

Her Yönüyle ilk C# Programımız

C# dili ortaya çıkalı daha hiç birşey yapmayan varsa ya da birşeyler yapıp da ne yaptığından emin olmayan varsa işte bu yazı tam size göre. Bu yazımızda klasik Merhaba Dünya programımızı yazacağız.Ama programımızı yazarken her şeyi adım adım öğreneceğiz. Unutmayın amacımız burada Merhaba Dünya yazmak değil. :) O halde aşağıdaki programı iyice inceleyin önce, şöyle bir süzün programı yukarıdan aşağıya, fazla detaylara inmeden yazımızı okumaya devam edin;

```
//dosya adı : Merhaba.cs
```

```
using System;
```

```
namespace MerhabaDunya
```

```
{  
    class Sınıf1  
    {  
        static void Main(string args[])  
        {  
            Console.WriteLine("Merhaba Dünya");  
        }  
    }  
}
```

Yukarıdaki ilk programımızı incelediğimize göre açıklamalarımıza geçebiliriz. Eğer önceden C++ ve Java ile ilgilenmiş arkadaşlar varsa yukarıdaki kodlar tanıdık gelebilir. Nitekim, her ne kadar Microsoft firması ilk başlarda bunu kabul etmese de C# dili Java ve C++ dillerinin harmanlanmasından oluşmuş bir dildir. Bugün bunu kabul etmeyen yoktur sanırım.

Yukarıdaki ilk programımızın kodunu ben Notepad ile yazdım. Ama kodu derleyip çalıştırmak için bir C# derleyicisine ihtiyacımız olacak. C# derleyicisi Visual Studio.NET ile kurulabileceği gibi www.microsoft.com web sitesinden .NET Framework yazılımını indirerek de kurulabilir. Eğer Visual Studio ortamında çalışıyorsanız yukarıdaki kodları Visual Studio .NET ' in sunduğu hazır proje şablonlarından rahatlıkla oluşturabilirsiniz. Visual Studio programını çalıştırdıktan sonra Project->New menüsünden dil olarak Visual C# ve proje şablonu olarak da "Console Application" seçerseniz, main işlevi içindeki kodlar dışındaki yapı otomatikmen oluşturulacaktır.Eğer .NET Framework yapısını kurduysanız Console Ekranından C# derleyicisini çalıştırmalısınız. Komut ekranını <csc Merhaba.cs> yazarak kaynak kodumuzu derleyebilirsiniz.

Şimdi kodlarımızı inceleyelim. İlk satırdaki <using System;> ifadesi System adlı bir isim alanının kullanılacağını belirtiyor.Peki nedir bu isim alanı(Namespace). İsim alanı kavramı son yıllarda program modüllerinin çok sayıda artmasından dolayı popüler hale gelmiştir.

Kolay ve hızlı programlama yapmamızı sağlayan bir takım hazır kütüphaneler her ne kadar işimizi kolaylaştırırsa da eğer isim alanları olmasaydı kullanacağımız her kütüphane bizim için işin içinden çıkılmaz bir hale gelebilirdi. Düşünün ki iki ayrı firma iki ayrı sınıf kütüphaneleri oluşturdu ve bu kütüphanelerin içinde aynı isimli birden çok sınıf yapısı var. Eğer biz programcı olarak iki firmanın da kütüphanesini kullanmak istiyorsak her ikisini aynı kod içinde kullanamayız. Çünkü aynı isimli sınıflar derleme aşamasında hata verecektir. Bu durumda yapılması gereken tek şey ya da en etkili yöntem isim alanlarını kullanmaktır. Yani bir sınıfa(class) ulaşabilmek için onun isim alanıyla çağırmak. İsim alanları hiyerarşik yapıda olabilir. Mesela System isim alanının altında başka bir isim alanı onun altında başkaları vs. İşte .NET isim alanı(namespace) hiyerarşisinin en tepesinde bulunan isim alanı System adlı isim alanıdır. En temel işlemlerimiz için bile bu isim alanını kullanmalıyız. Aksi halde programımız çalışmayacaktır. İsim alanlarını kullanmak için isim alanının başına using sözcüğü getirilir.

Soru: System isim alanının içinde Data isim alanında bulunan bir cs adlı sınıfı kullanabilmek için kaynak kodumuza ne eklememiz gerekir.

Cevap : Kaynak kodumuzun en başına aşağıdaki ifadeyi yazmamız gerekir.

`using System.Data;`

Bildiğiniz gibi C# dili %100 nesne tabanlı bir dildir. Yaptığımız herşey bir sınıf nesnesidir C# dilinde. Nesne olmayan hiçbirşey yoktur. C++ dilindeki main işlevini hatırlarsınız çoğunuz. Programımız c++ dilinde main işlevinden başlar ama main işlevi hiç bir zaman bir sınıf içinde olmamıştır.C# dilinde herşey sınıflarla temsil edildiği için main işlevi de bizim belirlediğimiz bir sınıfın işlevi olmak zorundadır. Yukarıdaki programımızda `<class Sınıf1>` ifadesi ile programımızda bir sınıf nesnesi oluşturuyoruz. Sınıf1 sınıfının bir işlevi olan main'in elbette eskiden de olduğu gibi özel bir anlamı vardır. Biliyorsunuz ki derleyiciler programın nerden çalışacağını bilmek isterler, aksi halde derleme işleminden sonra "programınız için başlama noktası bulunamadı" hatası alırız. Bu yüzden main işlevi bizim için eskiden de olduğu gibi programımızın başlangıç noktasıdır. Yani biz programda yapmak istediklerimizi main işlevi içinde gerçekleştireceğiz. Sınıf tanımlamalarımızı ise istediğimiz noktada yapabiliriz. Daha öncede dediğimiz gibi isim alanları birçok sınıfın veya tek bir sınıfın oluşturduğu kümedir. Bizim ana programımız da bir sınıf olduğuna göre Class1 sınıfını istediğimiz isimli bir isim alanına sokabiliriz. Yukarıda `<namespace MerhabaDunya>` yazarak isim alanını başlatıyoruz.

Şimdi main işlevinin içine bakalım, System isim alanında bulunan Console sınıfının bir metodu olan WriteLine() ile ekrana bir string ifadesi yazdırıyoruz. Biz burda iki tırnak ifadesi içinde yazımızı belirtmemize rağmen fonksiyonun kullanımı bununla sınırlı değildir. C# dilindeki fonksiyon aşırı yükleme (function overloading)kullanılarak fonksiyonu birçok parametrik yapıda kullanabilmemiz sağlanmıştır. Fonksiyon aşırı yükleme konusuna bundan sonraki yazılarımızda değineceğimizi belirtelim. WriteLine() işlevinin adından da anlaşılacağı gibi ekrana basmak istediğimiz yazıdan sonra satır atlama işlemi yapar.Bunu test etmek için bir tane "Merhaba Dünya" da siz yazdırın. Göreceksiniz ki siz belirtmemenize rağmen alt alta iki tane "Merhaba Dünya" yazısı çıkacak.

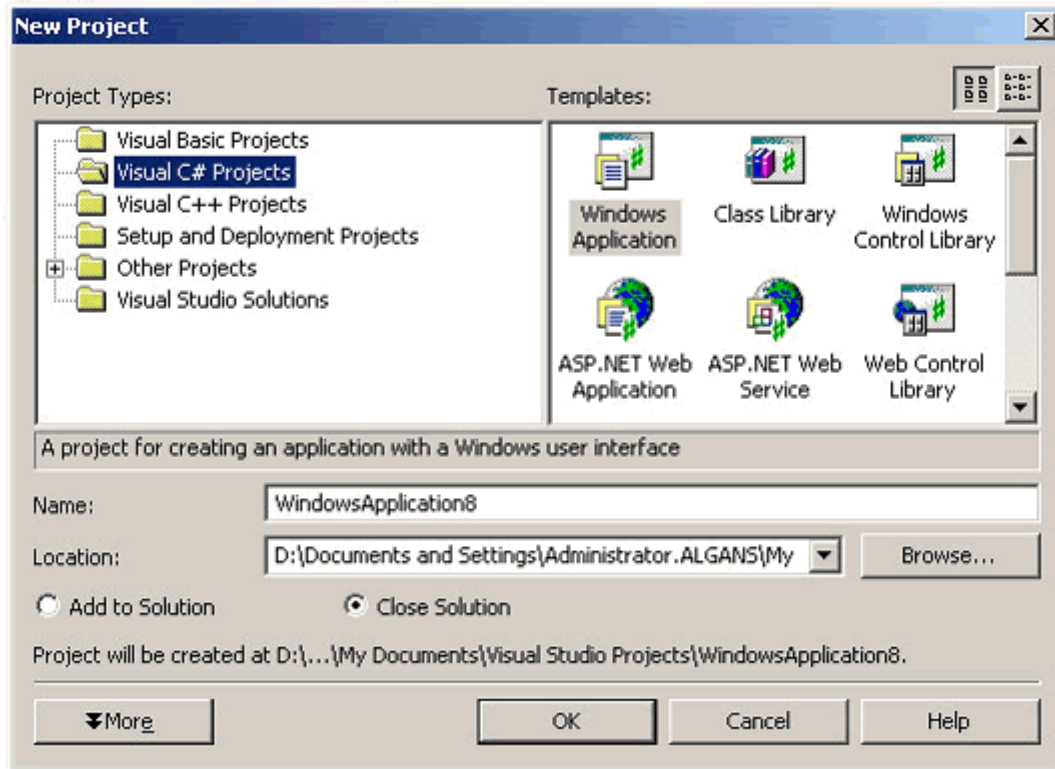
Eğer bu programı yazıp derlediyeseniz ne mutlu size ki C# dünyasına güzel bir adım attınız.

Visual C# ile Windows Menüleri Hazırlama

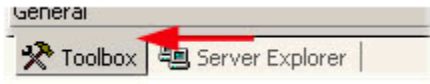
Merhaba, bu makalemizde, hemen hemen tüm Windows uygulamalarının temel yapı taşı olan Windows menülerinin nasıl hazırlandığını ve basit bir uygulamasını adım adım göreceğiz. Bildiğiniz gibi Windows menülerini şimdiye kadar Visual Basic ortamında çok basit bir şekilde yapmak mümkündür. Ama artık Visual C# ile menü hazırlamak hem daha kolay hem de daha eğlenceli. Bu makalede yapacağımız uygulamadaki amacımız, File ve Edit bölümünden oluşan Windows menüsünü tek bir Windows butonuyla aktif ya da pasif duruma getirmek.

Şimdi uygulamamızın ilk adımı olan yeni proje oluşturma sayfasını açalım.

File->New -> Project menüsünü kullanarak aşağıdaki gibi yeni bir proje oluşturalım. Proje tipi olarak Visual C# Project, template olarak da Windows Application seçtikten sonra projemize uygun isim verip OK butonuna tıklayalım.



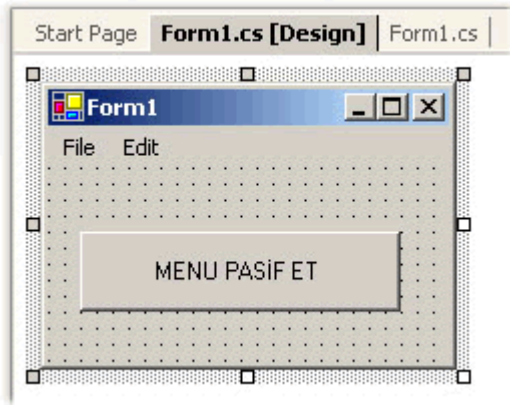
Projemizi oluşturduğumuzda Visual C# IDE 'sinin bizim için bir başlangıç formu oluşturduğunu görürüz. Bu form doğal olarak şu anda boştur. Toolbox menüsünü kullanarak Form üzerine istediğimiz kontrolleri sürükleyip bırak yöntemiyle yerleştirebiliriz. Ya da istediğimiz kontrolü çift tıklayarak da aynı işlevi gerçekleştirebiliriz. Eğer toolbox menüsünü göremiyorsanız ekranın sol alt köşesinde bulunan



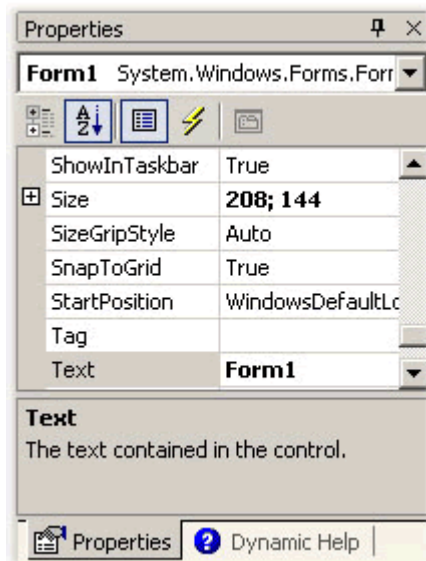
ToolBox ikonuna tıklayın. Şimdi formumuza basit bir MainMenu kontrolü ekleyelim. Yine ToolBox menüsünden aşağıdaki ikona şekline benzeyen kısma çift tıklayın. Eğer işlem başarılıysa formunuzun en üst kısmında edit edilmek üzere bir menü oluşacaktır.



MainMenu üzerine fare ile gelerek istediğiniz menü elemanlarını ekleyin. Ben önce File elemanını sonra Edit elemanını ve Edit elemanının içinde de Copy ve Paste menü elemanlarını aşağıdaki gibi oluşturdum.



Şimdi menü elemanlarımıza Properties penceresinden isim verelim. Aşağıda gördüğünüz pencereden form design penceresinden seçtiğiniz elemanla ilgili özelliklere ulaşabilirsiniz. Şimdi Edit menü elemanına tıklayarak Properties ekranındaki name özelliğine "menuEdit" yazalım. Burda menu elemanına verdiğimiz ismi daha sonra kod yazarken kullanacağımız için aklımızda kalacak bir isim vermemiz düzenli kod yazmak için önemli bir sebeptir. Menü elemanlarıyla işimiz bittiğine göre sıra menüyü kontrol edeceğimiz butonu yerleştirmeye geldi. ToolBox penceresinden "Buton" a çift tıklayarak forma bir buton yerleştirelim. Daha sonra butona tıklayıp Properties penceresinden buton ismi (Name) olarak "BizimButon" yazalım. BizimButon'un text özelliğine ise "MENU PASİF ET" yazısını yazalım. Bu yazıyı yazmamızın sebebi ise şudur: Menü elemanları varsayılan olarak aktif durumdadırlar. Bu yüzden menüyü pasif hale getirmek için bu yazıyı seçtik.



Evet, Form tasarım işlemi bitti. Şimdi sıra geldi BizimButon ile menüye aktif ve pasif durumları arasında geçiş yaptırmak. Tabi asıl işte şimdi başlıyor. Form üzerindeki butona çift tıklayarak kod yazma ekranına gelelim.

Gördüğünüz gibi Visual C# bizim için bir takım kodlar oluşturdu. Biraz bu hazır kodları ana hatlarıyla inceleyelim.

```
using System;  
using System.Drawing;  
using System.Collections;  
using System.ComponentModel;  
using System.Windows.Forms;  
using System.Data;
```

Yukarıdaki kodlarla programımızın kullanacağı bir takım sistemler derleyiciye bildiriliyor.

```
public class Form1 : System.Windows.Forms.Form
```

System.Windows.Forms.Form sınıfından yeni bir Form1(bizim form) sınıfı türetilerek bu form içindeki elemanlar tanımlanıyor.

```
private System.Windows.Forms.MainMenu mainMenu1;  
private System.Windows.Forms.MenuItem menuFile;  
private System.Windows.Forms.MenuItem menuEdit;  
private System.Windows.Forms.MenuItem menuItem3;  
private System.Windows.Forms.MenuItem menuItem4;  
private System.Windows.Forms.Button BizimButon;  
private System.ComponentModel.Container components = null;
```

private void InitializeComponent() işlevi ile Form1 sınıfı içindeki elemanlarla ilgili ilk işlemler yapılıyor. Elemanların form üzerindeki yeri ve elemanlara ait Properties penceresinden tanımladığımız bütün özellikleri bu işlev ile yerine getirilir.

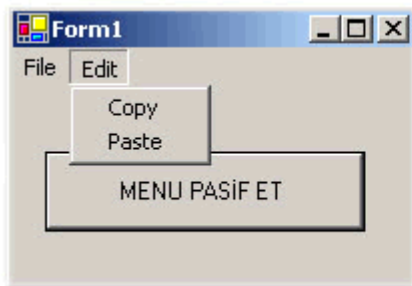
```
static void Main()  
{  
Application.Run(new Form1());  
}
```

Uygulamamızın Form1 üzerinden gerçekleştirileceğini belirtir.

İşte bu da bizim kodumuz :

```
private void BizimButon_Click(object sender, System.EventArgs e)
{
    if (menuEdit.Enabled)
    {
        menuEdit.Enabled=false;
        BizimButon.Text="MENU AKTIF ET";
    }
    else
    {
        menuEdit.Enabled=true;
        BizimButon.Text="MENU PASIF ET";
    }
}
```

Bu kodu yazabilmek için form design penceresinden BizimButon çift tıklayarak BizimButon_click() işlevinin içine geçelim. Yukarıdaki kodda eğer menuEdit aktifse pasif duruma getiriyoruz ve BizimButon 'a da "MENU AKTIF ET" yazıyoruz. Eğer menuEdit zaten pasifse Menuyu aktif hale getirip BizimButon yazısını da "MENU PASIF ET" yapıyoruz. Aşağıda her iki durum için programımızın çıktısı mevcuttur.



C ve C++ bakış açısıyla C# dili

Bildiğimiz gibi bilgisayarları programlamak için programlama dillerine ihtiyaç duyulur. Bu dillerden en popülerleri Basic, C, C++, Pascal, Java ve Assembler 'dır. Makina dili ise donanımı kontrol etmek için donanımı üreten firma tarafından tanımlanan komutlar kümesidir. Bazı programlama dilleri derleyicilere ihtiyaç duymasına karşın bazıları ise yorumlayıcılara ihtiyaç duyarlar, mesela bir c++ programını çalıştırabilmek için C++ derleyicisine ihtiyacımız varken, Perl ile yazılmış bir CGI scripti için komut yorumlayıcısına ihtiyacımız vardır. Derleyiciler programı çalıştırmadan önce kodları makina komutlarına çevirirler fakat yorumlayıcılar bir grup kodu satır satır ya da bloklar halinde yorumlayarak çalıştırırlar.

Aslında derleyiciler de, komut yorumlayıcıları da birer bilgisayar programından başka birşey değildirler. Yani c ve c++ dilleri bir giriş bekleyen ve çıkış veren birer bilgisayar programları gibi düşünülebilir. Giriş olarak kaynak kodu veren bu programlar çıkış olarak ise makina kodu üretirler.

C ve C++ dillerine kısa bir bakış:

C dili en popüler yapısal programlama dilidir. C dili Dennis Ritchie tarafından, Martin Richards ve Ken Thompson tarafından geliştirilen BCBL ve B dillerinin temelleri üzerine kuruldu.

C dili "The C Programming Language by Brian Kernighan and Dennis Ritchie" kitabıyla büyümüştür. C dili için, 1983 yılının büyük önemi vardır. Çünkü 1983 yılında ANSI standartlar komitesi C standartları için toplanmıştır. Bu standartlaşma süreci tam 6 yıl sürmüştür. Ve tabi ki şu anki standartların oluşumuna katkıda bulunan ANSI 99 standartları da diğer önemli bir gelişmedir.

C programcılar tarafından herhangi bir tür program geliştirmek için yazılmış genel amaçlı bir dildir. C ile bir düşük seviyeli sistem için program yazabileceğimiz gibi, yüksek seviyeli bir GUI (Grafik Arabirimi) tasarlamamız da mümkündür. Ve elbette kendi kütüphanemizi de C ile oluşturabiliriz. C dilinin ortaya çıkmasından bunca yıl geçmesine rağmen popüleritesini hiçbir zaman kaybetmemiştir. Günümüz programcılar çeşitli amaçlar için programlarını geliştirirken C dili ile yazılmış kaynak kodlarını kullanırlar.

Bjarne Stroustrup 1980 yılında C++ dilini ortaya çıkarmıştır. C++ dili C temelli ve C nin bir üst kümesi olarak düşünülebilir. C++ en popüler nesne temelli programlama dilidir. C++ dilinin ilk ismi "C with Classes" (C ile sınıflar) idi. C++ dili C diline nazaran daha etkili

ve güçlüdür. Ve en önemli özelliği ise C 'den farklı olarak nesne temelli bir dildir. Şu anda C++ dili ANSI ve ISO kuruluşları tarafından standartlaştırılmıştır. Bu standartların son versiyonu 1997 yılında yayınlanmıştır.

C# diline kısa bir bakış:

C#, güçlü, modern, nesne tabanlı ve aynı zaman type-safe(tip-güvenli) bir programlama dilidir. Aynı zamanda C#, C++ dilinin güçlülüğünü ve Visual Basic' in ise kolaylığını sağlar. Büyük olasılıkla C# dilinin çıkması Java dilinin çıkmasından bu yana programcılık adına yapılan en büyük gelişmedir. C#, C++ 'ın gücünden , Visual Basic 'in kolaylığından ve Java 'nın da özelliklerinden faydalanarak tasarlanmış bir dildir. Fakat şunu da söylemeliyiz ki, Delphi ve C++ Builder 'daki bazı özellikler şimdi C# 'da var. Ama Delphi ya da C++ Builder hiçbir zaman Visual C++ ya da Visual Basic 'in popülaritesini yakalayamamıştır.

C ve C++ programcıları için en büyük sorun, sanırım hızlı geliştirememedir. Çünkü C ve C++ programcıları çok alt seviye ile ilgilenirler. Üst seviyeye çıkmak istediklerinde ise zorlanırlar. Ama C# ile artık böyle bir dert kalmadı. Aynı ortamda ister alt seviyede isterseniz de yüksek seviyede program geliştirebilirsiniz. C# dili Microsoft tarafından geliştirilen .NET platformunun en temel ve resmi dili olarak lanse edilmiştir. C# dili Turbo Pascal derleyicisini ve Delphi 'yi oluşturan takımın lideri olan Anders Heljsberg ve Microsoft'da Visual J++ takımında çalışan Scott Wiltamuth tarafından geliştirilmiştir.

.NET framework'ünde bulunan CLR (Common Language Runtime), JVM (Java Virtual Machine)' ye, garbage collection, güvenilirlik ve JIT (Just in Time Compilation) bakımından çok benzer.

CLR, .NET Framework yapısının servis sağlama ve çalışma zamanının kod organizasyonu yapan ortamıdır. CLR, ECMA standartlarını destekler.

Kısacası C# kullanmak için CLR ve .NET Framework sınıf kütüphanesine ihtiyacımız vardır. Bu da demek oluyor ki C#, JAVA, VB ya da C++ değildir. C, C++ ve JAVA 'nın güzel özelliklerini barındıran yeni bir programlama dilidir. Sonuç olarak C# ile kod yazmak hem daha avantajlı hem daha kolay hem de etkileyicidir.

Bir başka yazıda buluşmak ümidiyle ...

C#'da İfadeler, Tipler ve Değişkenler

Bu derste C# dilindeki ifadeler, tipler ve değişkenler anlatılacaktır.

Dersimizin hedefleri :

- Değişken kavramının anlaşılması.
- C# dilinde bulunan basit tiplerin öğrenilmesi.
- C# dilindeki ifadelerin temel olarak anlaşılması.
- String veri tipinin öğrenilmesi.

Değişkenleri en sabit şekilde verilerin depolandığı yerler olarak tanımlayabiliriz. Değişkenlerin içine verilerimizi koyabiliriz veya değişkenlerimizin içindeki verileri C# programındaki işlemlerimiz için kullanabiliriz. Değişkenlerin tipini belirleyen faktör, onların içerdikleri verilerin çeşitleridir.

C# dilinde kullanacağımız her değişkenin bir tipi olmak zorundadır (Vbscript ve JavaScript gibi dillerde değişken tanımlarken onun tipini de ayrıca belirtmeye gerek yoktur.) Bir değişken üzerinde yapılan tüm işlemler de onun hangi tipte bir değişken olduğu göz önüne alınarak yapılır. Böylece programda bütünlük ve güvenlik korunmuş olur.

Boolean (doğru/yanlış), ve üç sayısal veri tipi; integer(tamsayı), floating point (ondalıklı sayı) ve decimal(muhasebe ve finansal işlemler için) C# dilinin en basit veri tipleri olarak sayılabilir.

Kod 1 : Boolean değerlerin görüntülenmesi : Boolean.cs

```
using System;

class Booleans

{

    public static void Main()
```

```
{  
  
    bool content = true;  
  
    bool noContent = false;  
  
    Console.WriteLine("It is {0} that C# Station provides C# programming language  
content.", content);  
  
    Console.WriteLine("The statement above is not {0}.", noContent);  
  
}  
  
}
```

Yukarıdaki Kod 1’de de görüleceği gibi boolean değişkenler ya true(doğru) veya false(yanlış) değerlerini alabilirler. Programın çıktısı şöyle olacaktır.

>It is True that C# Station provides C# programming language content.
>The statement above is not False.

Aşağıdaki tablo tamsayı tiplerini, boyutlarını ve alabilecekleri değer aralıklarını göstermektedir.

Type (Tip)	Size (in bits) (boyut)	Range (aralık)
sbyte	8	-128 to 127
byte	87	0 to 255
short	16	-32768 to 32767
ushort	16	0 to 65535
int	32	-2147483648 to 2147483647
uint	32	0 to 4294967295
long	64	-9223372036854775808 to 9223372036854775807
ulong	64	0 to 18446744073709551615
char	16	0 to 65535

Tamsayı tipleri kusursuz işlemler için çok elverişlidirler. Fakat char(karakter) tipi Unicode standartlarına uygun olarak bir karakteri temsil eder. Yukarıdaki tablodan da göreceğiniz gibi elimizde çok sayıda tamsayı tipimiz vardır. Bunlardan istedikleriminizi ihtiyaçlarımıza göre rahatça kullanabiliriz.

Bir sonraki tablo ise ondalık (floating point) ve decimal veri tiplerini,boyutlarını, hassasiyetlerini ve geçerli oldukları aralıkları listeler.

Type (Tip)	Size (in bits) (boyut)	Precision Range
float	32	7 digits 1.5 x 10 ⁻⁴⁵ to 3.4 x 10 ³⁸
Double	64	15-16 digits 5.0 x 10 ⁻³²⁴ to 1.7 x 10 ³⁰⁸

Decimal	128	28-29 decimal places 1.0 x 10 ⁻²⁸ to 7.9 x 10 ²⁸
---------	-----	--

Ondalıkli sayıları kúsuratli íşlemlerde kullanmak iyi olur. Bunun yanında muhasebe ve finansal íşlemler için decimal veri tipi daha uygun olacak şekilde tasarlanmıřtır.

Bilgisayar programları íşlemleri yaparken ifadeleri kullanırlar ve sonuç ortaya çıkartırlar. Programlarda yer alan ifadeler deęişkenler ve íşleçlerden (operatör) oluşurular. Bir sonraki tabloda íşleçleri, íşleçlerin işlem sıralarını ve işleme yönlerini görebilirsiniz.

Category (kategori)	Operator(s) (işleç/işleçler)	Associativity(işeme yönü)
Primary	(x) x.y f(x) a[x] x++ x-- new typeof sizeof checked unchecked left Unary + - ! ~ ++x --x (T)x	left
Multiplicative	* / %	left
Additive	+ -	left
Shift	<< >>	left
Relational	< > <= >= is	left
Equality	== !=	right
Logical AND	&	left
Logical XOR	^	left
Logical OR		left
Conditional AND	&&	left
Conditional OR		left
Conditional	?:	right
Assignment	= *= /= %= += -= <<= >>= &= ^= =	right

Sol işleme yönü demek işlemlerin soldan sağa doğru yapıldığıdır. Sağ işleme yönü demek işlemlerin sağdan sola doğru yapıldığıdır. Mesala atama işleçlerinin hepsinde önce sağ tarafın sonucu bulunur ve bu sonuç sol tarafa aktarılır.

Kod 2 : Unary Operators: Unary.cs

```
using System;

class Unary
{
    public static void Main()
    {
        int unary = 0;
        int preIncrement;
        int preDecrement;
        int postIncrement;
        int postDecrement;
        int positive;
        int negative;
        sbyte bitNot;
        bool logNot;
```

```

preIncrement = ++unary;
Console.WriteLine("Pre-Increment: {0}", preIncrement);

preDecrement = --unary;
Console.WriteLine("Pre-Decrement: {0}", preDecrement);

postDecrement = unary--;
Console.WriteLine("Post-Decrement: {0}", postDecrement);

postIncrement = unary++;
Console.WriteLine("Post-Increment: {0}", postIncrement);
Console.WriteLine("Final Value of Unary: {0}", unary);

positive = -postIncrement;
Console.WriteLine("Positive: {0}", positive);

negative = +postIncrement;
Console.WriteLine("Negative: {0}", negative);

logNot = false;
logNot = !logNot;
Console.WriteLine("Logical Not: {0}", logNot);

}

}

```

İfadeler işlemler yapılırken arka-artırma ve arka-azaltma işlemleri önce değişkenin değerini döndürür sonra değişken üzerinde artırma veya azaltma işlemini yapar. Diğer taraftan, ön-artırma ve ön-azaltma işlemleri önce değişken üzerinde artırma veya azaltma işlemini yapar sonra değişkenin son halini döndürür.

