör vardır. Fakat C# ve Java'da ismine "çöp toplayıcı (garbage collector)" denilen mekanizma zaten çöp duruma geçmiş nesneleri heap'ten yok eder. Pekiyi çöp toplayıcı mekanizma nasıl çalışır?

CLR belli bir anda bir nesnenin kaç referans tarafından gösterildiğini arka planda izlemektedir. Buna nesnenin referans sayacı denir. Örneğin:



Örneğin nesnenin referans sayacı 3 ise o nesneyi 3 referans gösteriyor durumadır. Nesnenin referans sayacı sıfıra düştüğünde artık o nesneyi hiçbir referans göstermiyor durumdadır. Hiçbir referans tarafından gösterilmeyen nesnelere çöp toplayıcı için "seçilebilir (eligible)" durumda olan nesneler denilmektedir. Bir nesnenin çöp toplayıcı tarafından seçilebilir duruma gelmesi o nesnenin hemen çöp toplayıcı tarafından silineceği anlamına gelmez. Sistem kendisi için uygun bir zamanda çöp toplayıcıyı devreye sokarak diğer çöp

duruma gelmiş nesnenelerle birlikte onu da silecektir.. Nesnenin seçilebilir duruma geldikten ne kadar zaman sonra silineceği konusunda standart bir belirleme yapılmamıştır. (Örneğin biz çöpümüzü çöp konyetnerine atar atmaz mı çöpçü geliyor? Çöpçü gece gelip bizimkiyle beraber diğer çöpleri de topluyor)

Sonuç olarak biz C#'ta new ile nesneleri heap'te yaratırız. Bunların silinmesiyle biz uğraşmayız. Çöp toplayıcı mekanizma programlamayı oldukça kolaylaştırmaktadır.

Çöp toplayıcı programcı tarafından da göreve davet edilebilir. Bunun için GC sınıfının static Collect isimli çağrılır. Örneğin:

```
GC.Collect();
```

Sınıf Bildirimleri İçerisinde Veri Elemanlarına İlkdeğer Verilmesi

Normal olarak sınıfın veri elemanları new işlemi sırasında new operatörü tarafından sıfırlanmaktadır. Fakat başlangıç metotlarında bunlara ilkdeğerleri atanabilir. Ancak C#'ta alternatif olarak sınıfn veri elemanlarına sınıf bildirimi içeriisnde ilkdeğer verilebilmektedir. Örneğin:

```
class Sample
{
    private int m_a = 10;
    private int m_b = 20;
    private string m_str = "Ok";
    private Random m_rand = new Random();

    public Sample()
    {
        //...
    }
    //...
}
```

Sınıf bildirimi içerisinde veri elemanlarına ilkdeğer verildiğinde aslında derleyici bütün bu ilkdeğerleri atama deyimlerine dönüştürerek sınıfın bütün başlangıç metotlarının ana bloğunun başına gizlice aktarır. Yani yine biz sanki bunlara başlangıç metotları içerisinde değer atamış gibi oluruz. Yukarıdaki sınıfın eşdeğeri şöyledir:

```
class Sample
{
    private int m_a;
    private int m_b;
    private string m_str;
    private Random m_rand;

    public Sample()
    {
        m_a = 10;
        m_b = 20;
        m_str = "Ok";
        m_rand = new Random();
        //...
    }
    //...
}
```

Derleyici bildirimdeki sıraya göre atama deyimlerini oluşturmaktadır. Sınıfın hiç başlangıç metodu olmasa bile derleyici default başlangıç metodunu kendisi yazıp yine bu atamaları yapar. Örneğin:

```
using System;

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

```
        {
            MyRandom mr = new MyRandom();

            mr.Generate(10, 0, 10);
        }
    }

    class MyRandom
    {
        private Random m_rand = new Random();

        public void Generate(int n, int low, int high)
        {
            for (int i = 0; i < n; ++i)
                Console.Write("{0} ", m_rand.Next(low, high));
            Console.WriteLine();
        }
    }
}
```

Pekiyi sınıf bildirimi içerisinde değer atamanın faydası var mıdır? Eğer sınıfın çok sayıda başlangıç metodu varsa ve biz bu başlangıç metotlarının hepsinde veri elemanlarına hep aynı değeri veriyorsak bunu ilkdeğer biçiminde belirtmek yazım kolaylığı sağlayabilmektedir.

Derleyici ilkdeğer verme işlemlerini ana bloğun başına taşımaktadır. Ancak sınıfın bir taban sınıfı varsa, taban sınıf başlangıç metodu sınıfın elemanlarına ilkdeğer verildikten sonra çağrılır. Örneğin:

```
class Sample : Mample
{
    private int m_a = 10;

    public Sample() : base()
    {
        //...
    }
    //...
}
```

Burada başlangıç metodunda önce m_a'ya 10 atanacak sonra taban sınıfın başlangıç metodu çağrılacaktır.

Sınıfın static veri elemanlarına da bu biçimde ilkdeğer verilebilir. Örneğin:

```
class Sample
{
    private static int m_count = 1;
    //...
}
```

Bu biçimde static veri elemanlarına ilkdeğer verildiğinde bu ilkdeğerler sınıfın static başlangıç metotlarının ana bloğunun başına aktarılmaktadır. Yani bu işlemin eşdeğeri şöyledir:

```
class Sample
{
    private static int m_count;

    static Sample()
    {
        m_count = 1;
    }
    //...
}
```

static başlangıç metotları ileri ele alınmaktadır.

Sınıfların ve Yapıların const Veri Elemanları

Sınıfların ve yapıların const veri elemanları ilkdeğer verilerek bildirilmek zorundadır. İlkdeğer verildikten sonra const veri elemanlarının değerleri bir daha değiştirilemez. Örneğin:

```
class Sample
{
    public const int Max = 10;

    static Sample()
    {
        //...
    }
    //...
}
```

const belirleyicisi okunabilirliği artırmak için kullanılmaktadır. Sınıfların ve yapıların const veri elemanları default olarak aynı zamanda static durumdadır. Yani hem const hem de static belirleyicileri bir arada kullanılamaz. Program içerisinde değişmeyecek birtakım öğeler varsa onların const olarak belirtilmesi okunabilirliği artırmaktadır. const veri elemanlarına verilen ilkdeğerlerin sabit ifadesi olması gerekir. Örneğin:

```
class Sample
{
    public const Random Rand = new Random();        // error!

    static Sample()
    {
        //...
    }
    //...
}
```

new operatörü bir sabit ifadesi belirtmemektedir. O halde sınıfın bir yapı ya da sınıf türünden veri elemanları const olamaz.

const Yerel Değişkenler

Yerel değişkenler de const olabilir. Fakat C# programcıları bunu pek kullanmamaktadır. const yerel değişkenlere verilen ilkdeğerlerin de sabit ifadesi olması gerekir. Yine ilkdeğer atanmasından swonra onların içerisindeki değerler değiştirilemez. Örneğin:

```
class App
{
    public static void Main()
    {
        const double pi = 3.14159265;
        //..
    }
}
```

Sınıfların readonly Veri Elemanları

Bir sınıfın veri elemanı readonly olabilir. readonly veri eiemanlarına ilkdeğerleri sınıf bildiriminin içerisinde ya da sınıfın başlangıç metotlarında verilebilir. Bunlara başlangıç metodu dışında değer atanamaz. readonly veri elemanları static değildir. Fakat istenirse static de yapılabilir. Örneğin:

```
class Sample
{
    private readonly Random m_rand;

    public Sample()
    {
        m_rand = new Random();      // geçerli!
        //...
    }
```

```
    public void Foo()
    {
        m_rand = new Random();      // error!
        //..
    }
    //...
}
```

Sınıfların Bitiş Netotları (Destructors)

Nesne yok edilmeden az önce çöp toplayıcı sistem tarafından çağrılan sınıfın metoduna bitiş metodu (destructor) denilmektedir. Bitiş metotlarının isimleri sınıf ismiyle aynıdır ama başında bitişik olarak ~ karakteri bulunur. Yani bunların isimleri ~sınıf_ismi biçimindedir. Bitiş metotları erişim belirleyicilerine sahip olamaz. Tıpkı başlangıç metotlarında olduğu gibi bitiş metotlarının da geri dönüş değerleri diye bir kavramları yoktur. Bitiş metotları overload edilemez. Yani sınıfın tek bir bitiş metodu bulunabilir. Bitiş metotları parametresiz olmak zorundadır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s = new Sample();
            //...
        }
    }

    class Sample
    {
        public Sample()
        {
            Console.WriteLine("constructor");
        }

        ~Sample()
        {
            Console.WriteLine("destructor");
            //..
        }
        //...
    }
}
```

Bitiş metotlarına neden gereksinim duyulmaktadır? Bitiş metotları nesne tarafından tahsis edilmiş olan birtakım kaynakların nesne yok edilmeden önce geri bırakılması için kullanılmaktadır. Fakat maalesef .NET'te bitiş metotları deterministik olmadığı için bitiş metotlarının da kullanımı çok faydalı olamamaktadır. Bitiş metotlarının deterministik olmaması demek onların tam olarak ne zaman ve akış neredeyken çağrılacağının belli olmaması demektir. Tam olarak ne zaman yapılacağı belli olmayan boşaltım işlemleri pek çok durumda faydalı olmayabilmektedir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            {
                Sample s = new Sample(10);
                Sample k = new Sample(20);
                k = null;
```

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

            Console.WriteLine("Two");
        }
    }

    class Sample
    {
        private int m_a;

        public Sample(int a)
        {
            m_a = a;

            Console.WriteLine("Constructor: {0}", a);
        }

        ~Sample()
        {
            Console.WriteLine("Destructor: {0}", m_a);
        }
    }
}
```

Bu programdaki nesneler için bitiş metotlarının ne zaman çağrıldığına dikkat ediniz.