Kod 2'de unary (tek) değişken önce sıfır olarak atanıyor. Ön-artırma (pre-increment) işleci uygulandığında, unary değişkenin değeri 1'e çıkıyor ve "preIncrement" değişkenine atanıyor. Hemen arkasında Ön-azaltma(pred-decrement) işleci sayesinde unary değişkenimiz tekrar sıfır değerini alıyor preDecrement değişkenine bu değer atanıyor.

Arka-azaltma (post-decrement) işleci unary değişkenimize uygularsak bu değişkenimizin değeri değişiyor ama önce değişkenin ilk değeri postDecrement değişkenine atanıyor. Sonra ise arka-artırma (post-increment) işlecini uygularsak unary değişkenimizin değeri azalıyor fakat postIncrement değişkenin değeri unary değişkenimizin ilk değeri olarak kalıyor.

Mantıksal değil işareti, doğru ifadeyi yanlış, yanlış ifadeyi ise doğru olarak değiştirir. Kod 2'in çıktısı şöyle olacaktır :

```

>Pre-Increment: 1
>Pre-Decrement 0
>Post-Decrement: 0
>Post-Increment -1

```

>Final Value of Unary: 0
>Positive: 1
>Negative: -1
>Logical Not: True

Kod 3. Binary Operators: Binary.cs

```
using System;

class Binary
{

    public static void Main()
    {
        int x, y, result;
        float floatResult;

        x = 7; y = 5;

        result = x+y;
        Console.WriteLine("x+y: {0}", result);

        result = x-y;
        Console.WriteLine("x-y: {0}", result);

        result = x*y;
        Console.WriteLine("x*y: {0}", result);

        result = x/y;
        Console.WriteLine("x/y: {0}", result);

        floatResult = (float)x/(float)y;
        Console.WriteLine("x/y: {0}", floatResult);

        result = x%y;
        Console.WriteLine("x%y: {0}", result);

        result += x;
        Console.WriteLine("result+=x: {0}", result);

    }

}
```

Kod 3: te birçok aritmetik işlemler yapılıyor. Bu işlemlerin sonucunu da sizler tahmin edebilirsiniz...

"floatResult" değişkeninin tipi ondalıklı sayı tipi olduğu ve "x" ve "y" değişkenlerimiz tamsayı tiplerinde oldukları onları açık biçimde ondalıklı sayı tipine çevirdik (explicitly cast) ve bu şekilde işlem yaptık.

Yukarıdaki kod parçasında bir de kalan (remainder %) işlecinin kullanımına dair örnek verdik. Bu işleç, iki sayının bölümünden kalan sayıyı sonuç olarak döndürür.

Son olarak yazdığımız ifadede yer alan atama işleci de (+=) C/C++ ve C# programcılarının sıklıkla kullandıkları bir atama ve işlem yapma türüdür. Bu ifade aslında şu ifadenin kısa yoludur : "result = result + x".

Şimdiye kadar burada sıkça gördüğünüz diğer veri tipi ise string (karakter dizisi veya karakter katarı)'dir. String veri tipi Unicode karakter tiplerinin bir listesini içerirler ve tek çift tırnak işaretleri arasında yazılırlar.

C# Kontrol yapıları ve seçme işlemleri

Bu derste C# dilindeki ifadeler, tipler ve değişkenler anlatılacaktır. C# dilinde kullanılan seçme veya kontrol ifadelerini öğreneceksiniz.

Dersimizin hedefleri :

- "İf" (eğer) ifadesinin kullanımı "
- "switch" (çoktan-seç) ifadesini kullanımı "
- "break" ifadesinin "switch" ifadesi içerisinde nasıl kullanıldığı
- "goto" ifadesinin yerinde ve etkili kullanılması

Önceki derslerimizde, her program belirli ifadeleri sırasıyla çalıştırıp bitiyordu. Program içinde inputlara veya program içinde yapılan hesaplara göre değişik işlemler yapılmıyordu. Bu derste öğrendiklerimiz de programlarımızın belirli şartlara göre değişik şekillerde çalışmasını sağlayacaktır. İlk seçme ifademiz "if". "if" kontrol yapısının 3 temel formu vardır.

Kod 1 : IF yapısının değişik formlarda kullanımı : IfSelection.cs

```
using System;

class IfSelect
{
    public static void Main()
    {
        string myInput;
        int myInt;

        Console.WriteLine("Please enter a number: ");
        myInput = Console.ReadLine();
        myInt = Int32.Parse(myInput);

        // Single Decision and Action with brackets
        if (myInt > 0)
```

```

    {
        Console.WriteLine("Your number {0} is greater than zero.", myInt);
    }

    // Single Decision and Action without brackets
    if (myInt < 0)
        Console.WriteLine("Your number {0} is less than zero.", myInt);

    // Either/Or Decision
    if (myInt != 0)
    {
        Console.WriteLine("Your number {0} is not equal to zero.", myInt);
    }
    else
    {
        Console.WriteLine("Your number {0} is equal to zero.", myInt);
    }

    // Multiple Case Decision
    if (myInt < 0 || myInt == 0)
    {
        Console.WriteLine("Your number {0} is less than or equal to zero.",
myInt);
    }

    else if (myInt > 0 && myInt <= 10)
    {
        Console.WriteLine("Your number {0} is between 1 and 10.", myInt);
    }

    else if (myInt > 10 && myInt <= 20)
    {
        Console.WriteLine("Your number {0} is between 11 and 20.", myInt);
    }

    else if (myInt > 20 && myInt <= 30)
    {
        Console.WriteLine("Your number {0} is between 21 and 30.", myInt);
    }

    else
    {
        Console.WriteLine("Your number {0} is greater than 30.", myInt);
    }

}
}

```

Kod 1'de tüm program boyunca tek bir değişkeni kullanıyoruz, "myInt". Kullanıcıdan etkileşimli olarak veri almak için önce "Lütfen bir sayı giriniz :" iletisini konsula yazdırıyoruz. "Console.ReadLine()" ifadesi ile program kullanıcıdan bir değer girmesini bekler. Bir rakam yazılınca ve enter tuşuna basılınca program yazılan değeri önce string

tipinde olan myInput değişkenine atanıyor. String olarak alınan veriyi program tamsayı tipinde bir değişken olarak kullanmak istediği için "myInput" tamsayı tipine dönüştürülmeli. Bu dönüşüm için "Int32.Parse(myInput)" komutunu kullandık (tip dönüşümleri ve Int32 gibi veri tipleri ileriki derslerde incelenecektir.) Daha sonra, dönüşümün sonucu "myInt" isimli değişkene aktarılıyor.

Artık istediğimiz tipte bir veriye sahibiz ve bunu "if" ifadesi içinde işleyebiliriz. "if" ifade bloğunun ilk formu şu şekildedir : if(mantıksal ifade) { "mantıksal ifade"nin doğru olması durumunda yapılması gerekenler }. Öncelikle "if" anahtar kelimesi ile başlamalıyız. Sonrası parantezler arasındaki mantıksal ifade. Bu mantıksal ifadenin doğru veya yanlış olması bulunur. Biz programımızda kullanıcıdan aldığımız sayının sıfırdan büyük olup olmadığını kontrol ediyoruz ">0" ile. Eğer sayı sıfırdan büyükse, mantıksal ifadenin sonucu doğrudur ve { } parantezleri arasındaki kod bloğu çalıştırılır. Eğer mantıksal ifade yanlış bir sonuç üretirse { } arasındaki kod bloğu çalıştırılmadan bloktan sonraki ifadelere geçer.

İkinci "if" ifadesi aslında birincisi ile aynıdır, ikincisi sadece blok içinde değildir. Eğer boolean ifade doğru sonuç üretirse, bu ifadeden hemen sonraki çalışır. Boolean ifadenin yanlış olması durumunda ise bu ifadeden hemen sonraki ifade çalıştırılmadan bir sonraki ifadeye geçilir ve o ifade çalıştırılır. Bu şekildeki "if" yapısını, boolean ifadenin doğru olmasında sadece bir tane ifade çalıştırılacaksa yeterlidir. Buna karşın "if" ifadesinin sonucuna göre birden fazla ifade işleme konulacaksa blok olarak { } parantezleri arasında yazılır. Benim kişisel önerim "if" den sonra çalıştırılacak ifade sayısına bakmadan, bu ifade(leri) her durumda blok olarak yazmaktır. İleride yeni programın okunmasında ve yeni işlevler eklenmesinde size hatalardan kaçmanıza yardım eder.

Birçok zaman size eğer/değilse türünde çalışacak bir "if" yapısı gerekebilir. Üçüncü tip "if" ifadesi eğer doğru değer üretirse şunları yap, doğru değilse "else" anahtar sözcüğünden sonraki kodları çalıştır türünde bir yapısı olarak yazılmalıdır.

Birden fazla mantıksal ifadeyi işlememiz gerektiğinde ise, if/else if/else tipinde bir "if" yapısını kullanmak gerekir. Dördüncü örneğimizde bu tip bir if yapısını görebilirsiniz. Bu tip yapı yine "if" ve boolean ifadesi ile başlar. Boolean ifade doğru ise hemen alttaki bloktaki kodlar çalıştırılır. Bunun yanında, boolean ifadenin değişik durumlarına göre "else if" iç yapısı kullanılır. "else if" de aynı if gibi bir boolean ifadeyi alır ve sonucu doğru hemen sonraki bloktaki kodları çalıştırır.

Boolean ifadenin alabileceği tüm ihtimallere göre bu şekilde "else if" ifadeleri sıralanabilir fakat en sonda bir "else" ile yukarıdaki tüm şartların yanlış olması durumunda çalıştırılacak kodları belirleriz. Bu dördüncü "if" yapısında da yine sadece bir tane if/else blok yapısı çalıştırılır.

"switch" yapısı da "if/else if/else" yapısına çok benzer.

Kod 2 : Switch ifadeleri : SwitchSelection.cs

```
using System;

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



```

    string myInput;
    int myInt;

begin:

    Console.WriteLine("Please enter a number between 1 and 3: ");
    myInput = Console.ReadLine();
    myInt = Int32.Parse(myInput);

    // switch with integer type
    switch (myInt)
    {
        case 1:
            Console.WriteLine("Your number is {0}.", myInt);
            break;
        case 2:
            Console.WriteLine("Your number is {0}.", myInt);
            break;
        case 3:
            Console.WriteLine("Your number is {0}.", myInt);
            break;
        default:
            Console.WriteLine("Your number {0} is not between 1 and 3.", myInt);
            break;
    }

decide:

    Console.WriteLine("Type \"continue\" to go on or \"quit\" to stop: ");
    myInput = Console.ReadLine();

    // switch with string type
    switch (myInput)
    {
        case "continue":
            goto begin;
        case "quit": Console.WriteLine("Bye."); break;

        default:
            Console.WriteLine("Your input {0} is incorrect.", myInput);
            goto decide;
    }
}
}
}

```

Kod 2'de birkaç tane "switch" yapısı örneğimiz var. "switch" yapısı yine "switch" anahtar kelimesi ile başlar ve sınanacak değişkeni parantez içinde belirtiriz. Switch yapısının çalışması için şu veri tiplerinden bir tanesini kullanmak gerekir : sbyte,short,ushort,int, long, ulong, char, string, or enum (enum daha sonraki bir derste işlenecektir.)

Birinci örneğimizdeki "switch" yapısı int tipinde bir değer almaktadır. Tamsayı

değişkenimizin alabileceği değerlere göre değişik işlemler yapabiliriz. "myInt" değişkenimizin her bir ihtimalini değerlendirirken "case" anahtar kelimesini, muhtemel değerini ve iki nokta üst üste ":" yapısında bir sına yapıyoruz. Örneğimizde, "case 1 :", "case 2: ", ve "case 3:" şeklinde yazdık. Sınama sonuçlarından uygun olanın hemen altında kod bloku yer alır. Bu kod blokundan sonra ise "break" veya "goto" ifadelerini kullanmamız gerekir.

İsterseniz "default" seçeneğini de "switch" ifadesi ile birlikte kullanabilirsiniz. "default" ifadesinin altındaki kod bloku, "default"tan önceki "case"lerin hiçbirini sınamayı geçemediği zaman çalışır ve tüm "case"lerden sonra gelir.

Her "case" 'den sonra "break" ifadesinin zorunlu olduğunu tekrar hatırlatalım. "break" ifadesi "switch" yapısından dışarı çıkmayı ve alttaki kodlara geçmemizi sağlar. "default" anahtar kelimesinin kod blokundan hemen sonra "break" koymak programcının isteğine kalmıştır. Switch ifadesinde iki tane dikkat edilmesi gereken husus vardır.

Birincisi, farklı durumları (case'leri) ard arda aralarına hiç kod yazmadan sıralamaktır. Aslında burada yapılan iş, değişkenimizin birden fazla değeri için tek bir "case" kod bloku oluşturmaktır. Bir case ve hemen arkasına başka bir case yazdığımızda program otomatik olarak bir sonraki "case" 'e geçer. Aşağıdaki kodu incelediğimizde, "myInt" değişkeni 1,2, veya 3 değerlerinden herhangi birini alırsa kendi değerini ekrana yazdırıyoruz. Diğer durumda ise değişkenimizin değerinin 1 ve 3 arasında olmadığını ekrana yazdırıyoruz.

```
switch (myInt)
{
    case 1:
    case 2:
    case 3:
        Console.WriteLine("Your number is {0}.", myInt);
        break;
    default:
        Console.WriteLine("Your number {0} is not between 1 and 3.", myInt);
        break;
}
```

Kod 2'de yer alan ikinci "switch" yapısı ise "goto" ifadesinin nasıl kullanılacağını göstermek amacıyla yazılmıştır. "goto" , programın belirli bir kısmında yer alan, özel etiket (label) ile belirtilmiş kısma atlamasına ve oradan itibaren çalışmaya devam etmesine yarar. Programımızda kullanıcı "continue" yazarsa "begin" olarak belirlenmiş etikete gider ve oradan çalışmaya devam eder. Aslında bu şekilde "goto" kullanmak etkili bir döngü olur. Eğer kullanıcı "end" yazarsa program "bye" yazar ve döngüden programımız çıkar.

Açıkça görülüyor ki "goto" kelimesini kullanmak bize programda belirli şartlar altında güç kazandırır. Yine de "goto" ifadesini programda sık bir şekilde kullanmak "sipagetti" kod olarak adlandırılan programlamaya yol açabilir ki, bu tür kodlama programı hem okurken hem de hataları ayıklarken büyük sorunlara sebep olabilir.

C# ile Temel Windows Formları Oluřturma

Bu yazımızda Windows programlamanın temel elemanlarından olan windows formlarının nasıl oluşturulduğunu ve nasıl kullanıldığını göreceğiz, windows formlarını açıklarken basit bir dört işlem yapan hesap makinası oluşturacağız.Windows formları derken neyi kastediyoruz? Textbox, label, button gibi önemli elemanların hepsi birer windows formudur. Bu windows formlarına ulaşmak için [System.Windows.Forms](#) isimalanını kullanıyoruz. Ve tabi ki programımızın aktif bir windows uygulaması olarak çalışması için de aşağıdaki isimalanlarını projemize ekliyoruz.

```
using System.Drawing;  
using System.Collections;  
using System.ComponentModel;
```

Programımızı yazmaya başlamadan önce programımızın kodlarını içerecek bir isim alanı oluşturalım. Ben buna CsHesapMakinası adi verdim.(Makaleyi okurken kaynak kodu incelemenizi tavsiye ederim) Siz istediğiniz başka bir isim kullanabilirsiniz.

Daha önceki makalelerimizde belirttiğimiz gibi programımızın çalışması için derleyiciye programın başlangıç noktasını bildirmemiz gerekirdi.Bu başlangıç noktası da genelde main() fonksiyonu oluyordu. Kodumuzu öyle ayarlayacağız ki main() fonksiyonu icra edildiğinde çalıştırmak istediğimiz windows formu ekranda görünsün. Bunun için main içine aşağıdaki kodu yazıyoruz.

```
Application.Run(new Form1()); // Yeni bir Form1 nesnesi oluşturularak uygulama olarak başlatılıyor.
```

Şu an için Form1 hakkında en ufak bir bilgiye sahip değiliz.Peki bu Form1 nasıl oluşturulacak. Yukarıda da bahsettiğimiz gibi Form1 sınıfından bir nesne oluşturmak için [System.Windows.Forms](#) isimalanını kullanmalıyız. Bu yüzden bu isim alanının altında bulunan Form sınıfından yeni bir Form1 sınıfı türetmemiz gerekecek, bu türetme işlemi aşağıdaki gibidir. Form1 sınıfını türettikten sonra Form1' içinde bulunacak elemanları tanımlıyoruz.4 işlemi yapmak için 4 buton, işleme giren değerler için 2 textbox ve 2 tane de label formu tanımlıyoruz.Tanımlama işlemi aşağıdaki gibidir.

```

public class Form1 : System.Windows.Forms.Form
{
    private System.Windows.Forms.Button button1;
    private System.Windows.Forms.Button button2;
    private System.Windows.Forms.Button button3;
    private System.Windows.Forms.Button button4;
    private System.Windows.Forms.Label label2;
    private System.Windows.Forms.TextBox deger1;
    private System.Windows.Forms.TextBox deger2;
    private System.Windows.Forms.TextBox sonuc;
    private System.Windows.Forms.Label isaret;
}

```

Şimdi bu windows formunun ekrana nasıl basıldığını inceleyelim. Main() işlevi içinde yeni bir Form1 nesnesi yaratıldığında Form1 nesnesine ait kurucu işlev olan Form1() işlevi çağrılır. (Kaynak kodu inceleyin). Form1() kurucu işlevinde ise InitializeComponent(); adlı bir fonksiyon çağırılarak Form1 nesnesine ait olan üye elemanlarla (button, label, textbox vs) ilgili ilk işlemler yapılır. Form1 açıldığı zaman Form1 içinde bulunan elemanlarla ilgili yapmak istediğimiz ilk özellikleri InitializeComponent() fonksiyonu içinde yapıyoruz.

```

private void InitializeComponent()
{
    // form1 içinde yer alacak elemanlar yaratılıyor(kaynak kodu inceleyin)

    this.deger2 = new System.Windows.Forms.TextBox();
    this.sonuc = new System.Windows.Forms.TextBox();
    this.button2 = new System.Windows.Forms.Button();
    this.isaret = new System.Windows.Forms.Label();
    this.label2 = new System.Windows.Forms.Label();
    this.SuspendLayout();
    //
    // elemanlarla ilgili başlangıç özellikleri veriliyor.
    //
    this.deger1.Location = new System.Drawing.Point(16, 8); //deger1 adlı textbox
    için yer bildirimi
    this.deger1.Name = "deger1";
    this.deger1.TabIndex = 0;
    this.deger1.Text = "0";

    this.name="Form1"; //Form1 nesnesinin kendisi için this anahtar sözcüğünü
    kullanıyoruz
    this.text = "Hesap Makinası ";
    this.BackColor = System.Drawing.Color.FromArgb(((System.Byte)(255)),
    ((System.Byte)(128)), ((System.Byte)(0)));
    this.ClientSize = new System.Drawing.Size(400, 149);
}

```

Şimdi sıra elemanlarla ilgili olayların birbirleri ile ilişkisine. Mesela bir buton formunun click olayının form tarafından yakalanabilmesi için aşağıdaki satırları yazmalıyız.

```

this.button4.Click += new System.EventHandler(this.button4_Click);
private void button4_Click(object sender, System.EventArgs e)
{
    isaret.Text="/";
    sonuc.Text=System.Convert.ToString(System.Convert.ToInt32(deger1.Text)/
    System.Convert.ToInt32(deger2.Text));
}

```

buton4_Click() işlevinde, çalışma zamanında bir nesnenin özelliklerinin nasıl değiştirildiğini görüyoruz.button4 bölme işlemi yaptığından isaret.Text="/"; yazdık. sonuc adlı textbox formunun Text özelliği bir string ifadesi olduğu için işlemlerimizi yaptıktan sonra sonuc.text ifadesine atama yapabilmek için System.Convert isim alanında bulunan ToString işlevini kullanarak ifadeyi String türüne dönüştürüyoruz. Aynı şekilde String olarak aldığımız türler için aritmetik işlem yapabilmek için yine aynı isim alanında bulunan.ToInt32 işlevi ile String türünü int32 formatına dönüştürüyoruz. Bütün bu işlemleri 4 butonumuz için yaptığımızda dört işlem yapabilen basit ve bol bol bug içeren (unutmayın amacımız sadece formların kullanımını öğrenmek) bir hesap makinamız olacak.

.NET Teknolojilerine Giriş

Günümüzde bilgisayar dünyasında internet olmazsa olmaz derecede önemli bir yer edinmeye başladı. Artık insanlar ev ve işyerlerinde kullandıkları uygulamalarına da internet üzerinden erişip kullanmak istiyorlar. Bu internetin getirdiği özgürlüğün kaçınılmaz bir sonucudur.Peki yazılım dünyası buna hazırmıydı? Geliştirilen her programı kolayca internet ortamında da çalıştırabilirmiydik? Bu soruların cevapları bir sene öncesine kadar hayır, olamaz veya şu andaki sistemler bu denli özgürlüğü bize sağlamıyor türündendi.

Microsoft'un ASP'si ile veya PHP ile yapılan uygulamalar tam olarak insanların isteklerine cevap veremiyordu. Her ne kadar iyi ve gelişmiş web uygulamalarını bir yere kadar yapabiliyorduksa da belirli bir noktadan sonra C++,Delphi veya VB ile geliştirdiğimiz modülleri web uygulamamıza ekleyerek sorunlarımızı halletmeye çalışıyorduk. Tabi bu tür yöntemler programın gelişme süresini uzatıyordu. Zamanın giderek önem kazandığı bir devirde haliyle programlarımızı da hızlı bir şekilde geliştirmemiz gerekiyor(du). Hızlı uygulama geliştirme(Rapid Application Development- RAD) geleneksel programlama araçlarıyla ve prgramcının yetenekleriyle çözüm bulunacak bir mesele değil. Artık programlama dilleri, dille birlikte gelen kütüphaneler ve bunlar hakkındaki dokümantasyonları ile birlikte değerlendiriliyor.

.NET ile birlikte programcının hizmetine sunulan 3400'den fazla sınıf, modern anlamda çok güzel bir geliştirme ortamı sunuyor. Bu sayede programlamaları daha hızlı bir şekilde geliştirme imkanına sahip bulunuyoruz. .NET kullanarak yazdığımız ASP.NET, Windows Forms veya mobil cihazlar için geliştirdiğimiz bir uygulamayı birinden diğerine dönüştürmek işi çok kolay bir şekilde yapılabiliniyor. Bu sayede aynı anda hem windows hem de web uygulamaları geliştirmek çok hoşunuza gidecektir :-).

.NET framework'unun bize sunduğu diğer güzel bir özellik ise platform bağımsızlığıdır. Artık yazdığınız Windows uygulamaları sadece Windows yüklü sistemlerde değil, .NET framework'unun kurulu olduğu tüm platformlarda çalışabilecektir. Her ne kadar şimdilik bu alt yapının sadece Windows versiyonuna sahip olsak da Linux grupları tarafından bu alt yapının Linux versiyonunu çıkartma yönündeki çabalar uzun bir süredir devam etmektedir.

Peki bunca hoş özellikleri bize sağlayan .NET alt yapısında program yazarken hangi dili veya dilleri kullanmak zorundayız? Bu konuda Microsoft çok radikal bir karar alarak gelecek için hazırlanmış yeni alt yapıda Common Language Runtime (CLR) ile uyumlu her .NET dilini kullanmamıza olanak sağlıyor. .NET ile gelen SDK'da C#, VB.NET ve Js.NET kullanarak program yazabiliyoruz. Diğer taraftan 30'un üzerinde programlama diliyle .NET uygulaması geliştirebilirsiniz.

CLR denen şey tam olarak nedir? .NET altyapısında programların çalışmasını kontrol eden ve işletim sistemi ile programımız arasında yer alan arabirimdir. Normalde yazdığımız programlar derlenirken makine diline çevrilirdi ve program bu şekilde işletim sistemi ile direkt bağlantı kurarak çalışırdı. Fakat platform bağımsız bir geliştirme ve yürütme ortamı istiyorsanız ne olacak? İşte tam bu anda CLR devreye girer ve .NET programlarını farklı platformlarda makineye ve işletim sistemine göre programımızı çalıştırır. Normalde bir Windows, Linux veya MACOS kurulu sistemler aynı programın kodunu çalıştıramazlar. Bu platformlar için programın ayrı ayrı yazılıp, onlara göre hazırlanmış derleyicilerde derlenmesi gerekir. Dünyada çok sayıda yaygın platform olduğunu düşünürsek, bunların herbiri için ayrı ayrı derleme işlemini tek bir işletim sisteminde yapmamız imkansız gibidir. Bu durumda çözüm , ortak bir aradil kullanmak ve herbir platform için bu aradile çevrilmiş programın kodunu çalıştıracak altyapıları hazırlamaktır.

Şimdi şu soruya sıra geldi: "İyi de .NET hangi aradili kullanıyor?" Sorumuzun cevabı MSIL(Microsoft intermediate Language) .NET platformunda hangi dili kullanırsak kullanalım yazdığımız programın kodu direkt olarak makine diline değil de MSIL'e çevrilir. Bu sadece programı çalıştırdığımız sistemde kurulu olan CLR çalışma anında MSIL kodlarını çevirerek programımızı çalıştırır, çalışma anında derleme işlemini ise JIT derleyicileri (Just in Time compilers) üstlenir.

Gelecek makalemizde JIT'ler, MSIL language, CTS (Common Type System) gibi daha teknik konuları detaylı olarak ele almayı düşünüyorum. Sizlere kolaylıklar dilerim.

.NET'in CLR, CTS ve JIT derleyicileri

Önceki yazımızda "dot NET" platformu konusuna giriş yapmıştık. [\(Yazıyı okumak için tıklayın\)](#) Burada ise daha detaylı olarak .NET kavramlarını inceleyeceğiz ve .NET'le Java'nın karşılaştırıldığı bir testin sonuçlarına yer vereceğiz.

.NET platformunda istediğimiz programlama dili ile program yazabileceğimizi önceki yazımızda söylemiştik. Bunun için tek şart, kullandığımız dilin .NET için yazılmış olan bir derleyicisine ihtiyacımız olduğudur. .NET uyumlu programlama dili oluştururken belirli standartlara uyulması gerekir. Bu standartlar CLS (Common Language Specifications - Dillerin ortak özellikleri) ile belirlenmiştir. CTS(Common Type System) ise veri tipleri, nesneler, arayüzler ve programlama dillerine ait özellikleri tanımlar ve CLS'in bir parçası olarak karşımıza çıkar. CLS'de tanımlanmış kurallara uymak şartı ile istersek kendi programlama dilimizi dahi geliştirebiliriz veya herhangi bir dili .NET platformunda uygulama geliştirmek üzere değiştirebiliriz.

CLR ,programlarımızı değişik şekilde derleyebilir. Varsayılan derleme türü JIT(Just IN TIME- çalışam anında derleme) 'dır. Program çalışırken daha önce derlenmemiş bir parçasına gelince hemen o kısmı da derler ve bunu hafızda chach'e koyar. Tekrar aynı program parçasını çalıştırmak gerekirse burayı hafızadan çalıştırır. Eğer RAM 'imizi yeteri kadar büyükse, programın tamamı derlenmiş ve hafızada depolanmış durumda olabilir. Bu durumda programımız çok hızlı çalışır.

Hafızamızın yeteri kadar büyük olmadığı durumlarda EconoJIT (Ekonomik JIT) derleyicisini kullanabiliriz. Bu derleyici ile programın derlenmiş kısımları hafızada depolanmaz ve her seferinde aynı program parçası derlenir. Tabi ki bu derleyici normal JIT'e göre programlarımızı daha yavaş çalıştırır. Ama RAM 'imizi çok daha az kullanır.

CLR ile gelen üçüncü derleyicimiz PreJIT(ön JIT derleyicisi) ise derleme işini program çalışmadan önce yapar ve tüm makine kodlarını bir yerde saklar. Çalışma anında çok hızlı olan programımız diğer JIT derleyicileriyle derlenmiş olanlara nazaran çok hızlı çalışır.

Kolayca görebileceğimiz birkaç noktaya da parmak basmak istiyorum. .NET ile yazdığınız programlar diğerlerine göre yavaş çalışır. Çünkü iki defa derleme aşamasından geçerler,

program kodu MSIL'ye, MSIL ise makine koduna çevrilir. Diğer taraftan .NET ile programlarımız platform bağımsız olacak, .NET uyumlu herhangi bir dil ile program geliştirebileceğiz ve programımız CLR altında daha güvenli bir şekilde çalışacaktır.

.NET performans testi linkindeki sonuçlara göre : Genelde C# Java'dan 3.30 kat daha hızlı. C# Visual C++ 6.0'dan ise 3.11 kat daha hızlı çalışıyor. Hatta VB.NET kodu VB 6.0'a nazaran 46.45 kat daha hızlı çalışıyor. :-)

Visual C# ile Programlamaya Giriş

Visual C#, Visual Studio ailesinin yeni üyesidir, bu yeni dil c ve c++ temelleri üzerine kurulmasına rağmen komponent temelli programlama tekniğini geliştirmek için birtakım yeni özellikler eklenmiştir. C# dilinin sentaksı C ve C++ programcılarına çok tanıdık gelecektir. Bundan şüpheniz olmasın.



Genel Açıklamalar

Bu yazıda göreceğimiz basit uygulamada QuickSort algoritmasını kullanarak nasıl basit bir C# projesinin oluşturulduğunu göreceğiz. Bu uygulamada bir c# programında en sık kullanılan yapılardan olan dosyaya ve console ekranına okuma/yazma, fonksiyon oluşturma ve basit dizilerin kullanımı açıklanacaktır.

Bu yazı kesinlikle C# dilinin tüm özelliklerinin anlatıldığı yazı değildir. Bu yazı C# dilini öğrenmek için sizlere bir başlangıç noktası sunar.

Önerilen Kaynaklar

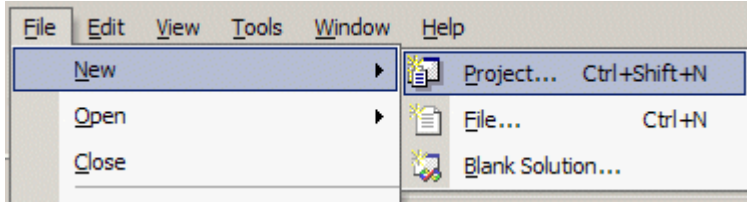
Visual Studio.NET (Beta 2 veya sonrası) örnek kaynak kodu derlemeniz için gereklidir. C/C++ bilgisi size bu yazıyı anlamanızda yardımcı olabilir ama gerekli değildir.

Adım 1. Projeye Başlama

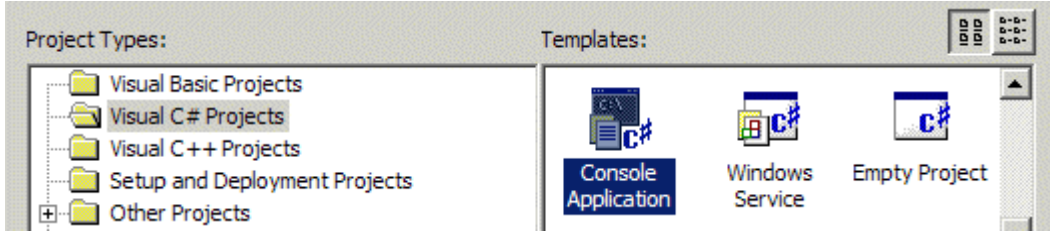
Visual Studio ile program geliştirme organizasyonu solution(çözüm) çalışma alanları üzerindedir. Solution dediğimiz ortam bir veya daha fazla projeyi içerebilir. Bu makale için tek bir C# projesi içeren bir solution oluşturacağız.

Yeni bir proje oluşturmak

1. Visual Studio.NET ortamından, **File | New | Project** menülerini seçin.



2. Soldan(Project Types) Visual C#, sağdan(Templates) ise Console Application butonlarını seçin.



3. Projenizin adını belirleyin ve projenizin hangi klasörde saklanacağını belirleyin. Bu klasör Visual Studio tarafından otomatik oluşturulur. Proje adı olarak ben quicksort yazıyorum. Siz istediğiniz adı verebilirsiniz.



4. OK tuşuna basalım ve yola koyulalım.

"Visual C# Solution" ortamımız

Visual Studio.NET içinde bir Visual C# projesi bulunan bir solution oluşturdu. Proje assemblyinfo.cs ve class1.cs adlı iki tane dosya içermektedir.

Bundan sonraki adımlarda projemizi nasıl derleyeceğimizi ve bu iki dosya hakkında detaylı bilgiyi öğreneceğiz.

Adım 2. Hello, World!

Kusura bakmayın ama geleneği bozmadan ilk defa C programlama dili ile yazılmış olan "Hello, World!" programını c# ile yazacağız. Bu bir gelenektir ve her yeni bir dili öğrenmeye başladığınızda bunu siz de göreceksiniz.

Kaynak Kodu Değiştirme

1. Solution Explorer 'da bulunan 'class1.cs'dosyasına çift tıklayın. Solution Explorer 'ı göremiyorsanız, view menüsünü kullanarak görünür hale getirebilirsiniz.
2. Şablonla oluşturulmuş koda aşağıda kırmızı ile yazılmış kısmı ekleyin (class1.cs).

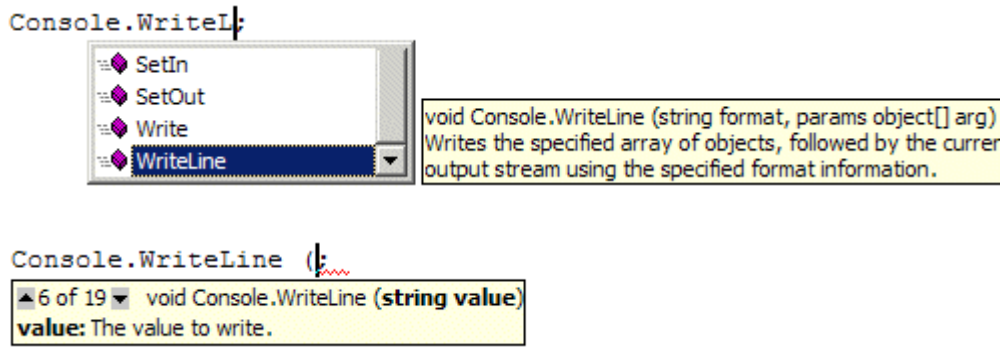
```
using System;  
  