Yönetilmeyen (unmanaged) kaynakların boşaltılması bitiş metotlarında değil IDisposable arayüzünün Dispose metotlarında yapılmalıdır. Fakat yine de bu kaynakların en kötü olasıkla bitiş metotlarında da yok edilmesi uygun olur. Tabi silmenin hem Dispose metodunda hem de bitiş metodunda iki kez yapılması uygun olmaz. Bunun için GC sınıfının SupressFinalize metodundan faydalanılabilir. Bu metot verilen nesne için artık bitiş metodunun çağrılmamaısnı sağlar. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample s = new Sample();
            s.Dispose();
            //...
        }
    }

    class Sample : IDisposable
    {
        public Sample()
        {
            Console.WriteLine("Unmaged kaynak tahsis ediliyor");
        }

        public void Dispose()
        {
            Console.WriteLine("Unmaged kaynak boşaltılıyor");
            GC.SuppressFinalize(this);
        }

        ~Sample()
        {
            Dispose();
        }
        //...
    }
}
```

Özet olarak eğer biz bir unmanaged kaynak kullanıyorsak sınıfımızın IDisposable arayüzünü desteklemesi gerekir. Boşaltımı da Dispose metodunda yapmalıyız. Fakat eğer Dispose çağrılmazsa diye bitiş metodunda da biz Dispose metodunu ayrıca çağırmalıyız.

Yukarıda da sözünü ettiğimiz gibi .NET'teki bitiş metodu deterministik değildir. Bir sınıfın başka bir sınıf türünden veri elemanının bulunduğu durumda (composition) bitiş metodunda bu veri elemanını kullanamayız. Çünkü önce elemana sahip sınıfın bitiş metodunun çağrılması yönünde bir garanti yoktur. Örneğin:

```
class Sample
{
    private Mample m_mample;

    public Sample()
    {
        m_mample = new Mample();
    }

    ~Sample()
    {
        // m_mample burada kullanılamaz. Çünkü daha önce yok edilmiş olabilir
    }
}
```

Bu durum C#'ta bitiş metotlarının kullanım alanını çok daraltmaktadır. Halbuki örneğin C++'ta bitiş metotlarının tam olarak ne zaman ve hangi sırada çağrılacağı bellidir. Bu nedenle orada kritik işlemler bitiş metotlarında yapılabilmektedir.

Aslında .NET'te çöp toplayıcının çağırdığı metot bitiş metodu değil Finalize isimli bir metottur. Finalize metodu object sınıfının protected virtual bir metodudur. Biz C#'ta bitiş metodu yazdığımızda aslında derleyici object sınıfının Finalize metodunu override eder.

C#'ta yapılar bitiş metotlarına sahip olamazlar. Yalnızca sınıflar bitiş metotlarına sahip olabilmektedir.

Sınıfların static Başlangıç Metotları

C#'ta başlangıç metotları static de olabilmektedir. static başlangıç metotlarına erişim belirleyicisi yazılamaz. static başlangıç metotları overload edilemez. static başlangıç metotlarının parametresi olamaz. Başlangıç metodunu static yapabilmkek için onun başına static anahtar sözcüğü getirilmelidir. Örneğin:

```
class Sample
{
    public Sample()     // Normal başlangıç metodu (static olmayan = instance)
    {
        //...
    }

    static Sample()     // static başlangıç metodu
    {
        //...
    }
    //...
}
```

Sınıfın static başlangıç metotları toplamda yalnızca bir kez çalıştırılmaktadır. Bu nedenle sınıfın static veri elemanlarına ilkdeğer atamasında ya da toplamda yalnızca bir kez yapılması gerekken işlemlerde static başlangıç metotlarından faydalanılır. static başlangıç metotları şu durumda çalıştırılır:

- O sınıf türünden bir nesne ilk kez new operatörüyle yaratıldığında.

- O sınıfın static bir elemanı ilk kez kullanıldığında.

Örneğin:

```
using System;

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

            Console.WriteLine("One");

            s = new Sample();

            Console.WriteLine(Sample.A);
        }
    }

    class Sample
    {
        private static int m_a;

        public Sample()     // Normal başlangıç metodu (static olmayan = instance)
        {
            Console.WriteLine("Instance constructor");
        }

        static Sample()     // static başlangıç metodu
        {
            Console.WriteLine("static constructor");
            m_a = 100;
        }

        public static int A
        {
            get { return Sample.m_a; }
            set { Sample.m_a = value; }
        }
       //...
    }
}
```

Bir sınıf türünden bir referansın bildirilmesi static başlangıç metodunun çağrılmasını gerektirmemektedir. C#'ta yapılar da static başlangıç metotlarına sahip olabilirler.

İç İçe Tür Bildirimleri (Nested Type Declarations)

Bir sınıf ya da yapı içerisinde bir sınıf, yapı, enum ve delege bildirimleri yapılabilir. Bu durumda içte bildirilen türün başına sınıfın diğer elemanlarında olduğu gibi erişim belirleyicileri de getirilebilir. (Yine default durum private). Örneğin:

```
class Sample
{
    private int m_s;

    public void Foo()
    {
        Test t = new Test();        // geçerli
        t.Bar();                    // geçerli
    }

    private class Test
    {
        private int m_t;
```

```
        public void Bar()
        {
            //...
        }
    }
}
```

Bir sınıfn içerisinde bildirilen bir sınıf yine bağımsız bir sınıftır. Yani veri elemanları bakımından bir içerme ilişkisi yoktur. Dış sınıf iç sınıfın veri elemanlarını içermez. Buradaki iç bildirim mantıksal anlamda bir sınırlama oluşturmaktadır. Yani örneğimizdeki Test sınıf ismi yalnızca Sample içerisinde doğrudan kullanılabilir. Dışarıdan doğrudan kullanılamaz. Başka bir deyişle Test sınıfı yalnızca Sample sınıfı kullanılsın diye bildirilmiştir. Buradaki Test dışarıda başka kişileri ilgilendirecek bir sınıf değildir. Eğer içteki sınıf public ise bu sınıf dışarıdan kullanılabilir. Ancak dış sınıf ismi ile niteliklendirilmek zorundadır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample.Test test = new Sample.Test();       // geçerli
            //...
        }
    }

    class Sample
    {
        private int m_s;

        public void Foo()
        {
            Test t = new Test();        // geçerli
            t.Bar();                    // geçerli
        }

        public class Test
        {
            private int m_t;

            public void Bar()
            {
                //...
            }
        }
    }
}
```

Yukarıda da belirtildiği gibi iç içe sınıf bildirimlerinde bir data içermesi söz konusu değildir. İç ve dış sınıflar bağımsız sınıflar gibidir. Ancak iç sınıfın dış sınıf elemanlarına erişiminde bir ayrıcalığı vardır. İç sınıf dış sınıfın her bölümüne erişebilir. Tabi bu erişim ancak iç sınıfın içerisinde dii sınıf türünden bir referans ya da sınıf ismiyle yapılabilir. Ancak dış sınıfın iç sınıfa özel bir erişim avantajı yoktur. Dış sınıf yine bir referans ya da sınıf ismi yoluyla isç sınıfn yalnızca public bölümüne erişebilir. Örneğin:

```
class Sample
{
    private int m_s;

    //...
    private class Test
    {
        private int m_t;

        public void Bar()
        {
```

```
            Sample s = new Sample();
            s.m_s = 10;     // private ama geçerli
        }
    }
}
```

Bir sınıf ya da yapının içerisinde başka türler de bildirilebilir. Örneğin bir enum türü bildirilebilir:

```
class Sample
{
    private Fruit m_fruit;

    public Sample()
    {
        m_fruit = Fruit.Banana;
        //...
    }

    private enum Fruit
    {
        Apple, Banana, Strawberry
    }
    //...
}
```

Generic'ler

Generic'ler konusu C#'a Framework 2.0 ile (Visual Studio 2005) ile eklenmiştir. Generic'ler C++'ta "template" ismiyle zaten eskiden beri bulunmaktaydı. Java'da yoktu. Ancak C#'a eklendikten sonra Java da generic konusunu bünyesine katmıştır.

Generic'ler dile temelde iki amaç için sokulmuştur:

1) Bazen aynı işlemleri farklı türler için defalarca yapmak zorunda kalırız. Örneğin int türden bir dizinin en büyük elemanını bulan aşağıdaki gibi bir metot yazmak isteyelim:

```
public static int GetMax(int[] a)
{
    int max = a[0];

    for (int i = 1; i < a.Length; ++i)
        if (max < a[i])
            max = a[i];

    return max;
}
```

Bu metot yalnızca int diziler için çalışır. Biz long bir dizinin en büyük elemanını bulmak istersek içi aynı olan fakat parametresi ve geri dönüş değeri farklı olan yeni bir metot yazmak zorunda kalırız:

```
public static long GetMax(long[] a)
{
    long max = a[0];

    for (int i = 1; i < a.Length; ++i)
        if (max < a[i])
            max = a[i];

    return max;
}
```

İşte generic'ler sayesinde biz bir şablon metot yazarız. CLR'de o şablona bakarak bizim için uygun türden metodu kendisi oluşturur. Örneğin:

```
public static T GetMax<T>(T[] a) where T: IComparable
```

```
{
    T max = a[0];

    for (int i = 1; i < a.Length; ++i)
        if (max.CompareTo(a[i]) < 0)
            max = a[i];

    return max;
}
```

Burada T herhangi bir türü temsil etmektedir.

2) Generic'ler sayesinde object türüne dayalı collection'lardaki kutulama ve kutuyu açma dönüştürmelerinin oluşturduğu zaman kaybı engellenir. Generic öncesinde C#'taki collection sınıflar hep object temelinde çalışıyordu. Örneğin ArrayList her şeyi object olarak tutup bize geri vermektedir. Bu durum int, long, double gibi yapı türlerinin saklanması sırasında kutulama ve kutuyu açma dönüştürmelerini zorunlu hale getirmektedir. Bu da göreli bir zaman kaybı oluştur. Örneğin elimizde int'leri tutabilen bir IntArrayList sınıfı olsun. Bu sınıfın içerisindeki dizi int türden olacaktır. Sınıfın Add metodu da indeksleyicisi de int türden olur. Pekiyi int bilgileri saklamak IntArrayList sınıfı mı yoksa nomal ArrayList sınıfını mı daha hızlı çalışır? Tabi ki IntArrayList sınıfını. Çünkü biz int'leri bu sınıfta saklarken hiçbir kutulama ve kutuyu açma dönüştürmesine gerek kalmayacaktır. Pekiyi o zaman long türü için ayrı bir sınıf yazmak gerekmez mi? Ya da diğerleri için? İşte generic'ler sayesinde bir sınıf şablon olarak bir kez yazılır. Sonra ona bakılarak sınıflar üretilir.