namespace quicksort  
{  
    ///  
    /// Summary description for Class1.
```

```

///
class Class1
{
    static void Main(string[] args)
    {
        //
        // TODO: Add code to start application here
        //
        Console.WriteLine ("Hello, C#.NET World!");
    }
}

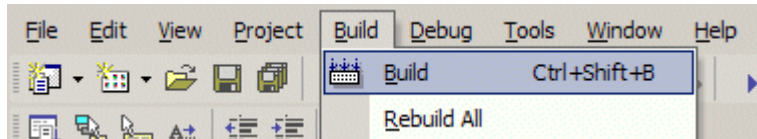
```

3. Dikkat edin siz kodunuzu yazdıkça Visual Studio size sınıflar ve fonksiyon adları hakkında bilgi verir, çünkü .NET Framework tip bilgisini yayınlamaktadır.

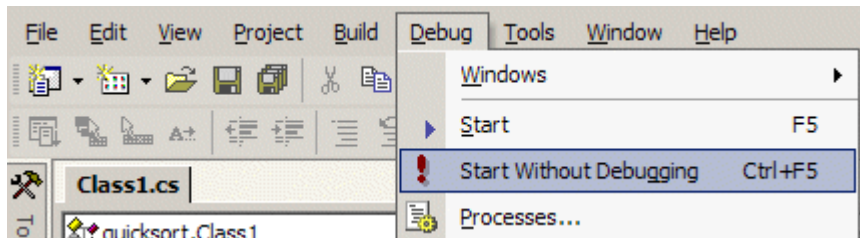


Uygulamamızı Derleyelim

1. Programımızda değişiklik yaptığımıza göre artık Build menüsünden Build 'ı seçerek programımızı derleyebiliriz.

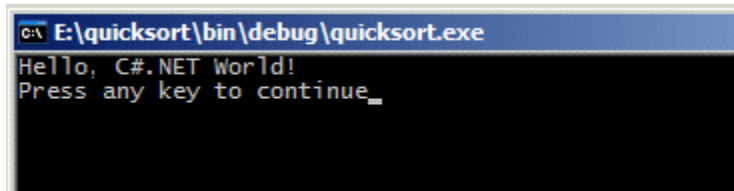


2. Hata ve mesajlar en altta bulunan "Output Window" denilen pencerede görünür. Eğer herhangi bir hata yoksa uygulamamızı Debug menüsü altında bulunan 'Start without Debugging' menüsüne tıklayarak çalıştırabiliriz.



Programımızın Çıktısı

Aşağıda programımızın Visual C# içinden çalıştırılarak oluşturulmuş çıktısının ekran görüntüsü mevcuttur.



Değişiklikleri Anlamak

System.Console sınıfına ait WriteLine() fonksiyonu kendisine argüman olarak gönderilen dizgeyi sonuna satır sonu karakteri de ekleyerek ekrana yazar.

Bu fonksiyon integer ve floating-point gibi diğer veri tiplerini de argüman olarak alabilir.

Program belleğe yüklendiğinde programın kontrolü Main() fonksiyonuna gelir. WriteLine() fonksiyonunu oraya yazmamızın sebebi budur.

Adım 3. Programın Yapısı

Şimdi basit bir Hello World uygulaması geliştirmiş olduk, şimdi de bir Visual C# uygulamasının basit componentlerini inceleyelim.

Kaynak Kod Yorumları

// karakterlerinden sonra gelen ve satırın sonuna kadar olan sözcükler yorum satırlarıdır ve C# derleyicisi tarafından görünmezler. Aynı zamanda birden fazla satıra yorum eklemek istiyorsak /* */ karakterleri arasına yorum yazarız.

```
// Bu satır derleyici tarafından görülmez
/* Aynı zamanda bu blok da
derleyici tarafından görünmez*/
```

Using Komutu

.NET Framework geliştiricilere yüzlerce yararlı sınıflar sunar. Mesela, Console sınıfı, console ekranına ait girdi ve çıktıları işler. Bu sınıflar hiyerarşik bir ağaç içinde organize edilmiştir. Aslında Console sınıfının tam ismi System.Console 'dur. Diğer sınıflar ise System.IO.FileStream ve System.Collections.Queue. içindedirler.

using komutu bize sınıfın ismini namespace(isim alanı) kullanmadan kullanabilmemizi sağlar.

Aşağıda kırmızı ile yazılan yazılar using komutunun uygulamasının sonucudur.

```
using System;
class Class1
{
    static void Main(string[] args)
    {
        System.Console.WriteLine ("Hello, C#.NET World!");
        Console.WriteLine ("Hello, C#.NET World!");
    }
}
```

Sınıf Bildirimi

C++ ve Visual Basic 'den farklı olarak C# 'da bütün fonksiyonlar bir sınıf içerisinde olmalıdır. C# 'da bir sınıf tanımlamak için class anahtar sözcüğü kullanılır. Bu durumda bizim uygulamamızda, Class1 sınıfı Main() adında bir fonksiyon içerir. Eğer sınıf bildirimini bir isim alanı blokları içine alırsak sınıflarımızı CSharp.QuickSortApp şeklinde bir hiyerarşi içine alabiliriz.

Sınıflar hakkında çok derin bir bilgi vermeyi düşünmüyorum. Fakat bizim örneğimizin neden bir parçası olduğunu açıklamakta fayda gördüm.

Main() Fonksiyonu

Program belleğe yüklendiğinde Main() fonksiyonu programın kontrolünü eline alır, bu yüzden başlangıç kodlarımızı daima Main() fonksiyonu içinde yazmalıyız. Komut satırı argümanları ise bir string dizisi olan args dizisine aktarılır. (mesela: delete x) Burada programımızın adı delete ise x bir komut satırı argümanıdır.

Adım 4. Console Girişi

Şimdi işlemlerimize bir QuickSort uygulaması geliştirerek devam edelim. İlk yapmamız gereken kullanıcıya hedef ve kaynak dosyasının isimlerini sormak olacaktır.

Source Code Modifications

1. class1.cs dosyasına aşağıda kırmızı ile yazılanları yazın. Burada sınıf ismi ve isimalanı ismi bizim için çok önemli değildir.

```
// namespaces ekleme
using System;

// namespace tanımlama
namespace MsdnAA
{
    // uygulama sınıfı tanımlama
    class QuickSortApp
    {
        // uygulama oluşturma
        static void Main (string[] szArgs)
        {
            // Programın hakkında
            Console.WriteLine ("QuickSort C#.NET Sample Application\n");

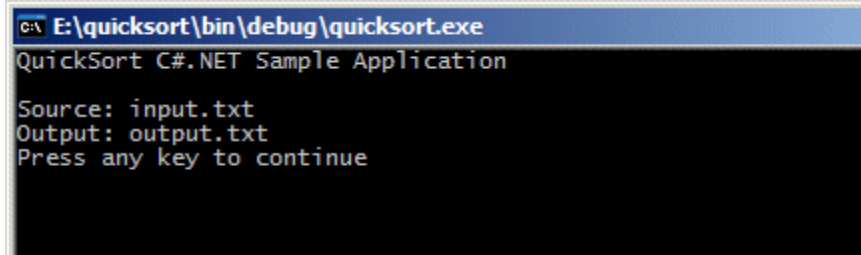
            // kullanıcıdan bilgi alma
            Console.Write ("Source: ");
            string szSrcFile = Console.ReadLine ();
            Console.Write ("Output: ");
            string szDestFile = Console.ReadLine ();
        }
    }
}
```

Console'dan Okuma

Console sınıfının ReadLine() metodu kullanıcıya bir giriş ekranı sunar ve geriye kullanıcının girdiği dizgeyi(string) geri döndürür. Bu metod bellek tahsisatını otomatik olarak kendi içinde yapmaktadır ve .NET garbage collector mekanizması sayesinde iade etmeniz gereken herhangi bir alan yoktur.

Program Çıktısı

Programı **Debug | Start Without Debugging** menülerini kullanarak çalıştırın. Aşağıda programımızın son halinin çıktısını görüyorsunuz.



```
C:\ E:\quickSort\bin\debug\quickSort.exe
QuickSort C#.NET Sample Application
Source: input.txt
Output: output.txt
Press any key to continue
```

Adım 5. Dizilerin Kullanımı

Programımız sıralama yapmadan önce girişten satırları alarak bir dizi içinde saklaması gerekir. Şimdi .NET temel sınıflarından olan ArrayList sınıfını inceleyeceğiz.

Kaynak Kod Değişikliği

1. class1.cs dosyası içinde aşağıda kırmızı ile gösterilen yerleri ekleyin.

```
// isim alanı ekleme
using System;
using System.Collections;

// isimalanı tanımlama
namespace c#nedircom
{
    // uygulama sınıfı tanımlama
    class QuickSortApp
    {
        // uygulama başlangıcı
        static void Main (string[] szArgs)
        {

            Console.WriteLine ("QuickSort C#.NET Sample Application\n");

            // Dosya isimlerini almak için komut yaz
            Console.Write ("Source: ");
            string szSrcFile = Console.ReadLine ();
            Console.Write ("Output: ");
            string szDestFile = Console.ReadLine ();

            // TODO: Read contents of source file
            ArrayList szContents = new ArrayList ();

        }
    }
}
```

ArrayList sınıfının kullanımı

ArrayList sınıfına direkt ulaşabilmek için System.Collections isimalanını projemize using komutuyla ekliyoruz. Bu sınıf dinamik olarak büyüyüp küçülebilen nesne dizileri için kullanılır. Yeni bir eleman eklemek için basit bir şekilde Add() metodunu kullanabilirsiniz.

Bu diziye eklenen yeni eleman orijinal nesne için referans olarak kullanılır, ve garbage collector alan iadesi için hazır olacaktır.

```
string szElement = "insert-me";  
ArrayList szArray = new ArrayList ();  
szArray.Add (szElement);
```

Dizinin var olan bir elemanına ulaşabilmek için, diziye ait Item() metoduna ulaşmak istediğimiz elemanın sıra(index) numarasını geçebiliriz.Kısaca [] operatörlerini kullanarak da istediğimiz elemana Item() metodunu kullanmadan da ulaşabiliriz.

```
Console.WriteLine (szArray[2]);  
Console.WriteLine (szArray.Item (2));
```

ArrayList sınıfının daha birçok metodu vardır, ancak biz bu uygulamada sadece ekleme ve okuma yapacağız. ArrayList sınıfına ait tüm metod ve özellikleri öğrenmek için MSDN kitaplığına başvurabilirsiniz.

Adım 6. File Girdi/Çıktı(Input/Output)

Şimdi isterseniz dosyadan okuma ve dosyaya yazma işlemlerini gerçekleştirelim. Dosyadaki her satırı okuyup, bir string dizisinin her elemanını bir satır gelecek şekilde ekleyeceğiz.Sonraki aşamada ise QuickSort algoritmasını kullanarak diziyi sıralı bir şekilde yazacağız.

Kaynak Kod Değişikliği

1. class1.cs dosyasına aşağıda kırmızı ile yazılanları yazın. Burada sınıf ismi ve isimalanı ismi bizim için çok önemli değildir.

```
// isimalanı ekleme  
using System;  
using System.Collections;  
using System.IO;  
  
namespace MsdnAA  
{  
  
    class QuickSortApp  
    {  
  
        static void Main (string[] szArgs)  
        {  
            string szSrcLine;  
            ArrayList szContents = new ArrayList ();  
            FileStream fsInput = new FileStream (szSrcFile,  
            FileMode.Open,FileAccess.Read);  
            StreamReader srInput = new StreamReader (fsInput);  
            while ((szSrcLine = srInput.ReadLine ()) != null)  
            {  
                // dizinin sonuna ekleme  
                szContents.Add (szSrcLine);  
            }  
            srInput.Close ();  
            fsInput.Close ();
```

```

// TODO: Buraya QuickSort fonksiyonu gelecek

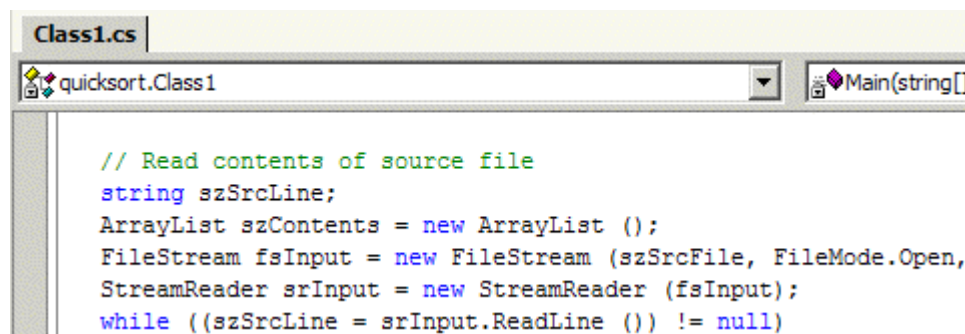
// sıraları satırları yazma
FileStream fsOutput = new FileStream (szDestFile, FileMode.Create,
FileAccess.Write);
StreamWriter srOutput = new StreamWriter (fsOutput);
for (int nIndex = 0; nIndex < szContents.Count; nIndex++)
{
    // hedef dosyaya satır yazma
    srOutput.WriteLine (szContents[nIndex]);
}
srOutput.Close ();
fsOutput.Close ();

// başarıyı kullanıcıya bildirme
Console.WriteLine ("\nThe sorted lines have been written.\n\n");
}
}
}

```

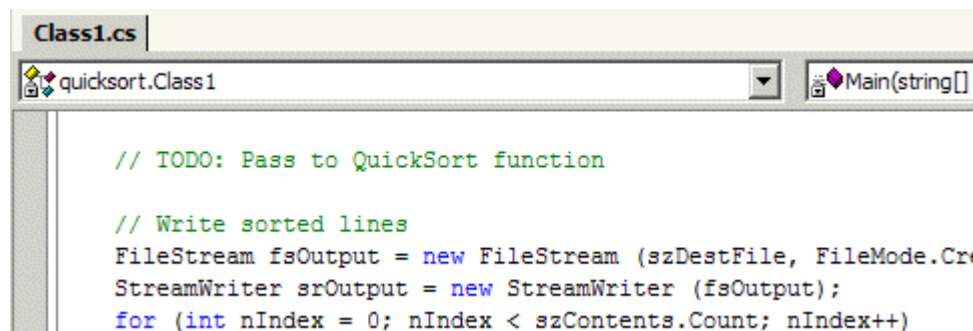
Kaynak Dosyadan Okuma

Kaynak dosyayı açmak için FileStream sınıfını ve StreamReader sınıfını kullanarak ReadLine() metodu ile dosyayı okuyoruz. ReadLine() metodunu dosyanın sonunu ifade eden NULL değerine geri dönene kadar çağırıyoruz. Döngü içinde okunan satırları dizinin içine ekliyoruz ve sonrada tüm nesneleri kapatıyoruz.



Hedef Dosyaya Yazma

Dosyaya yazarken dizinin sıralanmış olduğunu varsaydık ve öyle devam ettik. Aynı şekilde FileStream nesnesini ve StreamWriter sınıfını kullanarak hedef dosyaya yazma işlemini yaptık.



Adım 7. Fonksiyon Yaratma

Son aşamada diziye QuickSort algoritması ile sıralayan fonksiyonu yazmaya geldi. Bu fonksiyonu uygulamamızın ana sınıfının içine koyacağız.

Kaynak Kod Değişikliği

1. class1.cs dosyasına aşağıda kırmızı ile yazılanları yazın. Burada sınıf ismi ve isimalanı ismi bizim için çok önemli değildir.

```
// isim alanı ekleme
using System;
using System.Collections;
using System.IO;

namespace c#nedircom
{
    class QuickSortApp
    {
        static void Main (string[] szArgs)
        {
            ... ..

            // QuickSort fonksiyonuna parametrelerin geçişi
            QuickSort (szContents, 0, szContents.Count - 1);

            ... ..
        }

        // QuickSort fonksiyonu
        static void QuickSort (ArrayList szArray, int nLower, int nUpper)
        {
            if (nLower < nUpper)
            {
                // Ayırma ve sıralama işlemi
                int nSplit = Partition (szArray, nLower, nUpper);
                QuickSort (szArray, nLower, nSplit - 1);
                QuickSort (szArray, nSplit + 1, nUpper);
            }
        }

        // QuickSort bölmelere ayırma
        static int Partition (ArrayList szArray, int nLower, int nUpper)
        {
            // İlk elemanı bulma
            int nLeft = nLower + 1;
            string szPivot = (string) szArray[nLower];
            int nRight = nUpper;

            // Dizi elemanlarını bölme
            string szSwap;
            while (nLeft <= nRight)
            {
```



```

while (nLeft <= nRight)
{
    if (((string) szArray[nLeft]).CompareTo (szPivot) > 0)
        break;
    nLeft = nLeft + 1;
}
while (nLeft <= nRight)
{
    if (((string) szArray[nRight]).CompareTo (szPivot) <= 0)
        break;
    nRight = nRight - 1;
}

// Eğer uygunsa yer değiştirme işlemi yap
if (nLeft < nRight)
{
    szSwap = (string) szArray[nLeft];
    szArray[nLeft] = szArray[nRight];
    szArray[nRight] = szSwap;
    nLeft = nLeft + 1;
    nRight = nRight - 1;
}
}

szSwap = (string) szArray[nLower];
szArray[nLower] = szArray[nRight];
szArray[nRight] = szSwap;
return nRight;
}
}
}

```

QuickSort() Fonksiyonu

Bu fonksiyon alt sınır, üst sınır ve bir dizi olmak üzere 3 tane parametre almaktadır. QuickSort fonksiyonu Partition() fonksiyonunu çağırarak diziyi iki parçaya ayırır. Bu parçaların birinde belirlenen bir diziden önceki ,diğer parçasında ise sonraki elemanlar bulunur. Sonra fonksiyon tekrar kendini çağırarak bu iki parçanın sıralanmasını sağlar.

Yukarıdaki kodlarda yeni gördüğümüz metod CompareTo() metodudur. Bu metod iki string ifadesini karşılaştırır.

QuickSort Uygulamasını Çalıştırma

Bu adım uygulamamızın son adımı olacaktır.Şimdi programımızı derleyip çalıştırabiliriz. Programımızın çalışması için bir text dosyası oluşturmamız sıralama yapabilmek için gerekecek. Exe dosyasının bulunduğu dizini text dosyasını oluşturup programı çalıştıralım.

Name	Size	Type
example.txt	1 KB	Text Document
output.txt	1 KB	Text Document
quicksort.exe	5 KB	Application
quicksort.pdb	14 KB	Program Debug Database

example.txt - Notepad	output.txt - Notepad
File Edit Format View Help	File Edit Format View Help
Visual C++ Windows Embedded JavaScript Speech API	.NET Framework ASP.NET BizTalk Server DirectX API

Programın Çıktısı

Aşağıda programımızı çalıştırdıktan sonraki ekran çıktısı mevcuttur. Yukarıdaki ekranda ise output.txt dosyasında meydana gelen değişiklikleri görebilirsiniz.

```

C:\E:\quicksort\bin\debug\quicksort.exe
QuickSort C#.NET Sample Application

Source: example.txt
Output: output.txt

The sorted lines have been written.

Press any key to continue

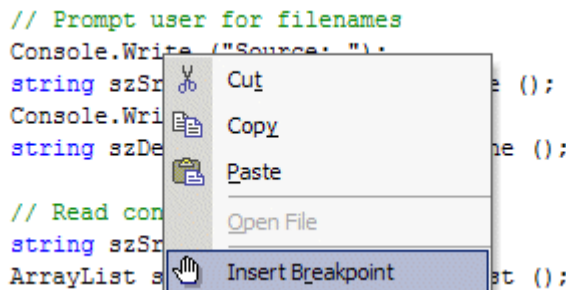
```

Adım 8. Debugger Kullanmak

Debugger aracı programımızın problemlerini çözmek için önemli ve gerekli bir araçtır. İyi bir başlangıç yaptığımıza göre son aşamada programımız içinde nasıl dolaşacağımıza ve QuickWatch' ı nasıl kullanacağımıza bakalım.

Breakpoint' leri Ayarlama

Program Debugger içinde çalışırken eğer breakpoint olan bir noktaya gelinirse program sonlandırılır ve debugger control 'ü bizim elimize geçer. Herhangi bir satıra breakpoint koymak için ilgili satıra sağ tıklayıp aşağıdaki gibi Insert BreakPoint menüsüne tıklayın.

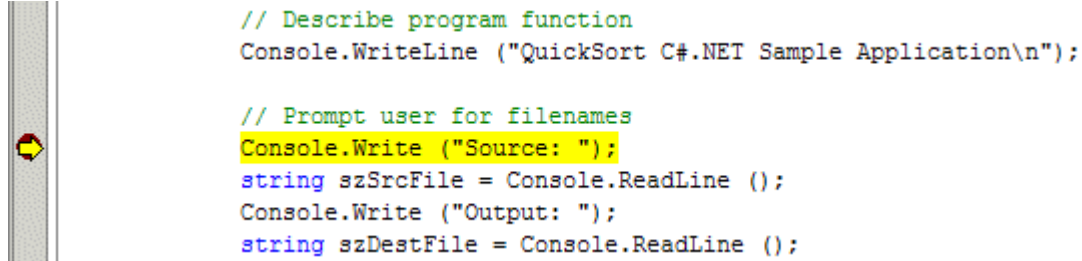


BreakPoint bulunan satırlar kırmızı ile gösterilir. BreakPoint ' ı kaldırmak için ilgili satıra sağ tıklayıp Remove BreakPoint menüsünü seçin.

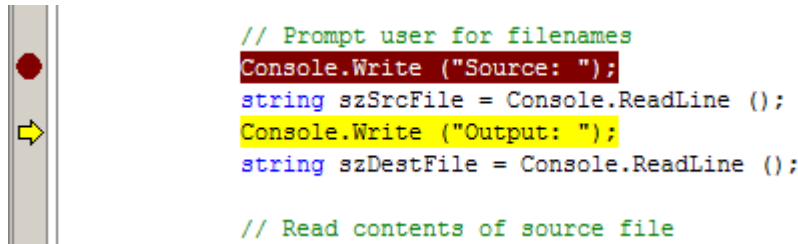
Program içinde Adım Adım ilerlemek

Aşağıdaki şekilde görüldüğü gibi bir breakpoint oluşturduktan sonra debugger ile programımızı çalıştıralım. Debug menüsünden daha önceden de yaptığımız gibi Start Without Debugging yerine Start menüsüne tıklayın. Bu şekilde programı çalıştırdığımızda program debugger ile çalışmaya başlayacaktır, ve breakpoint 'ler aktif olmaya başlayacaktır.

Program breakpoint 'in olduğu noktaya geldiğinde, debugger programın kontrolünü alır. Aşağıdaki şekilde görüldüğü gibi sarı okla programın geldiği nokta gösterilir.



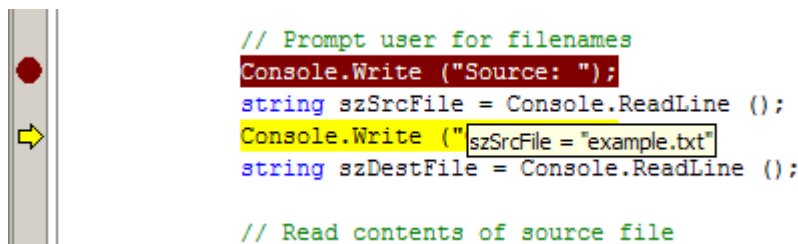
Kod içinde adım adım ilerlemek için menü çubuğundan Debug | Step Over ' ı seçerek sarı okun hareketini izleyin. Debug | Step Into komutu gitmek istediğimiz fonksiyona gitmemizi sağlar. Aşağıda iki defa Step Over menüsünü seçtikten sonra kaynak kodumuzun görüntüsü mevcuttur.



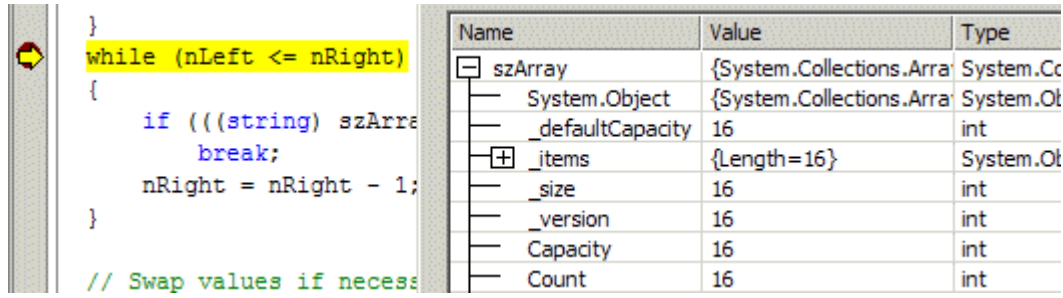
Eğer programımızın diğer bir sonlandırıcı nedene (yeni bir breakpoint, exception , exit, debug) gelene kadar devam etmesini istiyorsanız Debug | Continue menülerini seçin.

Değişken Değerlerini Takip Etme

Debugger 'ın kontrolü bizde iken farenin imlecini kaynak kodda istediğimiz değişken üzerine götürerek değişkenin o anki değerini öğrenebiliriz.



Aynı zamanda bir değişkene sağ tıklayarak QuickWatch menüsünü açabilirsiniz. QuickWatch ile değişken hakkında daha detaylı bilgiler elde edebilirsiniz. Mesela diziler hakkında aşağıdaki şekilde görüldüğü gibi bilgiler elde edilebilir.



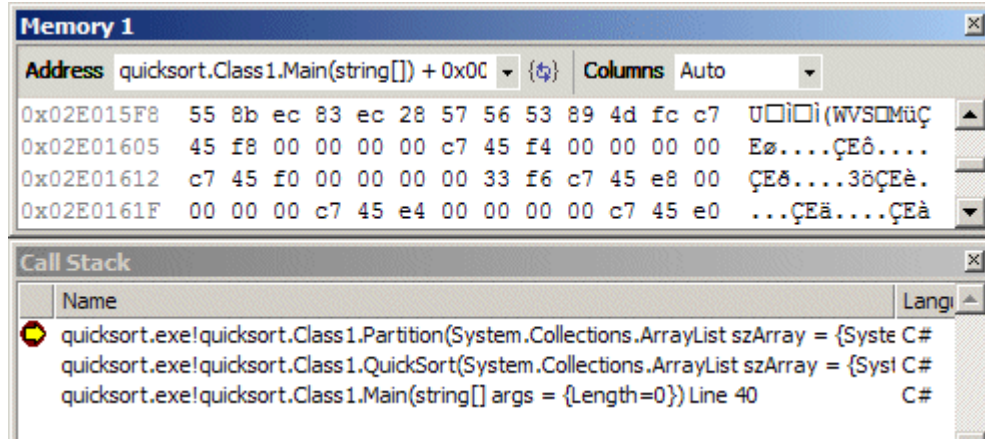
The screenshot shows a C# code snippet in the left pane and the Watch window in the right pane. The code is a while loop that iterates over an array, breaking if a condition is met, and then swaps values if necessary. The Watch window shows the state of the `szArray` variable, which is a `System.Collections.ArrayList` with a capacity of 16 and 16 items.