Generic Sınıflar ve Yapılar

Generic sınıf ya da yapı bildiriminin genel biçimi şöyledir:

```
<class/struct> <isim><<generic tür parametre listesi>>
{
      //...
}
```

Örneğin:

```
class Sample<T, K>
{
    //...
}
```

Burada T ve K'ya generic tür parametreleri denmektedir. Generic tür parametreleri sınıfın ya da yapının bildirimi içerisinde ve metotlarının içerisinde tür belirten sözcük olarak kullanılabilir. Tür parametreleri genellikle Pascal tarzıyla harflendirilir ve genellikle T, K gibi tek bir harfler tercih edilmektedir. Örneğin:

```
class Sample<T>
{
    private T m_a;

    public Sample(T a)
    {
        m_a = a;
    }
    //...
}
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
```

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

            s.Val = 10;
            Console.WriteLine(s.Val);
        }
    }

    class Sample<T>
    {
        private T m_val;

        public T Val
        {
            get { return m_val; }
            set { m_val = value; }
        }
    }
}
```

Generic sınıflar ve yapılar birer şablon bildirimdir. CLR bu şablona bakar, generic tür parametreleri yerine gerçek türleri koyarak asıl sınıf ya da yapıyı oluşturur. Bir generic sınıf ya da yapı kullanılırken kesinlikle açısal parantezlerle generic tür parametrelerinin hangi türler olarak açılacağının belirtilmesi gerekir. Örneğin:

```
Sample<int> s = new Sample<int>(10);
```

Generic sınıf ve yapılar açısal parantez olmadan yani tür argümanları belirtilmeden kullanılamazlar. Yukarıdaki örnekte T türü int olarak belirlenmiştir.

Generic sınıf ve yapılar yalnızca birer şablon belirtir. Biz generic sınıf ya da yapı türünden bir nesne yarattığımızda CLR tür argümanlarını yerine koyarak gerçek sınıfı o şablona bakarak oluşturmaktadır.

C#'ta collection sınıfların generic versiyonları da vardır. Örneğin ArrayList sınıfının generic versiyonu List<T> sınıfıdır. Generic collection'lar System.Collections.Generic isim alanında bulunur. Örneğin:

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

namespace CSD
{
    class App
    {
        public static void Main()
        {
            List<int> numbers = new List<int>();

            for (int i = 0; i < 100; ++i)
                numbers.Add(i);

            foreach (int number in numbers)
                Console.Write("{0} ", number);
            Console.WriteLine();

            List<string> names = new List<string>();

            names.Add("Ali");
            names.Add("Veli");
            names.Add("Selami");
            names.Add("Ayşe");
            names.Add("Fatma");

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

```
}
```

Pekiyi biz referans türleri için de ArrayList yerine List<T> sınıfını kullansak yine de hız kazancı sağlanır mı? Yapılan testlerde referans türleri için ArrayList ya da List<T>'nin kullanılması arasında ciddi bir fark oluşmadığı görülmüştür. Çünkü ArrayList sınıfı da referanslar için bir boxing, unboxing dönüştürmesi uygulamamaktadır. Fakat yine de referans için List<T> kullanılması tür güvenliği bakımından tercih edilebilir. Şöyle ki: List<T> türünün Add gibi metotları bizden T türünden değer ister. Halbuki ArraYlist bizden object istediği için her girişi kabul edebilmektedir. Örneğin:

```
List<string> names = new List<string>();

names.Add("Ali");
names.Add(123);           // error!
```

Halbuki:

```
ArrayList names = new ArrayList();

names.Add("Ali");
names.Add(123);           // geçerli!
```

Fakat yine de biz gerçekten bazen collection içerisinde heterojen türleri tutmak isteyebiliriz. Bu durumda ArrayList kullanılabilir. Ancak genel olarak artık ArrayList yerine List<T> türünün kullanılması tercih edilmelidir.

Aynı generic sınıf ya da yapının farklı türlerle açılımı aynı türden değildir. Yani örneğin biz Sample<long> türünü Sample<int> türüne atayamayız. Örneğin:

```
Sample<long> s = new Sample<long>();
Sample<int> k = new Sample<int>();

s = k;            // error!
```

Generic bir sınıftan normal bir sınıf türetilebilir. Tabii bu durumda taban bildirimde açılım türü belirtilmelidir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            B b = new B();

            b.Val = 10;
            Console.WriteLine(b.Val);
        }
    }

    class A<T>
    {
        private T m_val;

        public T Val
        {
            get { return m_val; }
            set { m_val = value; }
        }
        //...
    }

    class B : A<int>
```

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

Burada B sınıfı A<int> sınıfından türetilmiştir. Generic bir sınıfın taban sınıfı normal bir sınıf olabilir. Bu durumda generic sınıfın her türden açılımı o sınıftan türetilmiş olur. Örneğin:

```
class A
{
    //...
}

class B<T> : A
{
    //...
}
```

Burada B'nin her türden açılımı A'dan türetilmiştir. Generic bir sınıf başka bir generic sınıftan türetilebilir. Örneğin:

```
class A<T>
{
    //...
}

class B<T> : A<T>
{
    //...
}
```

Burada B'nin T türünden açılımı A'nın T türünden açılımından türetilmiş durumdadır. (Örneğin B<int> sınıfı A<int> sınıfından türetilmiş durumdadır.)