```
    }  
    while (nLeft <= nRight)  
    {  
        if (((string) szArray[nLeft]) > szArray[nRight])  
            break;  
        nRight = nRight - 1;  
    }  
    // Swap values if necessary
```

Name	Value	Type
szArray	{System.Collections.ArrayList}	System.Collections.ArrayList
System.Object	{System.Collections.ArrayList}	System.Collections.ArrayList
_defaultCapacity	16	int
_items	{Length=16}	System.Collections.ArrayList
_size	16	int
_version	16	int
Capacity	16	int
Count	16	int

Diğer Debugger Araçları

Visual Studio yukarıda bahsettiğimiz araçları dışında farklı araçlarda barındırır. Mesela Call Stack Wierver aracı ile o an hangi fonksiyonun çağrıldığı ile ilgili bilgiyi elde edebilirsiniz. Bellekteki değerleri ve bir prosesdeki thread 'leri de aynı anda görebileceğimiz araçlar mevcuttur.



The screenshot shows the Memory and Call Stack windows in Visual Studio. The Memory window displays the memory address `quicksort.exe!quicksort.Class1.Main(string[]) + 0x00` and the corresponding memory values. The Call Stack window shows the current call stack, including the `quicksort.exe!quicksort.Class1.Partition` method.

Memory 1

Address	quicksort.exe!quicksort.Class1.Main(string[]) + 0x00	Columns	Auto
0x02E015F8	55 8b ec 83 ec 28 57 56 53 89 4d fc c7	U□□□(WVSDMüÇ	
0x02E01605	45 f8 00 00 00 00 c7 45 f4 00 00 00 00	Eø....ÇEô....	
0x02E01612	c7 45 f0 00 00 00 00 33 f6 c7 45 e8 00	ÇEô....3öÇEè.	
0x02E0161F	00 00 00 c7 45 e4 00 00 00 00 c7 45 e0	...ÇEä....ÇEà	

Call Stack

Name	Lang
quicksort.exe!quicksort.Class1.Partition(System.Collections.ArrayList szArray = {Syste C#	C#
quicksort.exe!quicksort.Class1.QuickSort(System.Collections.ArrayList szArray = {Syste C#	C#
quicksort.exe!quicksort.Class1.Main(string[] args = {Length=0}) Line 40	C#

Son Söz & Sonuç

Bu yazının amacı sizlere Visual Studio ile nasıl basit bir uygulamanın yapılacağını adım adım göstermektir. Diğer .NET ve C# kaynaklarını araştırmak için sizi cesaretlendirebildiysek ne mutlu bize. Bu yazıyı buraya kadar okuduğunuza göre en azından şu aşamada çalışan bir projeniz var, istediğiniz gibi üzerinde değişiklikler yapıp sonucu görebilirsiniz.

C# ile Client/Server ve Socket programlamaya giriş

Bu makalemizde TCP protokolüyle basit bir Client/Server programı yapacağız, C# ile socket programlama yapabilmek için System.Net.Sockets isimalanı altında bulunan sınıfları kullanacağız. Yapacağımız programda server bir console uygulaması, client ise windows formlarını kullanarak yapacağımız windows uygulaması olacak. Amacımız basit bir Client/Server çatısı kurmak olduğu için uygulamamız çok basit olacaktır. Siz yazının tamamını dikkatlice incelediğinizde ve yaratıcılığınızı kullandığınızda çok daha gelişmiş uygulamalar yapabilirsiniz. Belki bir sunucu tabanlı script dili bile geliştirebilirsiniz :).

şimdi yazacağımız programda kullanıcı Windows uygulaması vasıtası ile server olan programımıza bağlanacak. Form üzerinde bulunan butona tıkladığımızda yine form üzerinde bulunan textbox girişindeki yazıyı server programımız alacak ve yazıda kaç karakter olduğunu client programına gönderecek. Client program ise bir mesaj kutusu ile kullanıcıya bildirecek. Öncelikle client olan kullanıcıdan mesajın geldiğini düşünerek Server programımızı yazalım. Server programımızı yazmaya başlamadan önce programda Scket programlama için kullandığımız sınıflara ve onların üye fonksiyonlarına kullandığımız kadarıyla bir göz atalım.

:::: TcpListener Sınıfı(System.Net.Sockets) ::::

TcpListener sınıfı TCP protokolü ile çalışan servislere bağlanmamızı sağlar. Mesela HTTP ve FTP protokolleri TCP servislerini kullanırlar. TcpListener sınıfının kurucu fonksiyonunu 3 değişik şekilde çağırabiliriz.

- 1-)** IPEndPoint sınıfını kullanarak IP numarası ve port numarası içeren bir bilgiyi kullanma yolu ile
- 2-)** IP adresi ve port numarasını geçerek çağırma
- 3-)** Sadece Port numarası ile çağırma. Bu durumda varsayılan ağ arayüzünüz TCP servislerini sağlayacaktır.

Biz bu programda 3. şıktaki gibi bir kullanımı tercih ettik.

```
public void Start();
```

TcpListener sınıfına ait bu metod network servislerinden ilgili port'u dinleyerek verileri almaya başlamamızı sağlar.

```
public Socket AcceptSocket();
```

TcpListener sınıfına ait bu metod veri transferi için geri dönüş değeri olarak bir Socket nesnesi döndürür. Bu geri dönen socket ilgili makinanın IP adresi ve port numarası ile kurulur. (kurucu işlev ile)

:::: Socket Sınıfı(System.Net.Sockets) ::::

Socket Sınıfı ile ilgili aşağıdaki örneği inceleyelim

```
Socket s = new Socket(AddressFamily.InterNetwork,  
SocketType.Stream, ProtocolType.Tcp );
```

Socket sınıfının bu kurucu işlevi parametre olarak AdressFamily dedigimiz adresleme semasi Somet tipi ve kullanacagimiz protokol tipini alır. Bu 3 parametere de .NET Framework class kütüphanesinde Enum sabitleri olarak tanımlanmıştır.

Programımızda yazdigimiz "Socket IstemciSoketi = TcpDinleyicisi.AcceptSocket();" satiri ile geri dönen soket nesnesinde bu 3 parametrede tanımlanmıştır.

```
public bool Connected();
```

Bu metod ile Soketin baglanip baglanmadigini geri dönen bool degeri ile anliyoruz. Eger socket hedef kaynaga bagliysa true degilse false degerine geri döner.

:::: NetworkStream Sinifi(System.Net.Sockets) ::::

NetworkStream sinifi kurucularından olan "void NetworkStream(Socket x);" fonksiyonu ilgili kendisine gönderilen soket nesnesine ait dataları NetworkStream türünden nesnede tutar. Bu programda kullandigimiz soket tipi stream olduğu için bu sinifi kullanıyoruz. NetworkStream sinifi içinde işlem yapabilmemeiz için ise System.IO isim alanında bulunan StreamReader ve StreamWriter siniflarını kullanacağız.

Ön bilgileri aldığımıza göre server programımızı yazalım. Aşağıdaki ilk kaynak kod server.cs dir. Satır aralarına size yardımcı olabilecek yorumlar ekledim. Kaynak dosyayı özellikle makaleme dosya olarak eklemiyorum ki siz aşağıdaki kodları tek tek yazıp daha iyi öğrenin.

:::: TcpClient Sinifi(System.Net.Sockets) ::::

Tcp servislerine bağlantı sağlamak için TcpClient sınıfı kullanılır. Istemci programımızda TcpClient sınıfının <public TcpClient(string, int);> kurucu işlevini kullanıyoruz. İlk parametre bilgisayar adı ikincisi ise port numarasıdır.

```
public NetworkStream GetStream();
```

Bu metod ile belirtilen port tan gelen veriler bir NetworkStream nesnesine aktarılır. GetStream metodunun geri dönüş değeri NetworkStream olduğu için atama işlemini NetworkStream türünden bir nesneye yapmamız gerekir.

Not: Yeşil ile yazılan satırlar yorum satırlarıdır. Html formatında bir alt satıra inmiş olan yorum satırlarını copy&paste ile programınıza aktarırken o satırları tekrar tek satır haline getirmeyi unutmayın, aksi halde programınız derlenemez.

```
//Server.cs
```

```
using System; // bunu her zaman eklememiz lazim
using System.IO ; //StreamReader ve StreamWriter siniflari için
using System.Net.Sockets; // Socket, TcpListener ve NetworkStream siniflari için

public class Server
{

    public static void Main()
    {

        //Bilgi alisverisi için bilgi almak istedigimiz port numarasini TcpListener sinifi ile
        gerçektestiriyoruz

        TcpListener TcpDinleyicisi = new TcpListener(1234);
        TcpDinleyicisi.Start();

        Console.WriteLine("Sunucu baslatildi..." ) ;

        //Soket baglantimizi yapiyoruz.Bunu TcpListener sinifinin AcceptSocket metodu ile
        yaptigimiza dikkat edin
        Socket IstemciSoketi = TcpDinleyicisi.AcceptSocket();

        // Baglantının olup olmadigini kontrol ediyoruz
        if (!IstemciSoketi.Connected)
        {
            Console.WriteLine("Sunucu baslatilamiyor..." ) ;
        }
        else
        {
            //Sonsuz döngü sayesinde AgAkimini sürekli okuyoruz
            while(true)
            {
                Console.WriteLine("Istemci baglantisi saglandi...");

                //IstemciSoketi verilerini NetworkStream sinifi türünden nesneye aktariyoruz.
                NetworkStream AgAkimi = new NetworkStream(IstemciSoketi);

                //Soketteki bilgilerle islem yapabilmek için StreamReader ve StreamWriter
                siniflarini kullaniyoruz
                StreamWriter AkimYazici = new StreamWriter(AgAkimi);
                StreamReader AkimOkuyucu = new StreamReader(AgAkimi);

                //StreamReader ile String veri tipine aktarma islemi önceden bir hata olursa bunu
                handle etmek gerek
                try
                {
                    string IstemciString = AkimOkuyucu.ReadLine();

                    Console.WriteLine("Gelen Bilgi:" + IstemciString);
```

```

//Istemciden gelen bilginin uzunlugu hesaplaniyor
int uzunluk = IstemciString.Length;

//AgAkimina, AkimYazini ile IstemciString inin uzunluğunu yazıyoruz
AkimYazici.WriteLine(uzunluk.ToString());

    AkimYazici.Flush() ;
}

catch
{
    Console.WriteLine("Sunucu kapatiliyor...");
    return ;
}
}
}

IstemciSoketi.Close();
Console.WriteLine("Sunucu Kapatiliyor...");
}
}

```

İşte buda Istemci programımız. Öncelikle şunu belirtiyimki aşağıdaki kodların çoğunu Visual C# kendiliğinden hazırladı, o yüzden size Tavsiyem Visual C# kullanmanız. Öncelikle aşağıdaki şekilde gördüğümüz form yapısını benzer bir form hazırlayın. Sonra da buton_click metodunu form_kapatma metodunu ve using ifadelerini ekleyin. Yada zamanınız çoksa aşağıdaki kodları teker teker yazın. (İsteyene kaynak kodu da gönderebilirim)



//client.cs

```

using System;
using System.Net.Sockets;
using System.IO ;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class Form1 : System.Windows.Forms.Form
{
    //Burda server da tanımladıklarımızdan farklı olarak TcpClient sınıfı ile serverdan gelen
    //bilgileri alıyoruz
    public TcpClient Istemci;
    private NetworkStream AgAkimi;
    private StreamReader AkimOkuyucu;
    private StreamWriter AkimYazici;

    private System.Windows.Forms.Button buton;

```



```

private System.Windows.Forms.TextBox textbox;

private System.ComponentModel.Container components = null;

public Form1()
{
    InitializeComponent();
}

protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        if (components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose( disposing );
}

private void InitializeComponent()
{
    //Bu satırları Visual C# oluşturdu.
    this.buton = new System.Windows.Forms.Button();
    this.textbox = new System.Windows.Forms.TextBox();
    this.SuspendLayout();

    this.buton.Location = new System.Drawing.Point(8, 40);
    this.buton.Name = "buton";
    this.buton.Size = new System.Drawing.Size(248, 23);
    this.buton.TabIndex = 0;
    this.buton.Text = "Sunucuya Baglan";
    this.buton.Click += new System.EventHandler(this.buton_Click);

    this.textbox.Location = new System.Drawing.Point(8, 8);
    this.textbox.Name = "textbox";
    this.textbox.Size = new System.Drawing.Size(248, 20);
    this.textbox.TabIndex = 1;
    this.textbox.Text = "Buraya Sunucuya göndereceğiniz yazıyı yazın";

    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    this.ClientSize = new System.Drawing.Size(264, 69);
    this.Controls.AddRange(new System.Windows.Forms.Control[] {
    this.textbox,
    this.buton});
    this.MaximizeBox = false;
    this.Name = "Form1";
    this.Text = "C#nedir?com";
    this.Closing += new
    System.ComponentModel.CancelEventHandler(this.form1_kapatma);
    this.Load += new System.EventHandler(this.Form1_Load);
    this.ResumeLayout(false);
}

```

```

//giriş noktamız olan mainde yeni bir form1 nesnesini çalıştırıyoruz
static void Main()
{
    Application.Run(new Form1());
}

//Form1 yüklendiğinde TcpClient nesnesi oluşturup AgAkımından(NetworkStream) verileri okuyoruz
private void Form1_Load(object sender, System.EventArgs e)
{
    try
    {
        Istemci = new TcpClient("localhost", 1234);
    }
    catch
    {
        Console.WriteLine("Baglanamadi");
        return;
    }
    //Server programında yaptıklarımızı burda da yapıyoruz.
    AgAkimi = Istemci.GetStream();
    AkimOkuyucu = new StreamReader(AgAkimi);
    AkimYazici = new StreamWriter(AgAkimi);
}

private void buton_Click(object sender, System.EventArgs e)
{
    //Kullanıcı butona her tıkladığında textbox'ta yazı yoksa uyarı veriyoruz
    //Sonra AkimYazici vasıtası ile AgAkımına veriyi gönderip sunucudan gelen
    //cevabı AkimOkuyucu ile alıp Mesaj la kullanıcıya gösteriyoruz
    //Tabi olası hatalara karşı, Sunucuya bağlanmada hata oluştu mesajı veriyoruz.
    try
    {
        if (textbox.Text=="")
        {
            MessageBox.Show("Lütfen bir yazi giriniz","Uyari");
            textbox.Focus();
            return ;
        }

        string yazi;
        AkimYazici.WriteLine(textbox.Text);
        AkimYazici.Flush();
        yazi = AkimOkuyucu.ReadLine();
        MessageBox.Show(yazi,"Sunucudan Mesaj var");
    }

    catch
    {
        MessageBox.Show("Sunucuya baglanmada hata oldu...");
    }
}

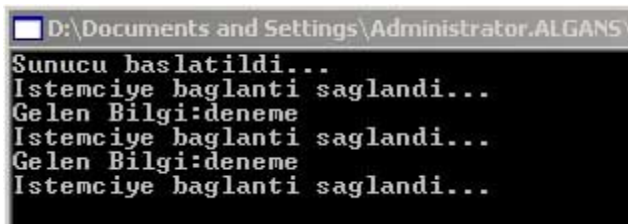
//Tve bütün oluşturduğumuz nesneleri form kapatıldığında kapatıyoruz.
public void form1_kapatma(object o , CancelEventArgs ec)

```

```
{
    try
    {
        AkimYazici.Close();
        AkimOkuyucu.Close();
        AgAkimi.Close();
    }

    catch
    {
        MessageBox.Show("Düzgün kapatilamiyor");
    }
}
```

Aşağıdaki server ve client programlarımızın aynı anda çalıştıkları sırada alınmış ekran görüntüleri mevcuttur.



Yazı hakkında sorularınızı bana sorabilirsiniz. Kaynak kodları özel bir istek geldiğinde buraya koyabilirim.

C#'ta Temel Metin Dosyası İşlemleri

Micsoft NET ile programcılarının hizmetine sunulan hazır sınıf kütüphaneleri sayesinde diğer dillerde programcılar uğraştıran birçok konu üzerinde program yazmak artık bir zevk haline geldi. Girdi/Çıktı (I/O) işlemleri de böyle zevkli hale gelen konulardan biridir. Biz bu yazımızda metin dosyası (text file) ile ilgili temel birkaç işlem üzerinde duracağız.

Metin dosyalarını oluşturmak, yazmak, içeriklerini okumak için System isim uzayında bulunan Text alt uzayındaki sınıfları kullanıyoruz. Aşağıdaki programımızda 3 tane metodumuz var. Birincisi, `DosyayaYaz()` metin dosyasını oluşturup bu dosya ya birkaç şey yazdırıyor. Bu metod önce `StreamWriter` sınıfından `dosya` isimli bir obje oluşturuyor. Daha sonra `StreamWriter` sınıfında bulunan `WriteLine()` metodu ile 2 satır yazıyoruz dosyamıza. Son olarak dosyamızı `dosya.Close()` ile kapatıyoruz.

İkinci metodumuz, `DosyadanOku()`, ise bir metin dosyasının içeriğini ekrana yazdırıyor. Bunun için önce `StreamReader` sınıfına ait `dosyaOku` nesnemizi oluşturuyoruz. Sonra dosyamızı `dosyaOku=File.OpenText(dosyaIsmi);` komutu ile açıyoruz. Dosyamızın ilk satırında bulunan yazıyı "yazi" isimli değişkenimize `yazi=dosyaOku.ReadLine();` ile aktarıyoruz. Bundan sonra ise eğer okuduğumuz satırda yazı varsa (yani dosyanın sonu değilse) o satırı ekrana yazdırıp bir sonraki satırı okuyoruz. Okuma ve ekrana yazdırma işlemlerini dosyanın sonuna kadar (yada okuduğumu satırın içeriğinin null olana kadar) devam ediyoruz. Son olarak ise `dosyaOku.Close()` ile dosyamızı kapatıyoruz.

Üçüncü ve son fonksiyonumuz ise metin dosyamızın sonuna birşeyler ekleyen `DosyayaEkle()`'dir. Yine `StreamWriter` sınıfından `dosya` ismini verdiğimiz bir nesne oluşturuyoruz. Dosyamızı `File.AppendText()` metodu ile açıyoruz ki bu metod sayesinde dosyanın sonuna istediğimiz veriyi kolayca ekleyebiliriz. `dosya.WriteLine("Bu da en son Append ile eklediğimiz satır...");` komutu ile tırnaklar arasında metni dosyamızın sonuna ekliyoruz. Her zamanki gibi açtığımız dosyayı işimiz bitince hemen `dosya.Close();` ile kapatıyoruz.

Aşağıdaki programı sisteminizde derleyip çalıştırmanızı ve hatta kod ile oynayıp değişiklikleri incelemenizde yarar olduğunu düşünüyorum. Herkese başarılar...

```
using System;  
using System.IO;  
using System.Text;
```

```
class TextFile
{
    public static void Main(string[] args)
    {
        // Metin dosyamıza birşeyler yazan fonksiyon..
        DosyayaYaz();

        // Metin dosyamızı okuyan ve ekrana yazan fonksiyon
        DosyadanOku("c:\\Deneme.txt");

        // Metin dosyamızın sonuna birşeyler ekleyen fonksiyon
        DosyayaEkle("c:\\Deneme.txt");

        Console.ReadLine();
    }

    static void DosyayaYaz()
    {
        //StreamWriter classından dosya isimli bir nesne oluşturalım
        StreamWriter dosya = new StreamWriter("c:\\Deneme.txt");

        //Dosyamıza birinci satırı yazalım
        dosya.WriteLine("Metin dosyamızın ilk satırı");

        //Buda dosyamıza yazdığımız ikinci satır
        dosya.WriteLine("İkinci satır...");

        //Dosyamızın kapatılım..
        dosya.Close();

        //Yazma işlemini başarı ile tamamladığımızı kullanıcıya bildirelim..
        Console.WriteLine("Dosya yazımı Başarı ile tamamlandı...");
    }

    static void DosyadanOku(string dosyaİsmi)
    {
        // Text dosyasından okuyan StreamReader sınıfına ait bir
        // dosyaOku nesnesini oluşturuyoruz
        StreamReader dosyaOku;

        // dosyadan okuyacağımız yazıyı string olarak depolamak için
        // yazı nesnemizi oluşturuyoruz.
        string yazi;

        //Dosyamızı okumak için açıyoruz..
        dosyaOku=File.OpenText(dosyaİsmi);

        //Dosyamızı okumak için açıyoruz ve ilk satırını okuyoruz..
        yazi=dosyaOku.ReadLine();

        /* okuduğumuz satırı ekrana bastırıp bir sonraki satıra geçiyoruz
        * Eğer sonraki satırda da yazı varsa onu da okuyup ekrana bastırıyoruz.
        */
    }
}
```

```
* Bu işlemleri dosyanın sonuna kadar devam ettiriyoruz.. */

while(yazi!=null)
{
    Console.WriteLine(yazi);
    yazi=dosyaOku.ReadLine();
}

// dosyamızı kapatıyoruz..
dosyaOku.Close();
}

static void DosyayaEkle(string dosyaIsmi)
{
    //StreamWriter classından dosya isimli bir nesne oluşturalım
    StreamWriter dosya;

    // dosyamızın sonuna birşeyler eklememek için açıyoruz..
    dosya=File.AppendText(dosyaIsmi);

    // dosyanın sonuna birşey ekliyoruz..
    dosya.WriteLine("Bu da en son Append ile eklediğimiz satır...");

    // Dosyamızı kapatıyoruz..
    dosya.Close();

    Console.WriteLine("Dosyanın sonuna başarı ile ekledik...");
}
}
```

C# Dilinde Yapılandırıcılara Aşırı Yüklenmesi

C# dilinde bulunan ve nesne yönelimli programlama kavramı içinde yer alan güzel bir özelliği yapılandırıcılara aşırı yüklenme konusunu bu yazımızda inceleyelim. Bazen bir nesneyi oluştururken bu işi birden farklı şekillerde yapmak zorunda kalırız.

Mesela elimizde bir programcı sınıfı var ve bu sınıftan oluşturduğumuz her programcı nesnesinin tüm özelliklerini oluştururken veremeyebiliriz. Programcı nesnelerimizi oluştururken bir kısım bilgileri sonradan elde etmek durumda kalabiliriz. Veya başka bir programda herhangi bir sınıfın bir örneğini oluştururken bu nesneye ait olan n tane özelliğin bir kısmına ihtiyaç duymadan girilmeyen parametrelere varsayılan değerler atayarak işimizi halletme şansımız da var.

Aşağıdaki programımızda, **Programci** sınıfımızın **yasi**, **adi**, **soyadi** ve **kullandigiDil** olmak üzere 4 tane özelliği bulunmakta. Sınıfımızın 4 tane yapılandırıcısı var. Bunların isimleri aynı (zaten yapılandırıcının ismi sınıf ismi ile aynı olur). Farklı olan ise aldıkları parametre sayıları ve tipleri olabilir.

Sırasıyla, birinci yapılandırıcı hiç bir değer almıyor. İkinci yapılandırıcımız iki tane değişken alıyor. Bunlardan **yas** ve **adi** değerleridir. Üçüncüsü ise **yas**, **adi** ve **soyadi** parametrelerini alarak nesnemizi oluşturuyor. Son yapılandırıcımız ise **yas**, **adi**, **soyadi** ve **kullandigiDil** değişkenleridir. İlk üç yapılandırıcı alınmayan **yas** değişkenine **0** diğerlerine **null** değerleri atıyor.

Sınıfımızın beşinci ve son metodu ise bu sınıftan ürettiğimiz bir nesnenin özelliklerini ekrana yazdırıyor. Eğer nesnenin bir özelliğinin değeri varsa onu yazdırıyor, yoksa bu özellik hakkında bir malumatımız yok gibisinden bir uyarı yazıyor ekrana.

Programımızın Main() fonksiyonu içinde önce 4 tane (a, b, c, ve d) programcı nesnesi oluşturuyoruz. Bunların herbirini ayrı yapılandırıcılar ile oluşturuyoruz. showOzellik() metodumuz ile bunların herbirinin özelliklerini ekrana yazdırıyoruz.

```
using System;
```

```
class OverLoadedFunctions  
{
```

```

static void Main(string[] args)
{
    Programci a = new Programci();
    Programci b = new Programci(23,"Ziya");
    Programci c = new Programci(27,"Kamuran","Kamiloğlu");
    Programci d= new Programci(30,"Hayrettin","Kütükçü","C#");

    a.showOzellikler();
    b.showOzellikler();
    c.showOzellikler();
    d.showOzellikler();

    Console.ReadLine();
}

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

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

    // İsmi ve yasini alan bir yapılandırıcı..
    public Programci(int yasi, string adi)
    {
        this.adi=adi;
        this.yasi=yasi;

        this.soyadi=null;
        this.kullandigiDil=null;
    }

    // İsmi, soyismini ve yasini alan bir yapılandırıcı..
    public Programci(int yasi, string adi, string soyadi)
    {
        this.adi=adi;
        this.yasi=yasi;
        this.soyadi=soyadi;

        this.kullandigiDil=null;
    }

    // İsmi, soyismini kullandığı dili ve yasini alan bir yapılandırıcı..
    public Programci(int yasi, string adi, string soyadi, string kullandigiDil)

```



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

public void showOzellikler()
{
    Console.WriteLine("*****");

    if(this.yasi!=0)
        Console.WriteLine("Yasi : "+this.yasi);
    else Console.WriteLine("Yasi bilgisi elimizde yok şu anda...");

    if(this.adi!=null)
        Console.WriteLine("Adı : "+this.adi);
    else Console.WriteLine("Adi bilgisi elimizde yok şu anda...");

    if(this.soyadi!=null)
        Console.WriteLine("Soyadı : "+this.soyadi);
    else Console.WriteLine("Soyadı bilgisi elimizde yok şu anda...");

    if(this.kullandigiDil!=null)
        Console.WriteLine("Kullandığı Programlama dili : "+this.kullandigiDil);
    else Console.WriteLine("Hangi Dili kullanıldığını bilmiyoruz be... :-(\n");

    Console.WriteLine("\n*****");
}
}
```

C# İsim uzayları (namespace) Hakkında

Bu yazımızda C# dilindeki namespace ler hakkında geniş bir bilgi edineceğiz. Bildiğiniz gibi programlama dillerinde, programcılar işlerini kolaylaştırmak için bir takım hazır kütüphaneler mevcuttur, bu kütüphanelerden bazıları standart olmakla birlikte bazıları programcılar tarafından sonradan geliştirilmiş ve kullanıcıların hizmetine sunulmuştur. Mesela MFC ve ATL gibi kütüphanelerin kendilerine has amaçları vardır, MFC kütüphanesi ile bir takım hazır C++ sınıflarına ulaşarak temelde zor olan bir takım Windows platformuna özgü işlemleri (forms, dialog box vs.) yapabiliriz. Bu da MFC programcılarının çalışır bir uygulama yapmak için daha az zaman harcar. Bu tür kütüphaneler Visual Basic te ve Java dilinde de vardır. Fakat bu dillerin aksine C# dili ile gelen hazır bir takım sınıf kütüphaneleri bulunmamaktadır, kısacası standart bir C# kütüphanesi mevcut değildir. Bu demek değildir C# ile işimiz daha zor olacak, aslında daha kolay, .NET Framework dediğimiz altyapının bize veya diğer programlama dillerini kullanan programcılara sunduğu bir takım temel türler ve sınıflar mevcuttur. Bütün bu sınıfları ve türleri binary düzeyde iyi organize edebilmek için .NET, namespace kavramını sıklıkla kullanmaktadır. Demekki .NET teki sınıf kütüphaneleri bir dilden bağımsız bir yapıdadır. MFC gibi sadece C++ için yada başka bir dil için geliştirilmemiştir. Çok normal olarak Visual Basic.NET kullanıcısı ile C# kullanıcısı aynı kütüphaneden faydalanırlar.

Namespace ler .NET Framework sınıf kütüphanesindeki veri türlerini ve sınıfları kullanabilmemiz için C# dilinde **using** anahtar sözcüğü ile birlikte kullanılır ve derleyiciye bildirilir. Diğer dillerde ise bu isim alanları farklı şekilde derleyiciye bildirilir, ama temelde yapılan iş .NET Framework sınıf kütüphaneslerini kullanma hakkı almaktır. Aşağıda C#, Visual Basic ve Managed C++ ile yazılmış 3 farklı ama aynı işi yapan 3 program görüyorsunuz. Programları dikkatlice incelediğinizde namespace lerin sadece eklenme biçimi ve namespace lerde ki sınıfların sentaks olarak kullanımı farklı. Bize sunduğu arayüzler ise özel durumlar dışında tamamen aynıdır.

[C#]

```
using System;  
  
public class C#nedir  
{
```

```

    public static void Main()
    {
        Console.WriteLine ("Merhaba, beni C# ile yazdılar.")
    }
}

```

[VB.NET]

Imports System

Public Module C#nedir

```

    sub Main()

        Console.WriteLine ("Merhaba, beni VB.NET ile yazdılar.")

    End Sub

```

[Managed C++]

using namespace System;

```

public static void Main()
{
    Console::WriteLine ("Merhaba, beni managed C++ ile yazdılar.")
}

```

Yukarıdaki programlarda gördüğümüz gibi .NET platformunu destekleyen bütün diller aynı sınıfı kullanarak ekrana yazı yazdırıyorlar. Bu sınıf System isim alanı içinde bulunan Console sınıfına ait bir fonksiyonla gerçekleştirilmektedir.

Namespace leri kendi yazdığımız kodların organizasyonu içinde kullanabiliriz. Hem böyle tür isimlerinin karışmasında önlemiş oluruz, zira bir tür ismi yada sınıf ancak kendi isim alanı içinde görünürlüğe (visibility) sahiptir. Mesela System isim alanını eklemeyen Console sınıfını kullanamayız. Aynı şekilde kendi yazdığımız sınıfları için de isim alanları tanımlayarak, kaynak kodumuzu istediğimiz bir şekilde organize edebiliriz. .NET Framework sınıf kütüphanesi hiyerarşik bir yapıya sahip olduğu için iç içe isim alanları tanımlanmıştır.

İsim alanlarının kullanımına bir örnek verecek olursak : Diyelimki 2D (iki boyutlu) grafikleri içeren bir sınıf kütüphanesi geliştiriyoruz, ve bu sınıf kütüphanesi içinde "Nokta" adlı bir sınıfımız var. Bu isim alanını tanımlamak için namespace anahtar sözcüğünün aşağıdaki gibi bir kullanımı vardır.

```

namespace 2DGraph
{
    public class Nokta
    {
        .....
    }
}

```

Yukarıdaki Nokta sınıfını kullanabilmek için programımıza using deyimi ile isim alanını eklememiz gerekir. Bu işlem aşağıdaki gibi yapılır.

```

using 2DGraph;

```

Nihayet 2DGraph isimli sınıf kütüphanesini oluşturduk ve başkalarının kullanımına sunduk. Bir süre sonra da 3DGraph isimli sınıfı altında 3 boyutlu grafik işlemleri yapan yeni bir sınıf kütüphanesi geliştirdik ve tekrar programcıların hizmetine sunduk. Yine aynı şekilde 3 boyutlu noktayı temsil etmek için Nokta sınıfımız olsun

```
namespace 3DGraph
{
    public class Nokta
    {
        .....
    }
}
```

Şimdi 2DGraph ve 3DGraph sınıf kütüphanelerinin her ikisini birden kullanmak isteyen bir programcı using ile isimlerini ekledikten sonra Nokta türünden bir nesne oluşturmak istediğinde derleyici bunun 2D Nokta mı yoksa 3D Nokta mı olduğunu nerden bilecek. Bunu çözenin iki yolu vardır. Birincisi veri tipi belirlerken aşağıdaki şekilde bir kullanım tercih edilir.

Veri türlerinin bu şekilde belirtilmesi pek tercih edilmeyen bir yöntemdir. Çünkü iç içe bir çok isimlerinin tanımlandığı durumlarda kaynak kodumuz gereksiz yere isimlerini yazmakla uzamaktadır. Bu hem okunabilirliği bozmakta hemde programcıya zaman kaybettirmektedir.

```
using System;
using 3DGraph;
using 2DGraph;

public class C#nedir
{
    public static void Main()
    {
        3DGraph.Nokta 3dnokta = new 3DGraph.Nokta();
        2DGraph.Nokta 2dnokta = new 2DGraph.Nokta();
    }
}
```

İkinci bir yöntem ise isim alanlarında bulunan sınıflar için takma isim (alias) kullanmaktır. Bu sayede isim alanlarını bir kez ekledikten sonra o isim alanında bulunan sınıflara doğrudan erişebiliriz. Bir isim alanındaki sınıfa takma ad aşağıdaki şekilde verilir.

```
using System;
using 3DGraph;
using 2DGraph;

public class C#nedir
{
    using Nokta2D = 2DGraph.Nokta;
    using Nokta3D = 3DGraph.Nokta;

    public static void Main()
    {
        Nokta2D 2dnokta = new Nokta2D();
    }
}
```

```
Nokta3D 3dnokta = new Nokta3d();  
}  
}
```

Yukarıda mavi yazı ile belirtilen yerlerde takma isimler tanımlanmıştır. Takma isimler ancak ve ancak tanımlanadıkları blok içinde geçerlidir. Başka bloklarda takma adları kullanmak derleme zamanında hataya yol açar.

Sonuç : İsim alanları component(program parçasığı) yazmanın en önemli parçasıdır. Bir "Merhaba Dünya" programı için isim alanı belirtmek sizde takdir edersiniz ki pek anlam taşımamaktadır. İsim alanları daha çok kodumuzun tekrar kullanılabilirliğini artırmak için geliştirilen sınıf kütüphanelerinde kullanılırlar.

C# Dilindeki Temel Veri Türleri

Her dilde olduğu gibi C# dilinde de önceden tanımlanmış ve dillerin temelini oluşturan veri saklamak için kullanılan bir takım veri tipleri vardır. Bu makalemizde C# dilinde kullanılan veri türlerine değineceğiz. C# dilinde temel olarak veri tipleri ikiye ayrılır, bunlar önceden tanımlanmış veri türleri ve kullanıcı tarafından tanımlanmış veri türleridir. Önceden tanımlanmış veri türleri de kendi arasında referans tipi(reference types) ve değer tipi(value type) olmak üzere ikiye ayrılır. Bu detaylı bilgileri vermeden önce veri tipleri nasıl tanımlanır, veri türlerine nasıl ilk değer verilir ve veri türlerinin faaliyet alanı gibi temel konulardan bahsetmek istiyorum.

Değişken Kavramı

Değişkenler bir programlama dilinde temel verileri saklamak ve bu verileri sonradan kullanmak için kullanılan bellek bölgeleridir. C# dilinde genel olarak bir değişken tanımlaması aşağıdaki gibi olmaktadır.

```
Veritipi veriadi ;
```

Örneğin C# dilinde işaretli ve 32 bitlik veriyi temsil eden "a" isimli bir değişken aşağıdaki gibi tanımlanır.

```
int a ;
```

Fakat yukarıdaki tanımlamada bir sorun var. "a" adlı değişkende herhangi bir değer tutulmaktadır. Bu yüzden C# derleyicisi şimdilik "a" değişkenini kullanmamıza müsaade etmez, çünkü "a" da neyin olduğu henüz belli değildir. Bu yüzden değişkenlere =(eşittir) operatörüyle ilk değerler atarız, ya da değişken tanımlamasından sonra, değişkene bir değer atarız. Bir değişkene bir değer atamak için iki farklı yöntem kullanılır. Aşağıda bu iki yönteme ait örnek bulunmaktadır.

```
int a = 10 ; //değişken tanımlanırken bellekteki değer 10 olarak düzenleniyor.
```

```
-----  
int b;  
b = 10 ; /*değişken tanımlandıktan sonra değişkene değer atanıyor.İşlevsel olarak bu  
iki kullanım açısından bir fark yoktur.*/  
-----
```

```
int a=10, b;  
b = 10 ; /*eğer bir satırda birden fazla değişken tanımlaması yapmak istiyorsak bu  
yapıyı kullanırız.Bu durumda a ve b int türden değişkenlerdir denir.*/  
-----
```

Önemli Not: C# dilinde bir değişkene herhangi bir değer atamadan onu kullanmak yasaktır. Yani derleme işlemi gerçekleşmez, örneğin aşağıdaki gibi bir kullanım derleme zamanında hata verecektir. Bu yüzden eğer bir değişkeni kullanmak istiyorsak yukarıda açıkladığımız gibi değişkenlere bir değer vermek zorundayız. Bu kural önceden tanımlanmış referans tipleri için de geçerlidir.

```
int a ;  
Console.WriteLine(a); //Bu ifadeleri içeren bir kod derlenemez.
```

Değişkenlerin Faaliyet Alanları (Scope)

C# dilinde programın genel akışı açılan ve kapanan parantezler içerisinde yazılır. Bu açılan ve kapanan parantezler arasındaki bölgeye blok denir. Tanımlanan bir değişkene, ancak tanımlandığı blok içerisinde ulaşılabilir. Örneğin aşağıdaki kısa örnekte tanımlanan örnekte her iki "a" değişkeni birbirinden bağımsızdır ve bellekte ayrı bölgelerde saklanırlar.

```
public class deneme  
{  
    public static void Main()  
    {  
        { //Birinci blok  
            int a=20 ;  
        }  
        { //İkinci blok  
            int a=20 ;  
        }  
    }  
}
```

Yukarıdaki örnekte birinci ve ikinci blokta tanımlanan "a" isimli değişkenler Main bloğu içinde geçersizdir. Birinci a değişkeninin faaliyet alanı 1.Blok ,ikinci a değişkeninin faaliyet alanı ise 2. Bloktur. Bu durumda Main() bloğunda Console.WriteLine(a); gibi bir ifade hatalıdır, çünkü Main bloğu içinde tanımlanan bir a değişkeni yoktur. Unutmamalıyız ki daha sonraki makalelerde detaylı olarak göreceğimiz for ve diğer döngüler de birer blok olduğu için bu bloklarda tanımlanan değişkenler döngü bloğunun dışında geçersiz olacaktır. Diğer bir önemli nokta ise faaliyet alanı devam eden bir değişkenin bir daha tanımlanmasının hataya yol açmasıdır. Örneğin aşağıdaki gibi bir durum derleme zamanında hata verecektir. Çünkü bir değişkenin faaliyet alanı bitmeden aynı isimli değişken tekrar tanımlanıyor.

```
public class C#nedir?com
{
public static void Main()
{
int a;
{
int a=20 ;
}
}
}
```

Gördüğünüz gibi Main bloğunda tanımlanan a değişkeninin faaliyet alanı açılan blokta devam etmektedir.Bu yüzden yukarıdaki gibi ifadeler geçersizdir.Üst seviyede açılan bloklar alt seviyedeki blokları kapsadığı için, birinci tanımlanan a değişkeni sonradan açılan blok içinde hala geçerlidir.

Yukarıda anlatılan duruma ters düşüyor gibi görünse de aşağıdaki gibi bir kullanım son derece legal bir durumdur. Bu konuyu daha sonraki makalelerimizde detaylı bir şekilde inceleyeceğiz.

```
public class C#nedir?com
{
    static int a = 10;
    public static void Main()
    {
        int a;
        {
            int a = 10 ;
        }
    }
}
```

Bu konu sınıflarla ilgili bir konu olduğu için detaylarına girmeyeceğiz ama şimdilik böyle bir kullanımın geçerli olduğunu bilmenizde fayda var.

Sabitler

Bir program boyunca değerinin değişmeyeceğini düşündüğümüz verileri sabit veriler olarak tanımlarız. Sabit veriler tanımlamak için tanımlama satırında **const** anahtar sözcüğünü kullanırız. **const** olarak tanımlanmış değişkenlerin en büyük avantajı program içinde sıkça kullandığımız değerleri aniden değiştirmek gerektiğinde görülür.Mesela matematiksel işlemler yapan bir programda pi sayısını const olarak tanımlayıp istediğimiz zaman pi sayısını değiştirebiliriz. Tabi bu işlemi const değilde normal bir değişkenle de yapabildik, ama şu da bir gerçek ki çok uzun programlarda sabit olmasını istediğimiz değişkeni yanlışlıkla değiştirebiliriz. Fakat const olarak tanımladığımız bir değişkenin değerini değiştirmeye çalıştığımızda c# derleyicisi derleme aşamasında hata verecektir. Bu da gözden kaçan bazı hata durumlarını minimuma indirmek demektir. Sabit ifadeleriyle ilgili bilmemiz gereken 3 önemli kural vardır. Bunlar şunlardır :

- 1-) Sabitler tanımlandıklarında değerleri atanmalıdır. İlk değer verilmeyen değişkenler const yani sabit olamazlar.
- 2-) Sabit ifadelere ancak sabit ifadelerle ilk değer atanabilir yani şu şekilde bir kullanım hatalıdır. **const int = a + b ;**
- 3-) Sabit ifadeleri içsel tasarım olarak zaten statik oldukları için, ayrıca statik olarak belirtmek hatalıdır ve kullanılamaz.(statik değişkenler ileriki yazılarda detaylı olarak anlatılacaktır.)

Basit bir sabit tanımlaması aşağıdaki gibi yapılmaktadır.

```
const double pi = 3.14 ; // double, kesirli sayıları tutmak için tanımlanmış bir veri türüdür.
```

Değer(value) ve referans(reference) tipleri

C# dilinde önceden tanımlanmış(c# dilinde varolan tipler) veri tipleri değer tipleri ve referans tipleri olmak üzere ikiye ayrılır. Bu iki veri tipi arasındaki farkı çok iyi kavramak gerekir. Daha önce dediğimiz gibi değişkenler bellekte bulunan verilerdir. Aslında bir değişkeni kullanırken o değişkenin bellekte bulunduğu adresteki veriye ulaşıyoruz. Değer tipleri değişkenin değerini direkt bellek bölgesinden alırlar. Referans tipleri ise başka bir nesneye referans olarak kullanılırlar. Yani referans tipleri aslında bir çeşit bellek bölgesi olan heap alanında yaratılan veri türlerinin (bunlara kısaca nesne de diyebiliriz) adreslerini saklarlar. Değer tipleri yaratıldıklarında stack dediğimiz bellek bölgelerinde oluşturulurlar, referans tipleri ise kullanımı biraz daha sınırlı olan heap dediğimiz bellek bölgesinde saklanırlar. C ve C++ dillerine aşina olan arkadaşların da tahmin ettiği gibi gösterici kavramı ile referans veri tipleri arasında çok fazla fark yoktur. Fakat C# dilinde kullanıcının direkt olarak kullanabileceği bir gösterici veri türü tanımlamak yoktur. Bunun yerine bazı değişkenler değer tip bazıları ise referans tipi olarak işlem görürler. Peki bunlar nelerdir? Temel veri tipleri olan int,double, float ve yapı nesneleri gibi veri türleri değer tipler, herhangi bir sınıf türü ise referans türüdür. İki değer tipi nesnesini birbirine eşitletken değişkenlerde saklanan değerler kopyalanarak eşitlenir ve bu durumda iki yeni bağımsız nesne elde edilmiş olur yani birinin değerini değiştirmek diğerini etkilemez, ancak iki referans tipini birbirlerine eşitlediğimizde bu nesnelerde tutulan veriler kopyalanmaz, işlem yapılan nesnelerin heap bölgesindeki adresleridir, yani iki nesne de aslında heap bellek bölgesinde aynı adresi gösterecekleri için birinde yapılan değişiklik diğerini de etkileyecektir. Referans tiplerini tanımlarken herhangi bir adresi göstermediğini belirtmek için null değere atanırlar.(Mesela: `y = null ;`)

CTS (Common Type System) Tipleri

.NET bir yazılım geliştirme platformudur. Aslında bütün veri tipleri CTS dediğimiz bir sistem ile tanınırlar. Yani C# dilinde ki veri türleri aslında CTS 'deki veri türleri için birer arayüz gibidirler. CTS sayesinde .NET platformu için geliştirilen bütün diller aynı veri tiplerini kullanırlar, tek değişen veri türünü tanımlama yöntemi ve sentaksıdır. Bu yüzden bizim C# dili ile tanımlayacağımız her veri tipinin CTS 'de bir karşılığı mevcuttur. Bu veri türleri ve CTS karşılıkları aşağıda tablolar halinde mevcuttur.

C# dilinde tanımladığımız bütün basit veri tipleri aslında CTS 'de bulunan bir yapı nesnesidir.C# dilindeki önceden tanımlanmış temel veri tipleri on beş tanedir. Bunlardan on üçü değer tipi ikisi de değer tipidir.

Önceden Tanımlanmış Value Veri Tipleri

Aşağıda temel value tiplerin C# dilindeki adı, CTS karşılığı, açıklaması ve kullanım aralığı bulunmaktadır.

C# taki adı	CTS Karşılığı	Açıklama	Max ve Min aralık yada değeri
sbyte	System.Byte	8 bit işaretli tamsayı	-128 : 127
short	System.Int16	16 bit işaretli	-32.768 : 32.767

		tamsayı	
int	System.Int32	32 bit işaretli tamsayı	-2.147.483.648 : 2.147.483.647
long	System.Int64	64 bit işaretli tamsayı	-9.223.372.036.854.775.808 : -9.223.372.036.854.775.807
byte	System.Byte	8 bit işaretli tamsayı	0 : 255
ushort	System.UInt16	16 bit işaretli tamsayı	0 : 65.535
uint	System.UInt32	32 bit işaretli tamsayı	0 : 4.294.967.295
ulong	System.UInt64	64 bit işaretli tamsayı	0 : 18.446.744.073.709.551.615
float	System.Single	32 bit tek kayan sayı	+yada - $1,5 \cdot 10^{-45}$: + ya da - $3,4 \cdot 10^{38}$
double	System.Double	64 bit çift kayan sayı	+yada - $5 \cdot 10^{-324}$: + ya da - $1,7 \cdot 10^{308}$
decimal	System.Decimal	128 bit ondalıklı sayı	+yada - $1,5 \cdot 10^{-28}$: + ya da - $7,9 \cdot 10^{28}$
bool	System.Boolean		true ya da false
char	System.Char	Karakterleri temsil eder	16 Unicode karakterleri

Şimdi tabloda verilen veri türleri ile ilgili tanımlamalara örnekler verelim :

```

long a = 0xEF20 ; // 0x öneki sayıları hexadecimal olarak yazmamızı sağlar.
ulong ul = 5698UL ; // Sayının sonuna UL koyarak UnsignedLong olduğunu belirtiyoruz.

float fl = 3.14f ;

decimal d = 65.25M;

bool b = false ;

char ch1 = 'a' , ch2 = '\\', '\"', 'm' ;

```

Önceden Tanımlanmış Reference Veri Tipleri

C# dilinde önceden tanımlanmış iki tane referans tipi vardır. Bunlar string ve object türleridir. Object türü C# dilinde bütün türlerin türediği bir sınıf yapısıdır. Kullanıcı tarafından sonradan tanımlanacak bütün veri tipleri de aslında Object türünden türemiş olacaktır. Bu da object türünden bir nesneye herhangi bir veri türünden nesneyi atayabileceğimiz anlamına gelir. Çünkü C# dilinde bütün nesneler bir object'dir. object 'ler özelleştirilerek farklı amaçlar için kullanılır. Herhangi bir nesneyi object türü ile eşleştirme kavramı boxing olarak adlandırılır. Boxing ve bunun tersi işlemi olan unboxing kavramlarını daha sonraki makalelerimizde detaylı olarak inceleyeceğiz.

Diğer bir referans tipi ise string türüdür. C ve C++ gibi dillerde string işlemleri yapabilmek için karakter dizileri tanımlanır ve bunlar string olarak işleme alınırlar ancak C# dilinde karakter dizileri tanımlamak yerine string adı ile yeni bir türü mevcuttur. String veri türü birtakım yararlı işler daha kolay bir şekilde yapılmaktadır. Mesela aşağıda iki string' in + operatörüyle arka arkaya nasıl eklendiği gösterilmektedir. + operatörü

burada string sınıfı için yüklenmiştir(overload). Overloading kavramı başlı başına bir makale konusu olduğu için burada değinmeyeceğim.

```
string s1 = "Hello " ;  
string s2 = ".NET" ;  
string s3 = s1 + s2;
```

Bir dilin sentaksı açısından özel anlamlar ifade eden karakterleri kullanmak istiyorsak bunları \ (escape) ifadesiyle belirtmek gerekir. Mesela bir dizin bilgisini içeren bir string nesnesini aşağıdaki gibi tanımlarız.

string yol = "C:\\docs\\xxx\\" ;// Bu tür kullanıma escape sequence kullanımı denir.

Escape sequence 'leri kullanmak yerine string içinde görünen ifadenin aynısı belirtmek için string ifadesinin önüne @ işareti kullanılır.Mesela ;

string esc = @"C:\docs\xxx\" // böyle bir kullanımda escape karakterini kullanmayı kaldırmış oluyoruz.

C# taki adı	CTS Karşılığı	Açıklama
object	System.Object	Bütün veri türlerinin türediği kök eleman
string	System.String	Unicode karakterlerinden oluşan string

Yukarıda C# dilindeki temel referans veri türleri tablo halinde gösterilmiştir.

Bir sonraki makalemizde C# temel kontrol yapılarını göreceğiz.

C# ile Windows Registry İşlemleri(Microsoft.Win32)

Hemen hemen her profesyonel uygulamada gördüğümüz Registry'ye yazma ve oradan okuma işlemlerinin nasıl yapıldığını basit bir uygulama ile anlatacağız. Düşününki bir uygulama geliştirdik ve uygulama her çalıştığında kullanıcıyı selamlamak istiyoruz ve uygulamayı kaçınıcı defa çalıştırdığını söylemek istiyoruz ona. Bunun bir çok yolu olmasına rağmen en güzel ve en güvenilir yolu ilgili bilgileri Windows un registry dediğimiz bölgesinde tutmaktır. registry dediğimiz yerler olmasaydı pek ala bu işi dosyaya yazma ve okumayla da yapabilirdik. Registry bölgesini okuma ve yazma amaçlı .NET framework sınıf kütüphanelerinden faydalanacağız. Bu sınıflar Microsoft.Win32 isim alanının altında bulunmaktadır. Bu sınıfların en çok kullanılan metodlarını ve özelliklerini anlatmaya başlamadan önce programızın yapısını kısaca anlatayım.

Bir console uygulaması oluşturacağız. Program ilk çalıştığında bize bundan sonraki açılışlarında bizi selamlaması için adımızı soracak. Daha sonra programı çalıştırdığımızda "Hoşgeldin Sefer. Programı 3. defa çalıştırıyorsunuz." diyecek. Programın kaç defa çalıştığını anlamak için ise program ilk açıldığında registry bölgesine "1" değerini yazacağız ve programın her çalıştığında o değeri bir artıracacağız. Böylece programın kaç defa çalıştığını öğrenmiş olacağız. Tabi eğer Windows un <regedit> aracıyla daha önceden uğraştıysanız bizim programlama yoluyla değiştirdiğimiz değerleri kendi ellerinizle gidip değiştirebilirsiniz. Demek istediğim burda şifre ve kullanıcı adı gibi bazı kişiye özel bilgilerin saklanması pek güvenli değildir.

Eğer şu ana kadar registry hakkında bir bilginiz yoksa Start->Run ' menüsüne gelip regedit yazarak registry hakkında biraz bilgi edinebilirsiniz. Bu programla rastgele değerler silerseniz bazı programlarınız zarar görebileceği için tavsiyem her hangi bir silme işlemi yapmayan ve sadece neler olup bittiğine bakın.

Şimdi C# ın büyük bir kolaylık sağladığı registry yazma ve okuma için geliştirilmiş RegistryKey sınıfının işlevlerini görelim.

:::: RegistryKey Sınıfı(Microsoft.Win32) ::::

Bildiğiniz gibi windowsun register yapısı ağaç şeklindeki klasörlere benzer. Her yeni anahtar altında bir alt anahtar açabildiğimiz gibi anahtarlar altında yeni "string" yada "int" gibi değerler oluşturup programla ilgili istediğimiz değerleri saklayabiliriz. Bu ise

klasörlerde oluşturduğumuz dosyalara benzer. Daha öncede dediğimiz gibi buraya regedit le kolayca ulaşabildiğimiz için güvenlik amaçlı bilgileri (şifre vs) veya programımızla ilgili kritik bilgileri(serial number vs) burada saklamamamız gerekir. Biz bu programdaki bilgilerimizi HKEY_LOCAL_MACHINE\Software altında csnedir isimli bir alt anahtar oluşturarak kaydedeceğiz.

RegistryKey sınıfı türünden bir nesne oluşturmak için ya RegistryKey sınıfının static üye fonksiyonu olan OpenSubKey() metodunu yada yada Register sınıfının static üyelerini kullanırız. Aşağıda detaylı olarak bu metodlar hakkında bilgi bulabilirsiniz.

:: CreateSubKey() Metodu ::

Geriye RegistryKey türünden bir nesne dödüren bu fonksiyon yeni bir alt anahtar oluşturur yada var olan bir anahtarı okumak için açar.Fonksiyonun prototipi aşağıdaki gibidir. Unutmayın bu metodu kullanabilmek için ilgili kullanıcının register bölgesine erişim hakkının olması gerekir. Aksi halde SecurityException hatası oluşur.

```
public RegistryKey CreateSubKey(string subkey);
```

:: OpenSubKey() Metodu ::

Bu metod iki şekilde kullanılabilir, overload edilmiş iki metod aşağıdaki gibidir.

```
public RegistryKey OpenSubKey(string);//Bu metod anahtar okumak amacıyla kullanılır ve geriye RegistryKey döndürür.
```

```
public RegistryKey OpenSubKey(string,bool);//Bu metod ilk metod ile aynıdır fakat eğer açılacak anahtara yazmada yapacaksak ikinci parametreyi true olarak girmemiz gerekir. Varsayalım olarak ReadOnly açılır.
```

:: DeleteSubKey() Metodu ::

Bu metod iki şekilde kullanılabilir, overload edilmiş iki metod aşağıdaki gibidir.

```
public void DeletSubKey(string);//Parametre olarak gönderilen alt anahtarı siler.
```

```
public void DeletSubKey(string,bool);//Parametre olarak gönderilen alt anahtarı siler.İkinci parametre ise belirtilen alt anahtarın olmaması durumunda "ArgumentNullException" hatasının yakalanıp yakalanmayacağını gösterir.Eğer true ise bu hata yakalanır, false ise herhangi birşey olmaz.
```

:: DeleteSubKeyTree() Metodu ::

Bu metod iki belirtilen anahtardaki bütün anahtarları siler.Bir dosyayı sildiğininide içindeki tüm dosyaları sildiğiniz gibi.Prototipi aşağıdaki gibidir.

```
public void DeletSubKeyTree(string);
```

:: DeleteValue() Metodu ::

İki şekilde kullanılabilir.Parametre olarak belirtilen değeri anahtardan siler.İkinci parametre ise DeleteSubKey() metodunda olduğu gibi hata yakalanıp yakalanmayacağını belirtir.

:: Flush() Metodu ::

Registry 'de yaptığımız değişiklikleri diske kaydetmek için bu metodun çağrılması gerekir.

:: GetSubKeyNames() Metodu ::

Be metod geriye döndürdüğü string dizisine ilgili anahtardaki alt anahtar isimlerini doldurur.Prototipi aşağıdaki gibidir.

```
public string[] GetSubKeyNames()
```

:: GetValue() Metodu ::

İlgili anahtardaki değerin içeriğini object türü olarak geri dönderir.İki şekilde kullanılabilir. Parametrik yapısı aşağıdaki gibidir.

```
public object GetValue(string)
public object GetValue(string,object) //eğer değer yoksa varsayılan olarak parametre
olarak verilen object geriye döner.
```

:: GetValueNames() Metodu ::

İlgili anahtardaki bütün değerleri bir string dizine aktarır.Parametrik yapısı aşağıdaki gibidir

```
public string[] GetValueNames()
```

:: SetValue() Metodu ::

Birinci parametresi ile belirtilen anahtara ikinci parametresi ile belirtilen bilgi aktarılır.Parametrik yapısı aşağıdaki gibidir.

```
public void SetValue(string,object)
```

:: Name Özelliği ::

Taban anahtardan itibaren(mesela HKEY_LOCAL_MACHINE) ilgili anahtarın tam yolunu verir.

:: ValueCount Özelliği ::

Anahtarda bulunan değerlerin sayısını verir.

RegistryKey sınıfının üye elmanları ve oluşturduğu exception sınıfları ile ilgili detaylı bilgiyi MSDN Online' dan yada .NET Framework SDK Documentation ' dan edinebilirsiniz.