Generic arayüzler söz konusu olabilir. Örneğin:

```
interface IX<T>
{
    void Foo(T a);
}
```

Burada IX<T> arayüzünü destekleyen sınıf ya da yapı Foo metodunu int parametresiyle bildirmek zorudandır. Örneğin:

```
interface IX<T>
{
    void Foo(T a);
}

class Sample : IX<int>
{
    public void Foo(int a)
    {
        //...
    }
    //...
}
```

Generic bir sınıf ya da yapı generic bir arayüzü destekleyebilir. Bu duurmda arayüz desteğinin sağlanması için metotların da generic parametrelerle yazılması gerekebilir. Örneğin:

```
interface IX<T>
{
    void Foo(T a);
}
```

```
class Sample<T> : IX<T>
{
    public void Foo(T a)
    {
        //...
    }
    //...
}
```

Generic Metotlar

Bir sınıfın tamamı değil yalnızca sınıfın belirli metotları generic olabilir. Metot bildiriminde metot isminden sonra açısal parantezler içerisinde generic parametreler belirtilirse bildirilen metot generic olur. Örneğin:

```
class Sample
{
    public void Foo<T>(T a)
    {
        //...
    }
    //...
}
```

Generic metotlar normal olarak açısal parantezler içerisinde generic tür parametrelerinin türü belirtilerek çağrılırlar. Örneğin:

```
using System;

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

            s.Foo<int>(10);
        }
    }

    class Sample
    {
        public void Foo<T>(T a)
        {
            Console.WriteLine(typeof(T).Name);
        }
        //...
    }
}
```

Burada biz T generic parametresini yalnızca Foo metodunda kullanabiliriz.

Derleyici belirli koşullar altında metodun argümanlarına bakarak generic parametrelerin türünü tahmin edebilir. Bu durumda biz metodu çağırırken açısal parantezleri kullanmak zorunda kalmayız. Örneğin:

```
using System;

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

```
            s.Foo(2.3);         // T double türünden
            s.Foo(100);         // T int türden
            s.Foo<double>(100); // T double türden
        }
    }

    class Sample
    {
        public void Foo<T>(T a)
        {
            Console.WriteLine(typeof(T).Name);
        }
        //...
    }
}
```

Eğer çağırma sırasına açısal parantezler kullanılmamışsa metodun parametre değişkeninin türü otomatik tespit edilmeye çalışılır. Eğer açısal parantezler kullanılmışsa her zaman açısal parantezler içerisindeki türe bakılmaktadır. Tabii otomatik tespitin yapılabilmesi için tüm tür parametrelerinin metot parametresinde kullanılmış olması gerekir. Örneğin aşağıdaki bir metot için otomatik belirleme yapılamaz. Bu metodun mutlaka açısal parantezlerle tür belirtilerek çağrılması gerekir:

```
public T Foo<T, K>(K a)
{
    //...
}
```

Burada biz metodu tek parametreyle çağırmak zorundayız. O zaman da T'nin türü bilinmeyecektir. O halde bu metot için derleyici otomatik tür tespiti yapamaz.

Generic Türlerde Tür Kısıtları (Type Constarints)

Normal olarak generic sınıf ve metotlarda generic tür herhangi bir olabilir. Böylece kod içerisinde derleyicinin her durumda geçerli sayabileceği ifadeler bulunmalıdır. Örneğin:

```
class Sample<T>
{
    private T m_a;

    public Sample()
    {
        m_a = null;     // error!
    }
    //...
}
```

Burada m_a = null ifadesi geçerli değildir. Çünkü T herhangi bir tür olarak açıldığında bu ifade geçerli olmalıdır; fakat değildir. İşte biz derleyiciye generic tür hakkında bazı kısıtlarda bulunursak derleyici bu kodu kabul edebilir. Örneğin biz derleyiciye generic parametrenin referans türlerine ilişkin olacağı garantisini verirsek derleycici yukarıdaki kodu kabul edecektir.

Üç tür generic kısıt vardır: "Birincil kısıtlar (primary constraints)", "ikincil kısıtlar (secondary constratints)" ve başlangıç "metodu kısıtı (constructor constratint)". Kısıtlar where anahtar sözcüğü ile aşağıdaki gibi belirtilir:

```
where <tür>:[birincil kısıtlar], [ikincil kısıtlar], [başlangıç metodu kısıtı]
```

Bu kısıtlar her tür için ayrıca bir where cümlesiyle yazılabilirler. Bu kısıtların yalnızca biri ya da birden fazlası olabilir. Ancak sıralama yukarıda belirtildiği biçimde yapılmalıdır.

Birincil kısıtlar class ve struct anahtar sözcüğünden ya da bir sınıf isminden oluşturulur. class anahtar sözcüğü ilgili tür parametresinin bir referans türünden olacağı anlamına gelir. struct ise ilgili türün değer türlerine ilişkin bir türden (yalnızca struct değil) olacağı anlamına gelir Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample<int, int> s;      // error!
            //...
        }
    }

    class Sample<T, K>
        where T:class
        where K: struct
    {
        private T m_a;
        private K m_b;

        public Sample()
        {
            m_a = null;     // geçerli
        }
        //...
    }
}
```

Burada artık biz Sample sınıfını kullanırken bu kısıtlara uymak zorundayız. Aksi halde derleme aşamasında error oluşur. Eğer birincil kısıt olarak bir sınıf ismi yazılırsa açım ya o sınıf türüyle ya da o sınıftan türetilmiş bir tür türüyle yapılmak zorundadır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample<Random> s = new Sample<Random>();      // geçerli
            //...
        }
    }

    class Sample<T>
        where T:Random
    {
        private T m_a;

        public Sample()
        {
           //...
        }

        public void Foo()
        {
            for (int i = 0; i < 10; ++i)
                Console.Write("{0} ", m_a.Next(100));       // geçerli
            Console.WriteLine();
        }
        //...
    }
}
```

İkincil kısıtlar bir grup arayüz isimlerinden oluşur. Bu durum ilgili türün o arayüzü desteklemek zorunda olduğu anlamına gelir. Yani biz açımı yaparken kullandığımız tür o arayüzü desteklemelidir. Örneğin:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Sample<FileStream> s = new Sample<FileStream>();      // geçerli
            //...
        }
    }

    class Sample<T>
        where T: class, IDisposable
    {
        private T m_a;

        public void Foo()
        {
            m_a.Dispose();      // geçerli
            //...
        }
        //...
    }
}
```

Burada T türü için şöyle bir kısıt oluşturduk: "T türü hem sınıf türü olarak açılmalı hem de IDisposable arayüzünü desteklemelidir."

Başlangıç metodu kısıtı ilgili türün default başlangıç metoduna sahip olduğu anlaına gelir. Böylece biz new operatörüyle o türden bir nesne yaratabiliriz. Başlangıç metodu kısıtı new() biçiminde belirtilir. Örneğin:

```
class Sample<T>
    where T: Random, new()
{
    private T m_a;

    public Sample()
    {
        m_a = new T();      // geçerli
        //...
    }

    public void Foo()
    {
        for (int i = 0; i < 10; ++i)
            Console.Write("{0} ", m_a.Next(100));       // geçerli
        Console.WriteLine();
    }
    //...
}
```

Burada artık T türü Random sınıfı türünden ya da bu sınıftan türetilmiş bir sınıf türünden olmak zorundadır ve ayrıca başlangıç metoduna da sahip olmak zorundadır.

Metotların ref ve out Parametreleri

Fonksiyon parametre bildirimlerinde parametreler için ref ya da out belirleyicileri kullanılabilir. Örneğin:

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

```
        //...
    }

    public static void Foo(ref int a)
    {
        //...
    }

    public static void Bar(out int a)
    {
        //...
    }
}
```

Bir parametre değişkeni yalnızca ref ya da yalnızca out ile bildirilebilir. Bildirimde hem ref hem de out parametreleri birlikte kullanılamaz. ref ve out parametreleri ilgili değişkenin adresinin metoda aktarılacağı anlamına gelir. Yani biz o parametre değişkenine atama yaptığımızda aslında ona geçirilen değişkene atama yapmış gibi oluruz. ref ya da out parametresine sahip bir metot çağrılırken artık argümanda da ref ya da out anahtar sözcüklerinin kullanılması gerekir. Örneğin:

```
using System;
using System.IO;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            int x = 10;

            Foo(ref x);

            Console.WriteLine(x);
        }

        public static void Foo(ref int a)
        {
            Console.WriteLine(a);

            a = 20;
        }
    }
}
```

ref ile out arasında basit bir fark vardır. Biz ref ile aktaracağımız parametre değişkenine başlangıçta değer vermek zorundayız. Dolayısıyla metotta buna değer atamak zorunda kalmayız.

```
int x;

Foo(ref x);     // error, x'e değer atanmış olmak zorunda
```

out parametresinde ise böyle bir zorunluluk yoktur. Fakat bu sefer metottan çıkana kadar parametre değişkenine değer atamak zorundayız. out bir parametre değişkeni metot içerisinde henüz değer atamadan kullanılamaz. Özetle ref ile out arasındaki farklılıklar şunlardır:

1) ref ile aktarım sırasında argüman olarak kullanılacak değişkenine değer verilmek zorundadır fakat out ile aktarımda böyle bir zorunluluk yoktur. Örneğin:

```
int x;

Foo(ref x);           // error!
Bar(out x);           // geçerli
```

Tabi argüman değişkenine değer atanmışsa biz yine onu istersek out ile geçirebiliriz.

2) ref parametre değişkenini metot içerisinde kullanabiliriz. Kullanmadan önce ona değer atamak zorunda değiliz. Halbuki out parametre değişkenini kullanmadan önce ona değer atamak zorundayız. Ayrıca out parametre değişkene metot içerisinde kullanılsın ya da kullanılmasın metottan çıkmadan değer atanması zorunludur.