Şimdi yazımızın başında bahsettiğimiz örnek uygulamamıza göz atalım.Aşağıda bulunan kaynak kodda satır aralarına size yardımcı olacak yorumlar ekledim.

```
//registry.cs
```

```
using System;
using System.Win32
//RegistryKey sınıfını kaynakkodda direkt kullanabilmek için bu isimalanını ekledik.
```

```
class CsReg
{
    public static void Main()
    {
        RegistryKey register;
        register = Registry.LocalMachine.OpenSubKey(@"Software\csnedir",true);
```

//HKEY_LOCAL_MACHINE/Software/csnedir anahtarını oluşturup anahtara yazma modunda açıyoruz.

```
if (register.GetValue("ad") == null)
{
    /*Bu if bloğunda programın ilk defa çalışması durumu ile ilgili işlemler
    yapılıyor.Programın ilk defa çalıştığını register.GetValue("ad") ==null ifadesi ile anlıyoruz.
    Kullanıcıdan isim alınıp registry de "ad" isimli anahtara yazılıyor ve tabili "Oturum"
    adında programı bir defa çalıştırdığını belirten 1 değeri yazılıyor*/
    Console.WriteLine("Lütfen adınızı yazınız");
    string ad = Console.ReadLine();
    register.SetValue("ad",(string)ad);
    register.SetValue("oturum",1);
    Console.WriteLine("Tesekkürler...");
}
else
{
    /*Bu blokta ise programın sonraki çalışmaları ile ilgili işlemler yapılıyor. Oturum sayısı
    registry den okunup aritmetik işlem yapabilmek için ilgili formata dönüştürüldükten sonra
    tekrar yeni değeri ile registry ye yazılıyor.Aynı şekilde registry den "ad" değeri alınarak
    kullanıcı selamlanıyor.*/
    string ad = (string)register.GetValue("ad");

    int oturum_sayisi=Convert.ToInt32(register.GetValue("oturum"))+ 1;
    register.SetValue("oturum",oturum_sayisi);
    Console.WriteLine("Hosgeldin " + ad);
    Console.WriteLine("Programı " + oturum_sayisi + " kez açtınız");
    Registry.LocalMachine.Flush();
}
}
```

Assembly, ILDASM.exe ve GACUTIL.exe Hakkında

Bu makalede kavram olarak en çok karıştırılan ve anlaşılması diğer konulara göre zor olan Assembly kavramını ve Visual Studio ile birlikte gelen GACUTIL ve ILDASM gibi önemli araçları inceleyeceğiz.

Assembly Nedir?

Hemen ilk başta belirtelim ki bu makalede bahsedeceğimiz Assembly'nin alt seviye bir programlama dili olan Assembly ile yakından uzaktan hiçbir alakası yoktur. Sadece bir isim benzerliği vardır. .NET platformunda yazdığımız kodlar sonucunda oluşturduğumuz bütün .exe uzantılı dosyalara ve .dll uzantılı dosyalara genel olarak Assembly denilmektedir. Projemize ait derlenmiş kodlar ve metadata dediğimiz bir takım özniteleyici kodlar Assembly'ler içerisinde bulunur. Assembly'lerin kabaca özellikleri aşağıdaki gibi sıralanabilir.

1-) Assembly'lerde metadata denilen veriler, Assembly'deki tür bilgileri ve başka kaynaklarla olan bağlantılar saklanır.

2-) Assembly'de(dll yada exe) kendilerine ait versiyon bilgisi tutulur. Hatırlarsanız klasik dll ve exe tipi dosyalarda versiyon bilgisi saklanmadığı için çeşitli uyumsuzluklar yaşanabilmekteydi. Mesela farklı iki firmanın hazırladığı dll 'ler aynı isimli olduğunda sonradan register edilen dll halihazırda bulunan dll 'in üzerinde yazıldığı için sistemde bulunan bazı uygulamalarda sorun çıkıyordu. Dll 'ler bu tür sorunlara yol açtığı için DLL Hell (Dll cehennemi) kavramı ortaya çıkmıştı. Aslında COM dll 'leri ile bu sorun bir nebze ortadan kalkmışsa da asıl çözüm Assembly'ler ile gelmiştir.

3-) Assembly'lerde versiyon bilgisi saklandığı için bir uygulama içerisinde farklı versiyonlara sahip Assembly'leri kullanabiliriz.

4-) Program kurma işlemi, Assembly 'ye ilişkin dosyayı direkt kopyalayarak yapılabilir. Eski sistemde DLL 'lerin register edilmesi gerekiyordu.

Assembly'lerin en önemli özelliklerinden birisi de **Application Domain** dediğimiz kavramdır. Application Domain sayesinde bir proses içinde birden fazla birbirinden bağımsız çalışan Assembly 'yi çalıştırma imkanına kavuşuruz. Bu konuya açıklayıcı bir örnek olması açısından aşağıdaki örneği inceleyebilirsiniz.

Bu örnekte iki tane Console uygulaması yapacağız. Bu iki uygulamaya ait çalışır durumdaki dosyalara artık Assembly diyebilirsiniz. Aşağıdaki ilk örnekte Main işlevi içerisinde ekrana bir yazı yazdırıyoruz.

```
//assembly1.cs

using System;
namespace assembly1
{
    class csnedir1
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Beni disardan yüklediler.");
        }
    }
}
```

Bu programı notepad 'da yazıp derledikten sonra komut satırında "csc assembly1.cs" yazıp assmby1.exe dosyasını oluşturun. Aynı klasör içine şimdi aşağıdaki assembly2.cs dosyasını oluşturun ve derleyerek assembly2.exe 'nin oluşmasını sağlayın.

```
//assembly2.cs

using System;
namespace assembly2
{
    class csnedir2
    {
        static void Main(string[] args)
        {
            AppDomain apd2 = AppDomain.CreateDomain("Csnedir");
            apd2.ExecuteAssembly("assembly1.exe");
        }
    }
}
```

Yukarıda da bahsettiğim gibi Application Domain sayesinde bir uygulama içerisine değişik assembly'ler yüklenebilir ve çalıştırılabilir. Application Domian kavramını System isim alanında bulunan AppDomain sınıfı temsil eder. Yeni bir Application Domain'i oluşturmak için AppDomain sınıfının overload (aşırı yüklenmiş) edilmiş CreateDomain statik üye fonksiyonları kullanılır. AppDomain sınıfının ExecuteAssembly fonksiyonuyla dışarıdan yeni bir Assembly yüklenir ve o satırdan itibaren yüklenen assembly çalıştırılır. (Yukarıdaki örnekte iki Assembly'nin de aynı klasörde olmasına dikkat edin.)

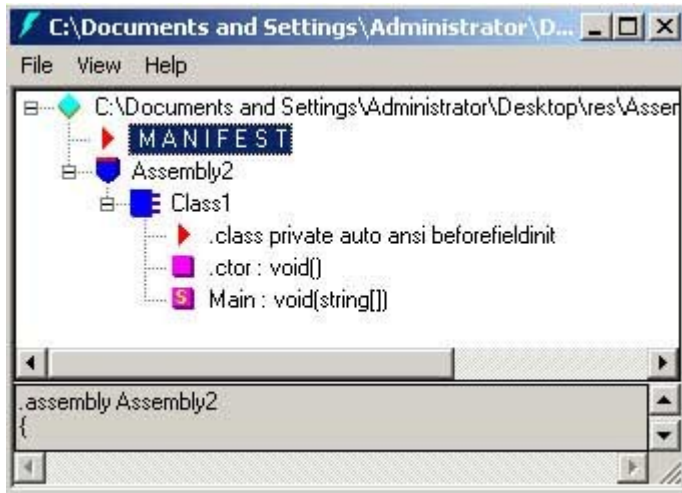
Assembly hakkında bu geniş giriş bilgisini verdikten sonra Assembly'nin fiziksel yapısından bahsedelim biraz. Bir Assembly belgesinde Assembly metadata, tür(type) metadata, kaynaklar ve IL(Intermediate Language) kodu bulunur. Bütün bu yapılar bir Assembly dosyasında bulunabileceği gibi Assembly metadata'lar sayesinde dışarıdaki kaynaklara referans da verilebilir. Assembly'lerin en önemli yapısı Manifest dediğimiz parçasıdır. Manifest assembly metadata'lar bulunur. Peki nedir bu Assembly metadata'lar? Bir Asembly adı, versiyonu gibi kimlik bilgileri, ilgili Assembly ile ilgili olan diğer dosyalar, başka Assembly'lere olan referanslar gibi bilgilerin tamamına Assembly metadata denir. İşte bu bilgilerin oluşturduğu kümeye **Manifest** denilmektedir.

Assembly'lerin Manifest bölümünde bulunan elemanlar temel olarak aşağıdaki tabloda verilmiştir.

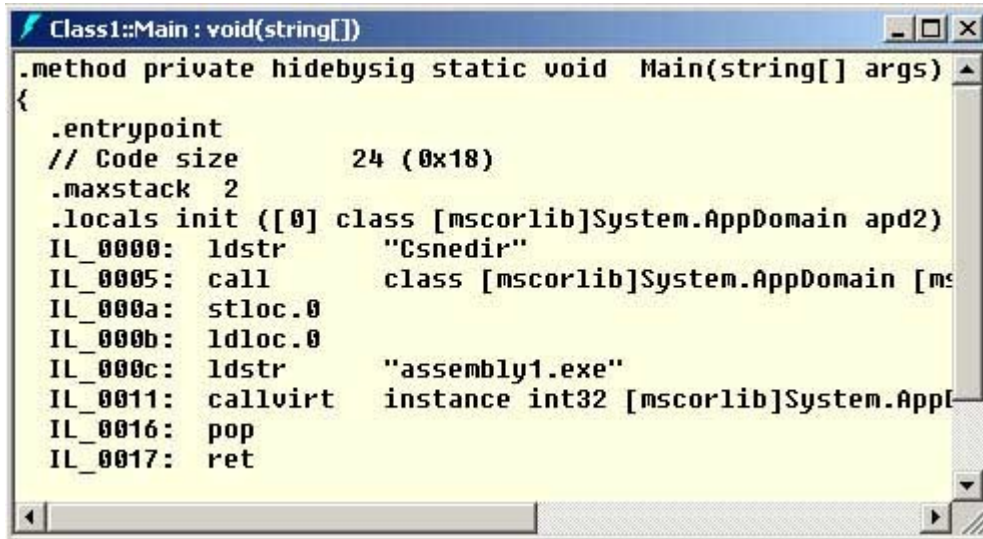
Özellik	Anlamı
AssemblyCompany	Assembly'nin firma bilgisi
AssemblyCopyright	Copyright bilgisi
AssemblyCulture	İlgili Assembly'ye ait kültür bilgisi(Almanya,İngiltere,Türkiye vs..)
AssemblyDelaySign	Gecikmeli imzanın olup olmayacağı (True ya da False)
AssemblyDescription	Assembly ile ilgili kısa açıklama
AssemblyFileVersion	Win32 sistemindeki dosya versiyonu
AssemblyInformationalVersion	CLR tarafından kullanılmayan ve okunabilirliği yüksek olan versiyon bilgisi
AssemblyKeyFile	Assembly'nin kayıt edilmesi için gereken anahtarın bulunduğu dosya
AssemblyKeyName	Kayıt için gereken anahtar sözcük
AssemblyProduct	Ürün adı
AssemblyTitle	Assembly'nin adı
AssemblyTrademark	Trademark bilgisi
AssemblyVersion	String şeklindeki Version numarası

Bu tablo MSDN kitaplığından alınmıştır

Assembly'ler private ve shared olmak üzere ikiye ayrılır. Bunları detaylı olarak açıklamaya başlamadan önce Assembly'leri görüntülemek için kullanılan ILDASM.exe aracını inceleyelim. ILDASM.exe MSIL kodu içeren assembly dosyalarını okur ve kullanışlı bir arayüz ile kullanıcıya sunar. ILDASM.exe programını çalıştırmak için komut satırına ILDASM.exe yazmamız yeterlidir. Açılacak pencereden File->Open menüsünü kullanarak görüntülemek istediğiniz Assembly dosyasını seçin.(Aşağıda gördüğümüz ekran görüntüleri Assembly2.exe'nin görüntüleridir.) Bir assembly'deki türler, sınıflar, fonksiyonlar çeşitli sembollerle temsil edilmiştir. Hangi sembollerin ne anlama geldiğini MSDN kitaplığından bulabilirsiniz.



Şimdi de açılacak pencereden Main bölümüne tıklayın, ve aşağıdaki görüntüyü elde edin. Buradaki bütün kodlar IL kodudur.



```
.method private hidebysig static void Main(string[] args)
{
    .entrypoint
    // Code size      24 (0x18)
    .maxstack 2
    .locals init ([0] class [mscorlib]System.AppDomain apd2)
    IL_0000: ldstr      "CsneDir"
    IL_0005: call       class [mscorlib]System.AppDomain [mscorlib]System.AppDomain::get_AppDomain()
    IL_000a: stloc.0
    IL_000b: ldloc.0
    IL_000c: ldstr      "assembly1.exe"
    IL_0011: callvirt   instance int32 [mscorlib]System.AppDomain::Load(byte[])
    IL_0016: pop
    IL_0017: ret
}
```

Diğer bölümlerde tıklayarak IL kodlarını inceleyebilirsiniz.

Yukarıda bahsettiğimiz gibi Assembly'ler private ve shared olmak üzere ikiye ayrılır. Normal olarak geliştirilen Assembly'ler private Assembly olarak adlandırılır. Bu tür assembly'ler uygulama ile aynı dizinde ya da alt dizinlerde bulunur. Versiyon ve isim uyumsuzluğu sorunu bu tür assembly'lerde olmamaktadır. Shared assembly'ler ise daha çok büyük projelerde mesela bir projenin bir firmaya ait farklı ofislerinde gerçekleştirildiği durumlarda kullanılır. Bu durumda assembly'lerin uyması gereken bazı kurallar vardır. Bunlardan en önemlisi strong name dediğimiz tekil bir isme sahip olmasıdır. Bu sayede shared assembly'ler tekil olduğu için bir sistemde global düzeyde işlem görürler. Yani farklı farklı uygulamalardan aynı anda shared assembly'lere ulaşılabilir. Şimdi shared assembly kavramını biraz açalım.

Shared Assembly

Assembly'ler varsayılan olarak private'tır. Bu yüzden bu tür Assembly'ler sadece bulundukları dizin içerisindeki uygulamalar tarafından görülür ve çalıştırılırlar. Oysa ki bazı durumlarda bütün programlar tarafından o Assembly'ye birbirlerinden bağımsız olarak erişilmesini isteriz. Bunu sağlamak için **Global Assembly Cache** denilen bir sistemden faydalanacağız. .NET 'in yüklü olduğu bütün sistemlerde Assembly Cache mekanizması vardır. Bu Assembly Cache'ye yüklü olan Assembly'ler bütün uygulamalar tarafından kullanılabilir. Bir assembly'yi GAC'a(Global Assembly Cache) yüklemek için 3 yöntem kullanılır:

- Assembly'leri Windows Installer 2.0 ile yüklemek
- Gacutil.exe isimli .NET ile gelen aracı kullanmak
- /assembly 'ye Assembly dosyasını kopyalamak (C:\winnt\assembly\)

Şimdi adım adım bir shared assembly oluşturmayı görelim. Bunun için ikinci yöntemi kullanacağım. Başlangıç olarak gacutil.exe programı hakkında bilgi vermek istiyorum. Gacutil(global assembly cache utility) programı .NET ile birlikte gelir. Assembly cache'ye yeni assembly yüklemek varolan assembly'leri listelemek ve silmek için kullanılır. Komut satırından aşağıdaki parametrelerle bu programı çalıştırabiliriz.

- * gacutil /l ---> GAC 'da bulunan bütün assembly'leri listeler.
- * gacutil /i assemblydll---> assemblydll adlı shared assembly'yi GAC 'a yükler.
- * gacutil /u assemblydll---> assemblydll adlı shared assembly GAC 'dan siler.

İlk adım olarak bütün shared assembly'lere strong name dediğimiz bir isim vermeliyiz. GAC 'da bulunan bütün assembly'lerin farklı bir adı vardır. COM teknolojisindeki globally unique identifier(GUID) 'e benzetebiliriz bu isimleri. Peki bu strong name 'leri nasıl oluşturacağız? Bu iş için yine .NET ile birlikte gelen **sn.exe** adlı aracı kullanacağız. **sn** programı aşağıdaki gibi kullanılarak bir tekil isim oluşturulur ve **anahtar.snk** adlı dosyaya yazılır.

sn -k anahtar.snk

anahtar.snk dosyasını oluşturduktan sonra bu anahtar ismi projemizdeki AssemblyInfo.cs dosyasındaki manifeset bölümüne ekliyoruz. Yani AssemblyKeyFile özelliğini anahtar.snk olarak değiştiriyoruz. Sonuç olarak AssemblyInfo.cs dosyası aşağıdaki gibi olacaktır.

```
.....  
[assembly AssemblyDelaySign(false)]  
[assembly AssemblyKeyFile(".././anahtar.snk")]  
[assembly AssemblyKeyName("")]  
.....
```

Bu işlemleri yaptıktan sonra projeyi "Build" edip oluşan Assembly dosyasını gacutil.exe yardımıyla GAC 'a ekliyoruz. Bu işlemi komut satırından aşağıdaki gibi yapmalıyız.

```
gacutil /i Assembly1.dll
```

Bu işlemi yaptıktan sonra shared olan bu assembly'ye istediğimiz .NET projesinden ulaşabiliriz. Tabiproject->Add reference menüsünden shared assembly'ye referans verdikten sonra yapabiliriz bunu. Shared assembly olduğu için yeni bir projede bu assembly kullandığımız da lokal bir kopyası oluşturulmaz dolayısıyla GAC üzerinden erişilir.

Kaynaklar

C# Primer (A Practical Approach) - Stanley Lipman
MSDN Kütüphanesi

Kaynak Dosyalarının Kullanımı(Resource Files)

Bir uygulamanın içindeki kaynaklar neler olabilir; resimler, müzikler ve yazılar(string). Bu makalede bu tür kaynakların, harici olarak programımıza nasıl ekleneceğini öğreneceğiz. Derlenmiş bir program içerisinde bir yazıyı değiştirmek çok zordur. Bu yüzden sonradan değişme ihtimali bulunan kaynakları yönetmek için .NET platformu bizim için büyük kolaylıklar sağlamıştır. Bazı durumlarda da programımızın farklı dillerdeki versiyonları olabilir. Bu durumda her dil için bir string kaynak dosyası(resource file) hazırlamamız yetecektir.

Bu makalede, .NET ile birlikte gelen ve kaynak dosyaları oluşturmada kullanılan **resgen.exe** adlı programı, System.Resources isim alanında bulunan **ResourceWriter** isimli sınıfı, ve bu oluşturulan kaynak dosyaları kullanmak için yine System.Resources isim alanında bulunan **ResourceManager** adlı sınıflarının kullanımını göreceğiz. Ve tabiki bunları anlatırken basit bir uygulama üzerinden anlatacağım.

Yukarıda da bahsettiğim gibi kaynak dosyalarda resim ve yazılar bulunabilir. İlk adımda basit bir .txt dosyasına istediğimiz yazıları alt alta yazalım. Resgen.exe yardımıyla bu txt dosyasından .NET platformu için özel bir kaynak dosyası oluşturacağız. Yalnız dikkat etmemiz gereken nokta şu : bu şekilde hazırlanacak bir kaynak dosyasına resimleri ekleyemiyoruz. Eğer kaynaklarımız sadece yazılar ise bu yöntemi kullanıyoruz. Eğer kaynak olarak resim eklemek istiyorsak birazdan anlatacağım ResourceWriter sınıfını kullanarak basit bir program yazacağız. Şimdi aşağıdaki terimler.txt dosyasını oluşturun.

Pointer = Gösterici
Function = Fonksiyon
Array = Dizi
Template = Şablon
yazilar.txt

Şimdi resgen.exe yardımıyla yazilar.txt den yazilar.resources adlı kaynak dosyayı oluşturalım. resgen.exe yi çalıştırmak için Start-> Programs -> Microsoft Visual Studio.NET -> Visual Studio.NET Tools -> Visual Studio.NET Command Prompt yolunu kullanabilirsiniz. Konsol ekranına

resgen yazilar.txt

yazarak yazilar.resources dosyasının oluşmasını sağlayan. yazilar.resources dosyasını bu şekilde kullanabileceğimiz gibi XML formatında bir kaynak dosyası da oluşturabiliriz. Bunun içinde konsol ekranına aşağıdaki komutu yazın.

resgen yazilar.resources yazilar.resx

Kaynak dosyalarının nasıl kullanıldığına geçmeden önce resimlerin de eklenebileceği bir kaynak dosyası hazırlayan bir program yazalım. Bu programda yukarıda da dediğim gibi System.Resources isim alanı altında bulunan ResourceWriter sınıfını kullanacağız. Bunun için aşağıdaki programı yazıyorum.

```
using System;  
using System.Drawing;  
using System.Resources;  
  
class Class1  
{
```

```

static void Main(string[] args)
{
    ResourceWriter resw = new ResourceWriter("yazilar2.resources");
    Image resim = Image.FromFile("logo.gif");

    string anahtar,deger;

    for(int i=0 ; i<=3 ; i++)
    {
        Console.Write("Kaynak için anahtar kelime girin: ");
        anahtar = Console.ReadLine();
        Console.WriteLine();
        Console.WriteLine("Girdiğiniz anahtarın değerini girini: ");
        deger = Console.ReadLine();

        resw.AddResource(anahtar,deger);
    }
    resw.AddResource("Cslogo",resim);
    resw.Close();
}
}

```

Biraz programı açıklamakta fayda var. Visual Studio.NET 'de yeni bir Console Uygulaması açın. Resimlerle iş yapabilmek için daha doğrusu Image sınıfını kullanabilmek için System.Drawing isim alanını eklememiz gerekir. Bunun için Project->Add Reference menüsünü kullanıp System.Drawing.dll i için projemize referans verelim. Kaynak dosyayı oluşturmak için ise System.Resources isim alanında bulunan ResourceWriter sınıfını kullanıyoruz. Programımızın başında yeni bir ResourceWriter nesnesi oluşturuyoruz. Varsayılan yapıcı işlevine ise oluşturacağımız kaynak dosyasının ismini veriyoruz. Daha sonra kaynak dosyasına ekleyeceğimiz bir resim dosyasından Image türünden bir nesne tanımlıyoruz. Yine aynı şekilde Image sınıfının yapıcı işlevine resim dosyasının adını gönderiyoruz. Dosyanın çalışan programla aynı klasör içinde olmasına dikkat edin. Aksi halde FileNotFoundException hatası alırız. Daha sonra bir for döngüsü yardımıyla 4 defa kullanıcıdan kaynak için anahtar ve değer girilmesini istiyoruz. Kaynak dosyasına kaynakları eklemek için ResourceWriter sınıfının overload edilmiş iki üye işlevini kullanıyoruz. Bu iki üye işlevinin prototipi aşağıdaki gibidir.

```

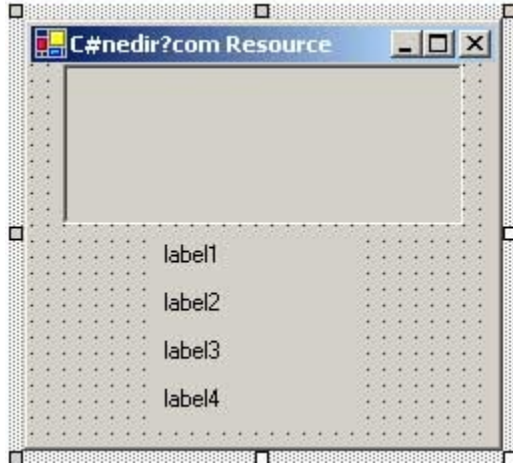
public void AddResource(string, object); // bu fonksiyonu kaynağa Image nesnesini eklemek için kullanıyoruz.
public void AddResource(string, string); // bu fonksiyonu ise iki string anahtar-değer ikilisini kaynak dosyasına girmek için kullanıyoruz.

```

Son olarak Close işlevi ile hafızada bulunan bilgiler yazılar.resource dosyasına yazılır. Bu işlevi kullanmadığımızda bilgiler dosyaya kaydedilmeyecektir.

Bu şekilde oluşturduğumuz kaynak dosyasının resgen.exe ile oluşturduğumuzdan tek farklı kaynak dosyasına bir resim bilgisinin binary olarak yerleştirilmesidir. Kaynak dosyası oluşturmanın iki yöntemini gördükten sonra şimdi bu kaynak dosyaları bir uygulamada nasıl kullanacağımızı görelim. Bu amaçla yeni bir Windows Uygulaması başlatalım. Amacımız bir picturebox yardımıyla kaynak dosyadaki binary resim bilgilerini göstermek ve yazıları da bir label kontrolü üzerinde göstermek. Öncelikle Visual Studio.NET 'de bulunan solution explorer penceresinden projemize hazırladığımız kaynak dosyasını eklememiz gerekir. Bunun için solution explorer'daki projemize sağ tıklayıp Add->Add Existing Item 'dan yazilar2.resource adlı kaynak dosyasını seçelim. Bu yöntemle kaynak dosyası projemize Embed Resource olarak eklenecektir. Properties penceresindeki Build Action özelliğini kullanarak bu ayarı değiştirebiliriz. Kaynak dosyasını

ekledikten sonra aşağıdaki gibi bir form penceresi tasarlayın. Form üzerine bir picturebox ve 4 tane label kontrolü ekleyin.



Formumuzu tasarladıktan sonra Form1 'in load metodunu aşağıdaki gibi düzenleyin.

```
private void Form1_Load(object sender, System.EventArgs e)
{
    ResourceManager rsm = new
    ResourceManager("winAppRes.yazilar2",Assembly.GetExecutingAssembly());
    pictureBox1.Image = (Image)rsm.GetObject("Cslogo");

    label1.Text = rsm.GetString("Pointer");
    label2.Text = rsm.GetString("Function");
    label3.Text = rsm.GetString("array");
    label4.Text = rsm.GetString("Template");
}
```

Yukarıdaki metotda form1 yüklendiğinde ResourceManager yardımıyla kaynak dosyadaki bilgileri ,form üzerindeki kontrollere yerleştiriyoruz. Bu kodun çalışabilmesi için System.Reflection(Assembly sınıfı için) ve System.Resources(ResourceManager sınıfı için) isim alanlarının projeye using ile eklenmesi gerekir. İlk olarak o an üzerinde çalışılan assembly için bir ResourceManager nesnesi oluşturuyoruz. Yeni bir ResourceManager nesnesi oluştururken ResourceManager 'in yapıcı işlevine kaynak dosyanın projedeki göreceli yolunu ve o an üzerinde çalıştığımız Assembly nesnesini geçiriyoruz. Benim oluşturduğum projenin ismi winAppRes olduğu için 1. parametre "winAppRes.yazilar2" olmalıdır. (Dikkat edin 1. parametrede kaynak dosyasının uzantısı olan .resource ekini yazmadık). 2. parametreye ise Assembly(System.Reflection) sınıfının statik üye işlevi olan GetExecutingAssembly ile dönen Assembly nesnesini geçiriyoruz. Kaynak dosyasındaki verileri formun üzerindeki kontrollere yerleştirirken ResourceManager sınıfının iki ayrı üye işlevini kullanıyoruz. Bunlardan GetObject işlevi ile pictureBox 'a kaynaktaki resmi aktarıyoruz. GetObject işlevinin parametresi kaynak dosyasındaki resime ait verinin anahtar adını geçiriyoruz. Hatırlarsanız kaynak dosyayı oluştururken resim için "Cslogo" anahtarını kullanmıştık. GetObject ile geriye dönen nesne object türünden olduğu için resim bilgisini pictureBox 'a yerleştirebilmek için (Image)rsm.GetObject("Cslogo") ifadesiyle tür dönüşümü yapıyoruz. GetString işlevi ile de kaynak dosyasındaki yazıları alıyoruz. Bu işlevden geriye dönen değer parametre olarak verilen anahtara ait string değeri olduğu için label kontrollerinin text özelliğine atayabiliriz.

Programı derledikten sonra aşağıdaki ekran görüntüsünü elde etmelisiniz.



Not : Eğer kaynak dosyasını resgen.exe ile XML formatında hazırlasaydık yukarıdaki işlemlerin aynısı yine geçerli olacaktı. Benim tavsiyem kaynak dosyalarını XML formatında hazırlamanız yönündedir. Hem Visual Studio.NET ortamında kaynak dosyayı daha rahat düzenleyebilirsiniz hem de XML formatındaki dosyaya istediğiniz ortamlardan rahatlıkla ulaşabilirsiniz.

C#'da ArrayList Sınıfının Kullanımı

Programların çoğunda birden fazla aynı tipte değişkenlere ihtiyaç duyarız. Bu sorunun çözümü olarak birçok dilde kullanılan veri yapıları ,dizilerdir. Bildiğimiz klasik dizilerin programlama tekniklerine getirdikleri kolaylıkların dışında birtakım kısıtlamaları da vardır. Bu makalede klasik dizilerde sık sık karşılaştığımız çeşitli sorunları ve bu sorunları nasıl çözebileceğimizi inceleyeceğiz.

.NET platformunun sınıf kitaplıklarında bulunan ve programcılarının işlerini çok kolaylaştıran ArrayList sınıfı ile klasik dizilerde karşılaştığımız sorunları nasıl çözeceğimizi göreceğiz.