Her ne kadar ref ve out genellikle değer türleriyle kullanılıyor olsa da aslında referans türleriyle de kullanılabilir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Random r;

            Foo(out r);
            Console.WriteLine(r.Next(10));
        }

        public static void Foo(out Random a)
        {
            a = new Random();
        }
    }
}
```

Burada biz a = new Random() ifadesinde aslında r'ye atama yapmış olduk. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            string str = "Ankara";
            Foo(ref str);
            Console.WriteLine(str);
        }

        public static void Foo(ref string s)
        {
            Console.WriteLine(s);
            s = "Eskişehir";
        }
    }
}
```

Metotların params Dizi Parametreleri

Metotların dizi parametreleri params belirleyicisi alabilir. Örneğin:

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

params dizi parametreli metotlar normal olarak dizi referansıyla çağrılabilirler. Örneğin:

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

Foo(a);
```

Fakat istenirse bunlar ayrıca sanki tek tek argüman veriliyormuş gibi de çağrılabilirler. Örneğin:

```
Foo(1);
Foo(1, 2, 3, 4, 5);
Foo();      // geçerli, sıfır elemanlı dizi olabilir
```

Aslında derleyici böyle bir çağrıda argümanları bir diziye toplayıp yine diziyi argüman olarak geçirmektedir. Yani örneğin:

```
Foo(1, 2, 3, 4, 5);
```

çağrısı için derleyici arka planda aslında:

```
Foo(new int[]{1, 2, 3, 4, 5});
```

işlemini yapar.

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Foo();
            Foo(1);
            Foo(1, 2);
            Foo(1, 2, 3);
            Foo(1, 2, 3, 4);
        }

        public static void Foo(params int[] a)
        {
            int total = 0;

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

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

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Foo("ali", 123, DateTime.Today);
        }

        public static void Foo(params object[] objs)
        {
```

```
            foreach (object o in objs)
                Console.WriteLine(o.ToString());
        }
    }
}
```

params dizi parametresinin yanı sıra metodun başka parametreleri de olabilir. Bu durumda params dizi parametresi her zaman sonda olmak zorundadır. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Foo();              // error!
            Foo(1);             // Foo(1, new int[]{})
            Foo(1, 2, 3, 4);    // Foo(1, new int[] {2, 3, 4});
        }

        public static void Foo(int x, params int[] a)
        {
            //...
        }
    }
}
```

Türlere İlişkin Type Nesneleri

Doğrudan isim alanlarının içerisinde bildirilen öğelere tür (type) denilmektedir. C#'ta 5 tür kategorisi vardır: class, struct, interface, enum ve delegate. CLR kullanılan her tür için toplamda bir tane System.Type isimli sınıf türünden nesne oluşturur. Biz bir türün Type nesne referansını typeof isimli operatör ile elde edebiliriz. typeof operatörünün genel biçimi şöyledir:

```
typeof(<tür ismi>)
```

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Type t;

            t = typeof(Sample);
            Console.WriteLine(t.Name);

            t = typeof(Color);
            Console.WriteLine(t.Name);

            t = typeof(string);
            Console.WriteLine(t.Name);
        }
    }

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

```
    enum Color
    {
        Red, Green, Blue
    }
}
```

Her tür için toplamda tek bir Type nesnesi oluşturulmaktadır. Dolayısıyla biz aynı tür için birden fazla kez typeof operatörünü kullanıyor olsak bile bu operatör bize hep aynı nesnenin referansını verir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            Type t1, t2;

            t1 = typeof(Sample);
            t2 = typeof(Sample);

            Console.WriteLine(object.ReferenceEquals(t1, t2));  // True
        }
    }

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

object sınıfının GetType isimli metodu ilgili referansın dinamik türüne ilişkin Type nesne referansına bize verir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            string s = "ankara";
            object o;
            Type t;

            o = s;
            t = o.GetType();
            Console.WriteLine(t.Name);      // String
        }
    }

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

Örneğin:

```
using System;

namespace CSD
{
    class App
    {
```

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

            try
            {

                Console.Write("Bir değer giriniz:");
                val = int.Parse(Console.ReadLine());
                Console.WriteLine(val * val);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.GetType().Name);
            }
        }
    }

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

Pekiyi Type sınıfı neden kullanılmaktadır. İşte bazı durumlarda bazı metotlar bizden bir türü almak için onun Type nesne referansını isterler. Biz de onu typeof ya da GetType metoduyla elde ederek veririz. Örneğin Enum sınıfının static GetNames metodunun parametrik yapısı şöyledir:

```
public static string[] GetNames(Type enumType)
```

Bu metot bizden enum türünün Type nesne referansını ister bize o enumdaki tüm sabitlerini yazı olarak verir. Örneğin:

```
using System;

namespace CSD
{
    class App
    {
        public static void Main()
        {
            foreach (string str in Enum.GetNames(typeof(Fruit)))
                Console.WriteLine(str);
        }
    }

    enum Fruit
    {
        Apple, Babana, Cherry, Apricot
    }
}
```