Klasik dizilerle çalışırken karşılaşılabileceğimiz temel sorunları şu şekilde sıralamak mümkündür:

- **Dizilerin sınırları sabittir.**
- **Dizilerin tüm elemanları aynı türden olmalıdır.**
- **Kullanmadığımız dizi elemanlarından dolayı bellek alanları gereksiz yere işgal edilmektedir.**

Örneğin sayısını bilemediğimiz bir dizinin eleman sayısını 500 olarak belirlediğimizi varsayalım. Çalışma zamanında dizimizin sadece 10 elemanını kullandığımız durumda diğer 490 elemanlık bellek alanı boş olarak kalır. Öte yandan dizimizde tutmak istediğimiz değişkenlerin sayısı 501 bir olduğu bir durumda "IndexOutOfRangeException" istisnai durumu ortaya çıkar ve program bu hatadan dolayı sonlanır.

Mesela aşağıdaki kodu derlemeye çalışalım:

```
using System;

class CshaprNedirCom
{
    static void Main(string[] args)
    {
        int[] intDizi= new int[10];

        try
        {
            intDizi[20]=5;
        }

        catch( Exception e)
        {
            Console.WriteLine(e.GetType());
        }

        Console.ReadLine();
    }
}
```

Yukarıdaki programda **intDizi** 'mizi 10 eleman alacak şekilde tanımlamamıza rağmen bu dizinin 20. elemanına ulaşip ona birşeyler atamaya çalıştık. Bu durumda programımız

çalışırken hata verdi. Çünkü dizinin sınırları bellidir ve bu sınırların dışına çıkamıyoruz. Eleman sayısı ihtiyacımıza göre değişen bir veri yapısı olması gerçekten hoş olmaz mıydı? Evet C#'da böyle bir dizi yapımız var. Bunun ismi **ArrayList**'tir.

ArrayList sınırları dinamik olarak değişebilen diziler olarak tanımlanır. Bu veri yapısı .NET sınıf kütüphanesinin **System.Collections** isim alanında bulunur. İsterseniz ArrayList'i nasıl kullanacağımızı bir örnekle inceleyelim:

```
using System;
using System.Collections; // ArrayList sınıfını kullanmak için
                           // System.Collection isim alanını eklemeliyiz..

class CshaprNedirCom
{
    static void Main(string[] args)
    {
        ArrayList aList= new ArrayList(); // aList isimli ArrayList nesnesi
        oluşturalım.

        // aList nesnemize sırası ile 5, 8, 1, 17 ve 20 değerlerini
        // Add metodu ile ekleyelim.
        aList.Add(5);
        aList.Add(8);
        aList.Add(1);
        aList.Add(17);
        aList.Add(20);

        // aList'in elemanlarını ekrana yazdırıyoruz:
        Console.WriteLine("\t aList'in elemanları:");
        foreach(int eleman in aList)
            Console.WriteLine(eleman);

        // aList dizimizden 8 ve 20 değerlerini çıkartalım:
        aList.Remove(8);
        aList.Remove(20);

        // aList dizimize 66 ve 4 değerlerini ekleyelim:
        aList.Add(66);
        aList.Add(4);

        Console.WriteLine("\n\t aList dizisinden 8 ve 20\' çıkartıp, 66 ve 4 ekledik:");
        foreach(int eleman in aList)
            Console.WriteLine(eleman);

        Console.ReadLine();
    }
}
```

Yukarıdaki örneğimizde öncelikle ArrayList sınıfını kullanmak için NET sınıf kütüphanesinin **System.Collections** isim alanını kullanacağımızı `using System.Collections;` ile bildiriyoruz. Main fonksiyonumuzun içindeki ilk satırda, `ArrayList aList= new ArrayList()` , **aList** ismini verdiğimiz ArrayList sınıfından bir nesne oluşturuyoruz. **aList** nesnemizi

oluşturduğunuz satırdan sonraki beş satırda ArrayList sınıfının **Add()** metodu ile **aList** adlı dizimize elemanları ekliyoruz.

Daha sonra **aList** diziminizin elemanlarını **ForEach** döngüsü ile tek tek ekrana yazdırıyoruz. ArrayList sınıfındaki bir nesnenin elemanlarını tek tek silmek için **Remove()** metodunu kullanırız. **Remove()** metodu ile istediğimiz elemanı diziden atabiliriz. Biz de 8 ve 20 elemanlarını diziden attık. Son olarak dizimize 66 ve 20 elemanlarını ekleyip dizinin son halini ekrana yazdırdık.

C# dilinde normal diziler bildiğiniz gibi sadece aynı tipten verileri tutar. Ama ArrayList sınıfına ait dizilerimiz her türlü nesneyi aynı dizi içinde tutabilir. Yani aynı dizide **int**, **float**, ve **string** tiplerindeki değişkenleri depolama şansımız var. Mesela aşağıdaki kod C# dili kuralları çerçevesinde geçerli bir koddur:

```
ArrayList karmaList= new ArrayList(); // karmaList isimli ArrayList nesnesi oluşturalım.
```

```
karmaList.Add("Ahmet");  
karmaList.Add(12);  
karmaList.Add(true);  
karmaList.Add(32.4f);  
karmaList.Add('c');
```

Bu kod ile **karmaList** isimli ArrayList nesnemizin içinde **string**, **int**, **bool**, **float** ve **char** tiplerinden oluşan verileri aynı anda saklarız. ArrayList sınıfının bize sunduğu diğer bir güzel özellik ise tek bir komut ile ArrayList dizimizin içerisindeki elemanları ters çevirebilmemizdir. Ters çevirme işlemi için **Reverse()** metodu kullanılır. İsterseniz **Reverse()** metodunu ve ArrayList'lerde nasıl birden farklı türdeki elemanları kullanacağımızı bir örnekle inceleyelim:

```
using System;  
using System.Collections; // ArrayList sınıfını kullanmak için  
                           // System.Collection isim alanını eklemeliyiz..  
  
class CshaprNedirCom  
{  
    static void Main(string[] args)  
    {  
        // karmaList isimli ArrayList nesnesi oluşturalım.  
        ArrayList karmaList= new ArrayList();  
  
        // karmaList'e değişik tipte elemanlar ekliyoruz.  
        karmaList.Add("Ali");  
        karmaList.Add(23);  
        karmaList.Add(false);  
        karmaList.Add(52.8d);  
        karmaList.Add('r');  
  
        // karmaList'in elemanlarını ekrana yazdırıyoruz:  
        Console.WriteLine("\t karmaList'in elemanları:");  
        foreach(object eleman in karmaList)  
            Console.WriteLine(eleman);  
    }  
}
```

```
karmaList.Reverse();           // karmaList'imizi ters çeviriyoruz.

Console.WriteLine("\n ---> karmaList'in elemanları ve türleri <---");
foreach(object eleman in karmaList)
Console.WriteLine("Türü: {0,15} değeri: {1,7}",eleman.GetType(), eleman);

Console.ReadLine();

}
```

Yukarıdaki programda **karmaList** isimli ArrayList sınıfından bir nesne oluşturduktan sonra onu değişik türlerden veriler ile doldurduk. Bu dizinin elemanlarını sıra ile foreach döngüsü yardımıyla ekrana yazdırdık. `karmaList.Reverse()` satırında ise dizimizi ters çevirdik. Son işimizde ise karmaList dizisinin elemanlarını ekrana tek tek yazdırırken aynı zamanda `eleman.GetType()` her elemanın türünü bulup yazdık.

Makalemizi ArrayList sınıfının temel metodlarının tablosunu vererek bitirmek isterim. Bu tablodaki metodların işimize yaracağı kanaatindeyim.

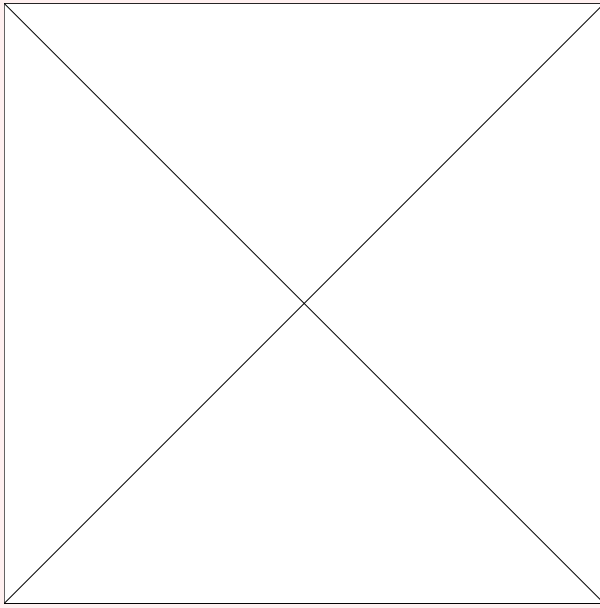
Add	Bir nesneyi ArrayList'in sonuna ekler.
BinarySearch	Sıralanmış bir ArrayList içinde bir nesneyi Binary search algoritması kullanarak arar.
Clear	ArrayList'in tüm elemanlarını siler. Sıfırlar.
Contains	Herhangi bir nesnenin ArrayList'in elemanı olup olmadığını kontrol eder.
Insert	Dizinin sonuna değilde istediğimiz bir yerine indeksini belirterek eklememizi sağlar.
Remove	Herhangi bir elemanı diziden siler.
Reverse	Diziyi ters çevirir.
Sort	ArrayList'i sıralar.

Her Yönüyle C#'da Yığın (Stack) Sınıfı

Bu yazımızda önemli veri yapılarından olan yığın (Stack) veri yapılarına giriş yapacağız. Yığın veri yapısının çalışma mantıklarını anladıktan sonra .NET sınıf kitaplıklarında yer alan **Stack** sınıfını C#'da nasıl kullanacağımızı inceleyeceğiz.

1. Yığın veri Yapısının Çalışma Şekli

Yığınlar genelde aynı tipten verilerin tutulduğu ve **Son Giren İlk Çıkar** (LIFO) çalışma mantığını kullanan veri yapıları olarak tanımlanır. Bir yığına en son giren eleman ilk olarak çıkar. Yığınları anlatırken en çok üst üste konmuş tabaklar veya herhangi bir nesne grubunda çok kullanılabilecek benzetirler. Mesela bir masanın üstünde sıra ile üst üste konmuş birden fazla tabaktan birisine ihtiyacımız olursa önce en üsttekini alırız. Bu aldığımız tabakların en son konulanıdır.



Yığınların çalışma prensibini daha iyi kavramak için yandaki canlandırmayı inceleyelim. Canlandırmada yığının dışında bulunan sayıları yığının içine koymak için **"Yığını Doldur"** düğmesine tıkladığımızda sırası ile 17, 23, 4, 55 ve 8'i yığına sokuyor. Yığın doluyken **"Yığını Boşalt"** düğmesine tıklayınca ise yığındaki sayıların yerleştirildikleri sıranın tersi sıra ile boşaltır. Bu durumda en son giren 8 sayısı ilk önce, sonra 55, sonra 4 ve bu sıra ile en son 17 sayısı yığından dışarı çıkar.

2. .NET Sınıf Kütüphanesi Yığın Sınıfı (Stack)

.NET sınıf kütüphanesinde yığın veri yapısını kullanmak için **Stack** sınıfını kullanırız. Normalde C ve C++ dillerinde yığın veri yapısını değişik veri türleri ve kendi tanımladığımız sınıflarla birlikte kullanmak zahmetli bir işti. (C'de sınıf kavramı yoktur!) Ama .NET ortamında yığınlarımız bize birçok konuda esneklikler sunarak programlamayı daha zevkli ve verimli hale getiriyor.

.NET'in yığın (**Stack**) sınıfını kullanmak için program kodunun baş tarafına **using System.Collections;** eklememiz gerekir. Buradan yığın sınıfı **System.Collections** isim alanında bulunuyor sonucunu da çıkartırız.

C# veya herhangi bir dilde yazılan yığın veri yapılarında **Push()**, **Pop()**, **Peek()** veya **Top()**, **Clear()** fonksiyonları ve **Count**, özelliği temel olarak bulunur. Temel fonksiyonların yanında **Clone()**, **CopyTo()**, **ToArray()**, **Contains()** ve **Equals()** metodları .NET'in yığın sınıfında yer alır.

Yığın sınıfının **Push()** metodu yığına yeni bir eleman ekler. **Pop()** metodu ile yığının en üstündeki elemanı yığından siler ve silinen elemanı geriye döndürür. Eğer yığının

tepesindeki elemanı öğrenmek istersek **Peek()** metodu işimize yarar. Bu metod yığının tepesindeki nesneyi döndürür ama bu nesneyi yığından silmez.

```
using System;
using System.Collections; // Stack sınıfı bu isim alanı içinde bulunur.

class YiginSinifi1
{
    public static void Main(string[] args)
    {
        // Stack sınıfından yigin nesnemizi tanımlıyoruz.
        Stack yigin = new Stack();

        // Yigini değişik değerlerde dolduruyoruz..
        yigin.Push(12);
        yigin.Push(5);
        yigin.Push(23);
        yigin.Push(34);
        yigin.Push(70);
        yigin.Push(8);

        Console.WriteLine("Yığımızın ilk hali...");
        ElemanlariYaz(yigin);

        // Yigininin tepesinden bir sayı aldık
        // ve bunu sayi değişkenine atayıp ekrana yazdıralım
        int sayi = (int) yigin.Pop();
        Console.WriteLine("\n Yığından {0} sayısını aldık", sayi);

        // Yigininin tepesinden bir sayı daha aldık
        // ve bunu sayi değişkenine atayıp ekrana yazdıralım
        sayi = (int)yigin.Pop();
        Console.WriteLine("\n Yığından {0} sayısını aldık", sayi);

        // Şimdi ise Yigininin tepesindeki sayıya bir bakalım
        // bu sayıyı yığından çıkarmıyoruz.. Sadece ne olduğuna bakıyoruz..
        sayi = (int) yigin.Peek();
        Console.WriteLine("\n Yığının tepesindeki sayı şu anda : {0}", sayi);

        Console.ReadLine();
    }

    public static void ElemanlariYaz(Stack yigin)
    {
        object obj = new Object();
        Stack yeniYigin = (Stack)yigin.Clone();

        if(yigin.Count!=0)
        {
            while(yeniYigin.Count>0)
            {
                obj = yeniYigin.Pop();
            }
        }
    }
}
```

```

        Console.WriteLine("\t" + obj.ToString());
    }
}
else Console.WriteLine("Yığın boş...!");
}
}

```

Yukarıdaki programda önce Stack sınıfından **yigin** isimli bir nesne oluşturuyoruz. Sonraki altı satırda yığımıza 12, 5, 23, 34, 70 ve 8 tamsayılarını **Push()** metodu ile ekliyoruz. **EkranaYaz()** ismini verdiğimiz static fonksiyonumuz (bu fonksiyon tam olarak optimize edilmiş bir fonksiyon değil!) ile yığımızda bulunan elemanları ekrana yazdırıyoruz. Daha sonra yığından iki tane elemanı **Pop()** metodu yardımıyla alıyor ve herbirini ekrana yazdırıyoruz. Programın son kısmında ise **Peek()** metodunu kullanarak yığının en üstündeki elemanın ne olduğunu öğreniyoruz.

Yığın sınıflarında bulunan diğer iki temel fonksiyonlar olan **Count** özelliği ve **Clear()** metodlarıdır. Bunlardan **Count**, yığın nesnesinde bulunan elemanların sayısını geriye döndüren bir özelliktir. Özellikler C# dilinde sınıflarda bulunan üye değişkenlerin değerlerini öğrenmemize ve onların değerlerini değiştirmemize yarayan bir tür fonksiyonlardır. **Count** özelliği eleman sayısını int tipinde döndürür ve sadece okunabilen (readonly) yapıdadır. Özellikler program içinde çağrılırken parantezleri kullanmayız. Eğer yığını boşaltmak/temizlemek istersek **Clean()** metodu işimizi yarayacaktır. **Clean()** metodu hiçbir parametre almaz ve hiçbir şey döndürmez. Herhangi bir yığın nesnesinin içinde bir elemanın olup olmadığını anlamak için **Contains()** metodu kullanılır. Bu metod aranacak nesneyi alır ve geriye **true** veya **false** değerlerini döndürür. İsterseniz aşağıdaki programda **Contains()** ve **Clear()** metodları ile **Count** özelliklerini nasıl kullanabileceğimizi görelim:

```

using System;
using System.Collections; // Stack sınıfı bu isim alanı içinde bulunur.

class YiginSinifi1
{
    public static void Main(string[] args)
    {
        // Stack sınıfından yigin nesnemizi tanımlıyoruz.
        Stack yigin = new Stack();

        // Yığımıza yeni elemanlar ekliyoruz.
        yigin.Push("Ahmet");
        yigin.Push("Sefer");
        yigin.Push("Cemal");
        yigin.Push("Onur");
        yigin.Push("Aziz");

        // Yığında kaç tane eleman bulunduğunu bulup yazalım.
        int elemanSayisi= yigin.Count;
        Console.WriteLine("\nYığımızdaki eleman sayısı: {0}", elemanSayisi);

        // Yığındaki elemanlar.
        Console.WriteLine("\nYığındaki elemanlar: ");
        ElemanlariYaz(yigin);
    }
}

```

```

//Contains() metodunun kullanımı:
if(yigin.Contains("Sefer"))
    Console.WriteLine("\nYığında Sefer elemanı var...");
else
    Console.WriteLine("\nYığında Sefer elemanı yok...");

// Yığını boşaltalım.
yigin.Clear();

// Yığını boşalttıktan sonra kaç tane eleman bulunduğunu bulup yazalım.
elemanSayisi= yigin.Count;
Console.WriteLine("\nYığımızdaki eleman sayısı: {0}", elemanSayisi);

Console.ReadLine();
}

public static void ElemanlariYaz(Stack yigin)
{
    object obj = new Object();
    Stack yeniYigin = (Stack)yigin.Clone();

    if(yigin.Count!=0)
    {
        while(yeniYigin.Count>0)
        {
            obj = yeniYigin.Pop();
            Console.WriteLine("\t"+ obj.ToString());
        }
    }
    else Console.WriteLine("Yığın boş...!");
}
}

```

Hemen üstteki programdan önceki programda yığımıza int tipinden nesneler (c#'ta primitive türler dahil herşey nesnedir!) yerleştirmiştik. Bu örnekte ise string sınıfına ait nesneleri yığımıza ekledik ve onlar üzerinde işlemler yaptık. Yani yığın sınıfımız herhangi bir nesneyi tutabilecek yetenekler sabit. İster temel veri türleri olsun (int, byte, double veya bool) ister kendi tanımladığımız veri türleri olsun yığın sınıfımıza ekleyip çıkartabiliriz.

Yukarıdaki programda **yigin** olarak oluşturduğumuz ve yığın sınıfındaki nesnemize beş tane veriyi ekliyoruz. Sonra yığında kaç tane eleman olduğunu bulmak için **Count** özelliğinden faydalıyoruz. Yığındaki elemanları yazdırmak için **ElemanlariYaz()** sabit fonksiyonumuzu kullanıyoruz. **Contains()** metodunu kullanımına örnek olması amacıyla if deyimi içinde **yigin.Contains("Sefer")** sorgusunu yapıyoruz. Eğer **Sefer** elemanı yığında mevcutsa ekrana yığında olduğunu, yoksa yığında olmadığını yazdırıyoruz. Programın geriye kalan kısmında yığını boşaltmak için **Clear()** metodunu kullanıyoruz, yığındaki eleman sayısını tekrar bulup bunu yazdırıyoruz. Eğer tekrar **ElemanlariYaz()** fonksiyonunu kullansaydık. "Yığın Boş..!" uyarısını alırdık!

.NET sınıf kütüphanesinde bulunan Yığın (Stack) sınıfının getirdiği kolaylıklar yukarıdakilerden daha fazladır. Mesela herhangi bir yığın nesnemizi başka bir yığının içine kopyalayabiliriz. Bir yığını başka bir yığına kopyalamak için **Clone()** metodunu

kullanabiliriz. Ayrıca bir yığın nesnesini herhangi bir dizinin içine kopyalamak için **ToArray()** metodu kullanılabilir. İki tane yığının birbirlerine eşit olup olmadığını öğrenmek için **Equals()** metodu hemen yardımımıza yetişir. Burada şunu belirtmekte yarar var: Equals() metodu sanal (virtual) bir fonksiyon olup c#'daki tüm nesnelerin türediği **System.Object** nesnesine aittir.

Bu makalede inceleyeceğimiz son program aşağıdadır. Bu programla **Clone()**, **Equals()** ve **ToArrayList()** metodlarını programlarımız içinde ne şekilde kullanacağımızı öğrenebiliriz.

```
using System;
using System.Collections; // Stack sınıfı bu isim alanı içinde bulunur.

class YiginSinifi1
{
    public static void Main(string[] args)
    {
        // Stack sınıfından yigin nesnemizi tanımlıyoruz.
        Stack yigin1 = new Stack();

        // Yığımıza yeni elemanlar ekliyoruz.
        yigin1.Push("Ahmet");
        yigin1.Push("Sefer");
        yigin1.Push("Cemal");
        yigin1.Push("Onur");
        yigin1.Push("Aziz");

        //İkinci yığınızı tanımlıyor ve yigin1'in
        // bir kopyasını yigin2'ye koyuyoruz..
        Stack yigin2= (Stack)yigin1.Clone();

        // yigin1'den bir eleman çıkartıyoruz.
        yigin1.Pop();

        //yigin1 ve yigin2 nesnelerimizin en üstteki
        // elemanlarına bir bakalım:
        Console.WriteLine(" Peek of Yığın2: "+ yigin2.Peek());
        Console.WriteLine(" Peek of Yığın1: "+ yigin1.Peek());

        //yigin1 ve yigin2 eşit mi? Bir bakalım:
        Console.WriteLine("\n yigin1 ve yigin2 eşit? --> "+ yigin1.Equals(yigin2));

        //yigin2'yi kopyalamak için yeni bir dizi oluşturalım:
        Array arr = new Array[5];

        // yeni oluşturduğumuz diziye yigin2'yi kopyalayalım:
        arr = yigin2.ToArray();

        // arr nesnesinin elemanları:
        Console.WriteLine( "\n\n arr nesnesinin elemanları:\n"+
            "\n\t"+arr.GetValue(0)+
            "\n\t"+arr.GetValue(1)+
            "\n\t"+arr.GetValue(2)+
            "\n\t"+arr.GetValue(3)+
```



```
        "\n\t"+arr.GetValue(4)    );

    Console.ReadLine();
}

public static void ElemanlariYaz(Stack yigin)
{
    object obj = new Object();
    Stack yeniYigin = (Stack)yigin.Clone();

    if(yigin.Count!=0)
    {
        while(yeniYigin.Count>0)
        {
            obj = yeniYigin.Pop();
            Console.WriteLine("\t"+ obj.ToString());
        }
    }
    else Console.WriteLine("Yığın boş...!");
}
}
```

Yazımızda bilgisayar programlama alanında en önemli veri yapılarından biri olan yığınların (Stack) nasıl çalıştıklarını ve .NET sınıf kütüphanesinde bulunan Stack sınıfını ve metodlarının nasıl işimize yarayacak şekilde kullanabileceğimizi öğrendik. Umarım bu yazının size gerçek manada yararı olur.

C#'ta Gösterici(Pointer) Kullanmak – I

Göstericiler(Pointer) alt seviye programlama için olmazsa olmaz yapılardır. Göstericiler nesnelerin bellekte tutuldukları adresleri saklayan veri yapılarıdır. Bu makalede C#'ta kullanımı çok fazla gerekli olmayan göstericilerin nasıl kullanıldıklarını inceleyeceğiz. Bu yazı göstericiler hakkında temel bilgilere sahip olduğunuzu varsaymaktadır.

.NET' in altyapısında gösterici kavramı sıklıkla kullanılırken göstericilerin açıkça kullanımı programcılar için gizlenmiştir. Bunun nedeni gösterici kullanımının programlardaki görülemeyen hatalara sıkça yol açabilmesidir. Özellikle dili yeni öğrenenler için gösterici hataları içinden çıkılmaz bir hale gelebilmektedir. C#'ta göstericiler yerine referans değişkenleri mevcuttur. Referanslar heap bellek bölgesindeki nesnelerin başlangıç adresini tutar. Ancak bu adresin değerine kesinlikle ulaşamayız. Oysa C ve C++ dillerinde stack bölgesindeki değişkenlerin de adreslerine erişebilmemiz mümkündür. Üstelik değişkenlerin adreslerini örneğin 0x965474 şeklinde elde etmemiz bile mümkündür. C ve C++ programcılarına pek yabancı gelmeyecektir bunlar, ancak programlama dünyasına C# ile giren biri için göstericilere neden ihtiyaç duyabileceğimiz pek anlamlı gelmeyebilir. Şunu da söyleyelim ki çok istisnai durumlar dışında göstericilere ihtiyacımız olmayacak, peki bu istisna durumlar nelerdir?

- **Geriye Uyumluluk(Backward Compatibility)** : COM ve WinAPI'deki fonksiyonlar gibi sık sık gösterici kullanan fonksiyonları C# ile programlarımızdan çağırabilmek için parametre olarak gösterici alan fonksiyonlara gösterici göndermemiz gerekebilir. Eğer C# ta gösterici kullanımına izin verilmemiş olsaydı tür uyumsuzluğu yüzünden gösterici kullanan eski COM ve DLL'lere erişebilmemiz mümkün olamazdı.
- **Performans** : C ve C++ dillerinin günümüze kadar çok popüler olmasının altında yatan nedenlerden biri de belleğe direkt erişimi sağladığı için performansın inanılmaz derecede yükselmesidir. Bizim için performansın çok önemli olduğu yerlerde gösterici kullanmamızdan daha doğal birşey olamaz.
- **Alt Seviye İşlemler** : Donanım arayüzleri ile direkt bir ilişki içerisinde olacak programlarda göstericilerin kullanımı mecburi gibidir. Bazen de belleğe kullanıcıların direkt erişebilmesi gereken programlar olabilir. Bu durumlarda da göstericilerden faydalanabiliriz.

Bütün bunların yanında gösterici kullanmanın bir çok sıkıntıya da beraberinde getireceği kesindir. Öncelikle kullanımının zor olması ve kestirilmesi zor olan hatalara yol açabilmesi bu sıkıntıların en başında gelenidir. Zaten C#'ta gösterici kullanacağımız zaman kodu **unsafe**(güvensiz) anahtar sözcüğü ile işaretlememiz gerekir. Aksi halde program derlenemeyecektir. Normal bir metot içinde gösterici kullanımı yasaklanmıştır.

Şimdi de unsafe anahtar sözcüğünün kullanımına örnekler verelim.

1-) unsafe olarak işaretlenen sınıfların bütün metotlarında gösterici kullanabiliriz.

```
unsafe class Sınıf
{
}
}
```

2-) Normal bir metot içinde herhangi bir bloğu unsafe olarak aşağıdaki gibi işaretleyip dilediğimiz gibi gösterici kullanabiliriz. unsafe bloklarının dışında ise gösterici kullanamayız.

```
int NormalMetot(int a, string str)
{
    unsafe
    {
    }
}
```

3-) Normal bir metodu unsafe olarak işaretleyip sadece o metodun içinde de gösterici kullanabiliriz.

```
unsafe int NormalMetot(int a, string str)
{
}
}
```

4-) Bir sınıfın üye değişkenlerinden biri unsafe olarak işaretlenip gösterici olarak bildirilebilir. Ancak bir metod içerisinde yerel bir gösterici tanımlanamaz. Yerel bir gösterici tanımlamak için unsafe olarak işaretlenmiş metod yada blok kullanılır.

```
class Sınıf
{
    unsafe char *ptr;
}
```

Gösterici Tanımlama ve Gösterici Operatörleri

Göstericiler aşağıdaki gibi tanımlanabilir.

```
char* ptr1, ptr2;
int* ptr3;
```

ptr1 ve ptr2 char türden bir gösterici iken ptr3 int türden bir göstericidir. Bir göstericide iki bileşen vardır. Bu bileşenlerden birincisi adres bileşenidir. Adres bileşeni nesnenin bellekte bulunduğu başlangıç adresidir. İkinci bileşen ise tür bileşenidir. Bu bileşen ise ilgili adresteki nesneye ulaşmak istediğimizde bellekten ne kadarlık bilgili okunacağını sağlar. Örneğin int türden bir adresteki bilgiyi okumak istediğimizde 4 byte'lık bir bilgi okunacaktır, aynı şekilde char türden bir gösterici ise 2 byte'lık bir bilgi okunacaktır.

& operatörü

Adres operatörü olarak bilinen bu operatör değişkenlerin veya nesnelerin bellekte bulundukları adresleri elde etmek için kullanılır. Bu operatör hangi tür değişkenle kullanılırsa o türden bir gösterici üretilir.

* operatörü

İçerik operatörü olan *, bir adresteki bilgileri elde etmek için kullanılır. Aşağıdaki programda bu iki operatörün kullanımına bir örnek verilmiştir.

```
using System;

class Class1
{
    unsafe static void Main()
```

```

{
    int* ptr1;
    char* ptr2;

    int a = 50;
    ptr1 = &a;
    int Adres = (int)ptr1;
    Console.WriteLine("{0:X}",Adres);

    char ch = 'A';
    ptr2 = &ch;
    *ptr2 = 'B';
    Console.WriteLine(ch);
}
}

```

Bu programı derleyebilmek için komut satırı derleyicisine programın içinde unsafe bloğunun olduğunu belirtmemiz gerekir. Bunun için komut satırına

csc /unsafe KaynakKod.cs

yada

csc -unsafe KaynakKod.cs

yazarak programı derleyebilirsiniz. Eğer Visual Studio.NET kullanıyorsanız projeye sağ tıklayıp proje özelliklerine gelip Build kısmından "Allow unsafe code blocks" kısmını true olacak şekilde değiştirin.

Programı çalıştırdığınızda ekrana a değişkeninin adresi ve B karakteri yazılacaktır. Adres bilgisini ekrana yazdırmadan önce göstericide tutulan adresi normal bir türe nasıl dönüştürdüğümüze dikkat edin. Bunu yapmamızın sebebi Console.WriteLine() metodunun gösterici parametres alan versiyonunun olmamasıdır.

C#'ta tür güvenliğinin ne kadar önemli olduğunu vurgulamak için aşağıdaki deyimleri örnek verebiliriz.

```

int* ptr1;
*ptr1 = 50;

```

Bu deyimleri içeren bir program derlenemeyecektir. Çünkü ptr1 göstericisinde hangi adresin tutulduğu belli değildir. Bu yüzden adresi belli olmayan bellek bölgesine bir değer yerleştirmek imkansızdır. C ve C++ dillerinde bu kullanım tamamen geçerlidir. Ama gelin bir de bunun sakıncasına bir göz atalım. ptr1 göstericisi tanımlandığında ptr1 de rastgele bir adres değeri bulunmaktadır. Bu adreste rastgele bir adres olduğu için o an bellekte çalışan kendi programımızdaki bir değişkenin adresi bile olabilir. Yada sistemde çalışan başka bir prosesdeki elemanların adresleri olabilir. Kaynağını bilmediğimiz bir adresteki değeri * operatörü ile değiştirdiğimizde hiç tahmin edemeyeceğimiz çalışma zamanı hataları alabiliriz. İşin kötüsü adresler rastgele olduğu için programımızın test aşamasında bu hatalar oluşmayabilir. Adresler nasıl rastgele ise hataların oluşması da rastgeledir. Programımız piyasada iken böyle bir hatanın farkına varılması iş maliyetlerini ne kadar artırdığını siz tahmin edin artık. Bütün bu dezavantajlar göstericilerin asla gereksiz olduğu anlamına gelmemelidir. Sadece göstericileri kullanırken daha dikkatli davranmamız gerektiğinin göstergesidir.

Göstericiler arasında tür dönüşümleri mümkündür. Örneğin int türden bir gösterici char türden bir göstericiye aşağıdaki gibi dönüştürülebilir.

```
char* ptr1
int* ptr2;

ptr1 = (char*)ptr2
```

Aynı şekilde bir gösterici de tamsayı türlerine dönüştürülebilir. Adresler tam sayı türünden birer sayı oldukları için bunu yapabilmemiz son derece normal bir durumdur. Bir önceki çalışan programda da Console.WriteLine() metodu ile ekrana yazmak istediğimiz nesnenin adresini tamsayı türlerinden birini çevirdiğimizi hatırlayın. Örneğin aşağıdaki deyimleri içeren bir program derlenemeyecektir.

```
char* ptr1
char ch = 'A';
ptr1 = ch;

Console.WriteLine(ptr1)
```

Göstericilerle ilgili diğer önemli yapı ise **void** göstericilerdir. void olan göstericilerde tür bilgisi saklanmamaktadır. void göstericilere herhangi bir türden gösterici atanabilir. void göstericiler daha çok eskiden yazılmış API fonksiyonlarında void parametre alan fonksiyoları programlarımız içerisinden çağırmak için kullanılır. void göstericilerin kullanımına aşağıda bir örnek verilmiştir.

```
int* ptr1;
void * x;
x = ptr1;
```

sizeof operatörü

sizeof operatörü temel türlerin ve yapıların bellekte ne kadar alan kapladıklarını verir. Örneğin sizeof(int) = 4, sizeof(char) = 2 ' dir. sizeof operatörünü sınıflar için kullanamayız ancak tanımlayacağımız yapılar için kullanabiliriz.

Bu yazının sonuna geldik. Bir sonraki yazımızda yapı göstericileri tanımlamayı sınıflar ile göstericiler arasındaki ilişkiyi, gösterici aritmetiğini inceleyip örnek bir gösterici uygulaması yapacağız.

Kaynaklar:

C# and The .NET Platform (Andrew Troelsen)
Professional C# 2nd Edition (Wrox)
MSDN Kütüphanesi

C#-JAVA' nın Sonu mu?

JAVA'nın gün geçtikçe yaygınlaşan kullanım alanları, Microsoft firmasını gerçekten güç duruma düşürdü. 1995 yılında JAVA ufukta göründü ve geçtiğimiz 6 yıl içerisinde kendisi

ile rekabet eden diğer bütün popüler diller üzerindeki üstünlüğünü gösterdi. Başlangıçta Microsoft bu duruma direndi ancak daha sonra, JAVA meşhur Visual Studio nun Visual J++ adı altında bir parçası oldu. JAVA' yı çıkaran Sun Microsystems, Microsoft aleyhine lisans anlaşmasını ihlal ettiği gerekçesiyle dava açtı ve Microsoft buna karşılık bir bedel ödemek zorunda kaldı.

Sonunda ortalık yatıştı ve Microsoft Visual J++ 'ın yeni versiyonlarını çıkarmaya devam etti. Fakat Microsoft'un JAVA dilini Visual Studio paketinin içine koyma çabası başarıya ulaşamadı ve en sonunda başarısız oldu. Elbette ki bu başarısızlık ta pazarlamanın iyi yapılamamasının da etkisi olmuştur. Sonuç olarak Microsoft 1998 yılından sonra Visual J++ için yeni bir versiyon çıkarmadı, ve yakın bir gelecek içinde de bu tür planlarını askıya aldı.

Sonunda Microsoft anladı ki bir yerlerden ödünç alınan ürünler uzun vadede işe yaramayacak, bu yüzden Microsoft, daha çok JAVA'ya benzeyen ancak JAVA da olmayan bir takım yeni özellikleride ekleyerek C# (C Sharp) isimli yeni bir programlama dili tasarlamaya başladı. Bu makalenin amacı C# hakkındaki şüphelerinizi gidermek ve JAVA ile karşılaştırmaktır.

Peki, Microsoft 'un ilgilendiği bu pakette önerdikleri nelerdi?

Bu pakette Microsoft'un önerdiği iki ana yapı vardı:

.NET Mimarisi **Yeni bir dil - C#**

.NET Mimarisi

.NET mimarisinin 3 ana bileşeni vardır.

Herhangi bir dil tarafından kullanılabilecek Web Servisi denilen yeni bir ortak servisler kümesi. Bu servisler ara kod(intermediate kod) dediğimiz mimariden bağımsız bir yapı içerisinde çalıştırılır. Bu ara kod daha sonra çalışma zamanında CLR denilen birim tarafından çalıştırılır, ki bu bahsedilen CLR birimi programla ilgili kaynakları yöneten ve programın çalışmasını izleyen bir birimdir.

.NET'in birincil amacı geliştiriciler açısından Web Servislerini kullanarak mimariden bağımsız uygulamalar geliştirmeyi kolaylaştırmaktır. Örneğin bir servise hem PC hem PDA hemde Mobil cihazlardan erişilme gibi.

Şimdi bu yapıları tek tek inceleyelim.

Web Servisleri

Web Servisleri internet standartları olan XML ve HTTP üzerinden hizmet verebilen yapılardır. Daha teknik bir dille söylemek gerekirse, web servisleri internet üzerinden erişilebilen bir kütüphane, bir fonksiyon yada bir veritabanı gibi düşünülebilir. Örneğin; bir web servisi herhangi bir şirkete ait bir yıllık ciroyu hesaplayıp veren bir fonksiyon içerebilir. Bu fonksiyon parametre olarak bir yıl değeri almaktadır. Dolayısıyla istemci uygulamamız geri dönüş değeri 2000 yılına ait ciroyu Yıllık_ciro(2000) gibi bir fonksiyonu çağırarak elde edebilir.

Aslına bakarsanız bir firmada farklı platformlara yönelik hazırlanmış uygulamalar arasında ki senkronizasyonu sağlamak amacıyla Web Servisleri kullanılabilir. Buna rağmen Web Servisi kelimesindeki "Web" sözcüğü ile internet ortamında web servislerinin kullanımını ön plana çıkarmak için kullanılmıştır. Fakat normal uygulamalarda da çoğu durumda Web servislerinden faydalanmamız mümkündür.

Web servislerinin en güzel yani servisin her yerden istenebilmesidir. Kısacası web servisi uygulama alanlarına Windows, Linux, Unix, Mac-Os gibi platformların yanısıra komut satırından çalışan diğer ortamlar bile girebilir. Yani Web servisleri sayesinde platform bağımsız uygulamalar geliştirebiliriz. İnternet standartlarının platform bağımsız olması ve web servislerinin internet üzerinden çağırılması, web servislerinin herhangi bir işletim sisteminin yada programlama dilinin tekelinde olmamasını sağlamıştır.

Ara Kod(Intermediate Code)

Ara kod! Kulağa pek hoş gelmiyormu? Bu bana birşeyi hatırlatıyor....hımmmm..... JAVA? Evet, elbette. Microsoft açık bir şekilde .NET ile JAVA' nın çalışma platformundan esinlendiğini kabul etti böylece. .NET ortamında, programlar bir ara dil üzerinden ikili koda çevrilir. İşletim sisteminden ve donanımdan bağımsız olan bu dile MSIL(Microsoft Intermediate Language) denir. (Çevirenin notu: MSIL artık IL olarak anılmaktadır.) IL kodu, JAVA platformundaki JVM de olduğu gibi CLR tarafından derlenip makina koduna çevrilir. IL kodunu makine koduna çeviren derleyicilere ise JIT(Just In Time) derleyicileri denilmektedir.

IL kodu şeklinde derlenmiş bir uygulamaya taşınabilen çalıştırılabilir(Portable Executables) uygulama denir. CLS nin şartı bütün dillerin ortak bir noktada toplanmasını sağlayacak kurallar oluşturmaktır. Ancak burada bir problemle karşılaşıldı, oda bütün dillerin farklı semantik yapıya sahip olmasıdır, örneğin her dilin farklı özellikleri olabilir, bazı diller prosedürel tekniği destekler, bazıları nesne yönelimi tekniğini destekler, bazılarında operatör yükleme varken bazılarında yok, ve diğerleri.

Bir dilin .NET uyumlu olması için CLS(Common Language Specification) standartlarına uyması gerekir. Dolayısıyla bir dilin .NET ortamına uyumlu olması için sağlaması gereken iki koşul vardır:

CLS ile uyumluluk
Kaynak kodu IL koduna çevirecek derleyici

Peki yukarıda bahsedilen komplek yapı bize ne kazandıracak? Diller arası çalışma? Bütün .NET uyumlu dillerde yazılmış kodlar ara koda çevrileceği için herhangi bir dil de yazılmış bir nesneyi başka bir dilde yazdığımız programda oluşturabiliriz. Örneğin .NET uyumlu C# dilinde yazılmış bir sınıf ile, VB programcısı bir nesne oluşturabilir.

Peki JAVA'nın bu durumdaki rolü ne? Yada Microsoft JAVA yı devre dışı mı bıraktı? Tam olarak değil. JAVA nın .NET dilleri arasında olması düşünüldü. JAVA kodunu IL koduna çevirecek derleyici Rational firması(ünlü UML yazılım geliştirme aracını geliştiren firma) tarafından hazırlanmaktadır.(Not: bu yazının yayıldığı tarihte JAVA derleyicisi yeni yazılıyordu). Peki bu eskisiyle aynı JAVA mı olacak? Aslına bakarsanız hayır. JAVA kodları artık byte kod yerine IL koduna çevrilecek.Yeni JAVA, J2EE'nin sağladığı RMI, JMS, JDBC, JSP vs gibi API' lerden faydalanamayacak.Yeni geliştirilen JAVA ile EJB mantığı yerini .NET'in dağıtık nensne modeline bırakmıştır.

CLR(Common Language Runtime)

Yukarıda da bahsettiğim gibi CLR, konsept olarak JVM(Java Virtual Machine)'ye çok benzemektedir. JVM bellek yönetimi için kaynakların düzenlenmesi, gereksiz bilgi(çöp bilgi) toplama ve işletim sistemi ile uygulama arasındaki iletişimi sağlayan bir birimdir.

CLR, dizilerin sınırlarının aşılması, tahsis edilmeyen bellek alanına ulaşılması, bellekteki gerekli alanların üzerine yazılması gibi programların çökmesine sebep olan geleneksel hataların durumunu izler. Bu izleme hızlıktan fedakarlık sağlamak demektir. Microsoft bu

yöntemle performansın %10 oranında düşeceğini kabul etmiştir, fakat bu sistemler devamlılık ve güvenilirlik açısından daha ön plandadır.

.NET Senaryoyu Sunuyor

Bu yılın ortalarına doğru .NET'in tam sürümü piyasada olacak(Çevirenin not: Yazının yazıldığı tarihe göre düşünmelisiniz) Microsoft mimarisinin kullanıldığı eski uygulamalar .NET uyumlu Windows 2000 sunucularında hala geçerliliğini sürdürecektir. Eski mimarideki uygulamaların yeni gelecek bu mimariye taşınabilmesi için gerekli araçların olacağını Microsoft garanti etmiştir, ancak bunun iyi bir çözüm olacağı görülüyor. Halihazırdaki taşımak için geliştirilen araçların bu işlemi %100 yapamayacağı açıktır. Geliştiriciler bu taşıma işleminden dolayı epeyce bir zaman harcayacak.

.NET'in Geleceği

.NET'in gelişi ile birlikte, elektronik uygulamaların altyapısının değişmesi gerektiğini inkar edemeyiz. Her dil, .NET uyumlu olması için yapısal bir takım değişiklikler yapmalıdır. Java dilini bir düşünün. Hangi Java daha tercih edilir sizce, JVM ile JAVA mı yoksa .NET ile JAVA mı? Bu sorunun cevabını siz okurlarıma bırakıyorum.

Yeni Bir Dilin Doğuşu

Microsoft .NET'in 27 farklı dil desteğinin olduğunu iddia ediyor.(Çevirenin not: Şu an bu dillerin sayısı daha fazla) Bir dili .NET uyumlu hale getirmek dilin varolan mimarisine yeni yamalar eklemek demektir. Microsoft bu durumu değerlendirerek temel olarak .NET mimarisi hedef alınarak yeni bir dil tasarladı. Bu dilin adı C-Sharp yada C#. Bu dili daha çok JAVA yı andırıyor olmasına rağmen var olan diğer dillerden de güzel özellikler alınıp C# diline eklenmiş gibi gözüküyor.

C# nedir?

Geçen iki yüzyıl içerisinde C ve C++ dilleri ticari ve iş uygulamaları geliştirmek için en çok kullanılan diller olmuştur. Bu iki dil programcılara büyük miktarda kontrol sağlamasına rağmen, üretim açısından çok olanak sağlamıyordu.

Bu dilleri Visual Basic gibi dillerle karşılaştırdığımızda aynı uygulamayı geliştirmenin C ve C++ dillerinde daha fazla zaman aldığı bir gerçektir. Yıllardan beri istenilen tek şey güçlülük ve üretim hızının arasında bir denge kuracak bir dilin olmasıydı. Geçtiğimiz yıllarda bu problem Sun Microsystems tarafından anlaşıldı. Ve diğer dillerin çalışma mantığından tamamen farklı olan JAVA dilini geliştirdi. Bytecode konsepti, JVM ve açık kaynak kod vs ve sonuç olarak bedeva olması bu farklılıklardan bir kağıdır.

Fakat unutulmuş bir şey vardı. JAVA ile yazılan kodlar başka dillerde kullanılamıyordu. Tekrar kullanılabilirlik denen bu özelliği JAVA desteklemiyordu. Böylece, tekrar kullanılabilirliği standartlaştıracak bir ortamın gerekliliği ortaya çıktı. Bu ortam .NET ile sağlandı, zira .NET ile uyumlu dillerin sağlaması gereken CLS standartları mevcuttur. Fakat daha öncede dediğim gibi .NET ile tam uyumlu çalışacak ve diğer dillerin güzel özelliklerini sağlayacak yeni bir dile ihtiyaç vardı.

Hakkında konuştuğum bu dil C-Sharp diye okunan C#'tan başka birşey değildir.

C#, modern, nesne yönelimli, tip güvenliğine büyük önem veren, temel hesaplama ve haberleşme gibi geniş bir yelpazedeki uygulamaları geliştirmek için .NET platformu için tasarlanmış yeni bir dildir.

C#, C++ programcılarının üretkenliği ve C++ dilinin güçlüğünün göstergesi olan kontrol mekanizmalarından taviz vermeden hızlı bir şekilde uygulama geliştirme olanağı tanımaktadır.

C# ve JAVA'nın Karşılaştırılması

İyi bir dilin sağlaması gereken şeyler nelerdir? Yada programcıların gönlünü kazanmak için bir dil yapısında neyi barındırmalıdır? Bu sorular on yıllardan beri dil tasarımcılarının üzerinde durduğu tartışmaya açık sorulardır. Bütün programcılar tarafından ittifakla kabul edilmiştir ki, bu soruların cevabını en iyi şekilde veren dil JAVA'dır. C#, sentaks olarak JAVA'ya çok benzemektedir, ancak derinlerine daldıkça Microsoftun bu dili tasarlamak için çok efor sarfettiğini ve bu yeni eklenen özellikler için Microsoft'a teşekkür etmemiz gerektiğini anlarız.

Şimdi isterseniz JAVA'nın özelliklerini tek tek ele alıp bunları C#'ın ilgili özellikleriyle karşılaştıralım.

Intermediate Language(Ara dil)

JAVA kaynak kodu byte koduna çevirirken C# MSIL(IL) koduna çevirir. IL dili .NET uyumlu bütün dillerde yazılmış olan programların ortak olarak derlendiği dildir. Fakat IL ve Byte Kod'un çalışma mantığında ince bir farklılık vardır. JAVA'daki bytecode'lar yorumlanırken IL kodu derlenerek makina koduna çevrilir.

Interoperability (Diller arası uyumluluk)

JAVA'nın gücü platform bağımsız olmasından kaynaklanmaktayken C# aynı zamanda diller arasındaki uyumluluğuda sağlar. Yani dil bağımsızlığı. Güzel!. Şimdi bunları biraz daha açalım: JAVA ile yazılmış bir program JVM'nin olduğu bütün sistemlerde çalışması söz konusu iken, C# ile yazılan bir kod diğer .NET uyumlu diller tarafından tekrar kullanılabilir. Örneğin bir dilde yazılan sınıf, diğer .NET uyumlu diller ile rahatlıkla kullanılabilir.

Bellek Yönetimi

JAVA otomatik bellek yönetimi sağlamaktadır.(daha teknik bir deyimle gereksiz bilgi toplama mekanizması denir.) Bu özellik programcılar tarafından takdirle karşılanmıştır. Fakat eski C/C++ programcıları JAVA diline geçmeye çalışınca bu özellik onları rahatsız ediyordu. Bu tür programcıların problemlerini de göz ardı etmeden, C# otomatik bellek yönetiminin yanında programcının belleği kendisinin yönetmesini sağlayan sistem sunmuştur. Ne demek bu? Basit. C#'ta hala pointer kullanabiliyoruz. (Çevirenin Notu: Müthiş!)

Harici Kodlara Referans

Harici kodlar C# ve JAVA'da benzer şekilde ele alınmıştır. JAVA dilinde import anahtar kelimesi kullanılırken C#ta using anahtar kelimesi kullanılmaktadır. JAVA paket(packages) kullanırken, C# isim uzayları(namespace) kullanır. Fakat bunların anlamları aşağı yukarı aynıdır.

Veri Tipleri

Bir dilin gücü, dilin desteklediği farklı veri türleri tarafından belirlenir. Veri tipleri programcılara güçlüğü ve esnekliği sağlayan varlıklardır. C#, JAVA'daki bütün veri tiplerini sağlamıştır, bunun yanı sıra JAVA'da olmayan bazı türler de eklenmiştir, örneğin

bazı işaretli(unsigned) veri türleri JAVA da yoktur. Ayrıca C# ta kayan noktalı(floating point) bir veri türü olan 12 byte'lık decimal türü de mevcuttur.

Alan Düzenleyicileri (Field Modifiers)

C# taki Alan düzenleyicileri temel olarak JAVA dilindeki gibidir. Değiştiremeyen ya da sabit bir değişken tanımlamak için JAVA daki final dan farklı olarak read only ve const belirleyicileri kullanılır. const olan alan düzenleyiciler, ilgili değerın IL kodunun bir parçası olduđu ve sadece çalışma zamanında hesaplanacağı anlamına gelir.

Kontrol Mekanizması Oluşturma

if-else, switch, while, do-while, for, break, continue deyimleri her iki dilde aynıdır. Fakat C# ta yeni bir kontrol daha vardır. C# taki bu yeni yapı koleksiyonlar arasında dolaşmak için gerekli olan for each yapısıdır.

```
int slist(Arraylist alist)
.....
foreach (int j in alist)
{
.....
}
```

Yukarıdaki yapıda j döngü değişkeni olarak adlandırılır. Bu döngü değişkenine her iterasyonda alist dizisinin int türden olan elemanı atanır.

Arayüz ve Sınıf Bildirimi

JAVA daki extends ve implements anahtar sözcükleri C# te yerini iki nokta işaretine(:) bırakmıştır.

İstisnai Durumları Ele Alma(Exception Handling)

C# ta catch bloğundaki argüman isteğe bağlıdır. Eğer catch ile argüman belirtilmemişse, bu catch bloğu try bloğunda fırlatılacak herhangi bir hatayı yakalamak için kullanılır. Bütün catch bloğuda C# ta kullanılmayabilir. Ayrıca C# ta throws(çevirenin not: throw ile karıştırmayın) anahtar sözcüğü yoktur.

Arayüzler

C# ta bir sınıf isteğe bağlı olarak açıkça bir arayüzü uygulayabilir. Açıkça uygulanan metotlar arayüz ve sınıf arasındaki tür dönüşümü sayesinde çağrılabilir.

Kalıtım

JAVA ve C# ta sadece tekli türetme mevcuttur. Eğer çoklu türetme yapmak gerekiyorsa arayüzleri kullanmak tek yoldur.

Çok Biçimlilik(Polymorphism)

Sanal metotlar çok biçimliliği gerçekleştirmek için kullanılır. Bunun anlamı taban sınıfların, türemiş sınıflara ait aşırı yüklenmiş metotları çağırabilmesidir. JAVA da bütün metotlar sanaldır, fakat C# ta türemiş bir sınıftaki metodu taban sınıf ile çağırabilmek için metodun açıkça virtual anahtar kelimesi ile işaretlenmesi gerekir. C# ta override anahtar kelimesi bir metodun türeyen sınıfta yeniden yazılacağını bildirmek için gereklidir. Sanal olmayan bir metodu yeniden uygulamaya çalışmak derleme zamanı hatasına yol

açacaktır. Fakat eğer türemiş sınıftaki metot new anahar sözcüğü ile işaretlenirse program hata vermeden derlenecektir.

Pekala, sonuç nedir? C# te yeni bir şey bulabildinizmi, yoksa JAVA nın üvey kardeşi gibi mi duruyor? Microsoft bu uğraşının karşılığını ilerde alacak mı? Yeni bir dilin var olan uygulamalar üzerinde, değişik platformlarda hatta programcılar üzerindeki etkisi ne olacak? Bu can alıcı soruların cevabını ancak zaman verebilir. Fakat bu arada beyninizde bu soruları yavaş yavaş çözün ve düşünün. **C# gerçekten JAVA nın sonumu?**

Bu makale <http://www.csharphelp.com/archives/archive86.html> adresindeki yazıdan Türkçeye tercüme edilmiştir.

C# ile Zamanlayıcı Kullanmak (System.Timers)

Windows programlama ile uğraşanlar Timer kontrolünü sık sık kullanmışlardır. Bu yazıda System.Timers isim alanında bulunan Timer sınıfı ile bir programdaki rutin olarak

yapılması gereken işleri belirli bir zaman aralığı içinde ne şekilde yapabileceğimizi öğreneceğiz. Ancak alışılmışın dışında bir grafik arayüzü olan program yerine uygulamamızı komut satırından çalışacak şekilde geliştireceğiz. Burdaki amacımız zamanlayıcı dediğimiz timer nesnelerinin çalışma prensibini yakından görmektir.

Şöyle bir konsol uygulaması yapmanız gerektiğini düşünün. Her 5 dakikada bir, belirli bir kaynaktaki değişiklikler kontrol edilecek. Eğer kaynakta bir değişiklik varsa kullanıcıya uyarı verilecek. Bunu klasik yöntemlerle ne şekilde yapabildiniz bir düşünün? Yada ben söyleyeyim. Muhtemelen sonsuz bir döngü içerisinde sistem saatini kontrol edip belirli bir aralık geçmiş ise kaynağı kontrol eden işlevi çağıracaktınız. Eğer zamansal bir işlem yapıyorsak ki bir olayı bir zaman dilimi içinde sürekli gerçekleştirmek zamana bağlı bir olaydır; mutlaka sistem saatinden faydalanmamız gerekir. Sistem saatini makine düzeyinde işleyen bir birim olarak düşünebilirsiniz. Bilgisayar bilimlerinde bir çok algoritma sistem saati üzerine kurulmuştur. Zaten eğer zamanı ölçebilecek bir birim olmasaydı şu anki teknolojik imkanlara kavuşmamız mümkün olamazdı. Sonuç olarak zaman ne kadar değerliyse zamanı ölçebilmek te o derece önemlidir.

Bu yazıda bir metodu bir zaman dilimi içerisinde tekrar tekrar çağırmayı öğreneceğiz. Bunun için System.Timers isim alanında ki Timer isimli sınıfı kullanacağız. Timer sınıfında bulunan **Elapsed** olayı timer nesnesinin **Interval** özelliği ile belirlenen aralıkta çalıştırılacak metodun referansını tutmaktadır. Metodun referansı Timer sınıfının içindeki delegate(temsilci) ve olay(event) veri yapıları ile sağlanmaktadır. (Delegate ve Event veri yapıları ile ilgili ayrıntılı yazılar ileriki zamanlarda yayınlanacaktır.)

Şimdi isterseniz Timer sınıfının yapıcı metotlarını inceleyelim.Timer sınıfının iki tane yapıcı metodu vardır. Bu metotlardan birincisi herhangi bir parametre almamaktadır. İkincisi ise double türden bir parametre almaktadır. Double türden olan bu parametre Timer nesnesinin Elapsed olayının ne kadar sürede meydana geleceğini belirtmektedir. Bu metodun prototipi aşağıdaki gibidir.

```
public Timer(double TickSayısı)
```

TickSayısı milisaniye cinsindendir. Örneğin her 1 saniyede çalışmasını istediğimiz bir metot için TickSayısı 1000 olan bir Timer kullanmamız gerekir. Saniyede on defa çalışmasını istediğimiz bir metot için ise TickSayısı 100 olan bir Timer nesnesi kullanmalıyız. Eğer yapıcı metot herhangi bir parametre ile kullanılmaz ise varsayılan TickSayısı olan 100 ile Timer nesnesi oluşturulur.

Çalışma zamanında timer nesnelerinin TickSayısını değiştirmek yada Tick sayısını öğrenmek için Interval özelliği kullanılabilir. Timer nesneleri varsayılan olarak aktif değildir. Bu yüzden Timer nesnelerinin işleyebilmesi için bool türden olan Enabled özelliğinin true olarak ayarlanması gerekir. Yada Timer sınıfının Start() metodunu kullanarak zamanlayıcıyı başlatmanız gerekmektedir.

Timer sınıfı ile ilgili en önemli nokta Interval ile belirtilen süre dolduğunda hangi metodu çağıracağını nerden anlayacağıdır. Bunun için temsilciler ve olaylar kullanılır. Timer sınıfı tasarlanırken yapısında tanımlanan event ve delegate veri türleri ile bu mümkün olmaktadır. Timer sınıfının Elapsed olayı bu işe yaramaktadır. Elapsed olayına += operatörü ile olay yöneticisi yardımıyla(event handler) yeni metotlar ekleyebiliriz. Timer sınıfının Elapsed olayı için ElapsedEventHandler isminde özel bir temsilci tanımlanmıştır. Bu temsilci nesnesine parametre olarak verilecek metodun parametreik yapısı aşağıdaki gibi olmalıdır.

```
SaniyelikIs(object o, ElapsedEventArgs a)
```

ElapsedEventArgs sınıfı Timer nesnesi ile ilgili ayrıntılı bilgiler tutmaktadır. Örneğin ElapsedEventArgs sınıfının SignalTime özelliği ile Elapsed olayının meydana geldiği saati ve tarihi öğrenebiliriz. Örneğimizi verdikten sonra ElapsedEventArgs sınıfını daha iyi öğreneceksiniz.

Timer sınıfının diğer bir önemli elemanı ise Stop() metodudur. Bu metodu çağırdığımız anda Timer nesnesinin Elapsed olayı artık meydana gelmez. Stop() metodu yerine Timer nesnesinin Enabled özelliğini false yapmak ta diğer bir çözümdür.

Timer sınıfı ile ilgili bu bilgileri verikten sonra bir örnek ile bilgilerimizi pekiştirelim. Örneğimiz de her saniye de saati ekrana yazdıracak bir metod bildireceğiz. Bu metod bir Timer sayesinde her saniye çağrılacaktır. Örneğin kaynak kodu aşağıdaki gibidir.

```
using System;
using System.Timers;

class Zamanlayici
{
    static void Main()
    {
        Timer t = new Timer(1000);
        t.Elapsed += new ElapsedEventHandler(SaniyelikIs);
        t.Start();

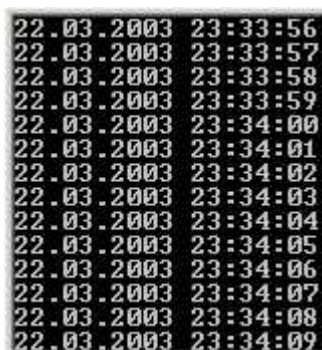
        while(true)
        {

        }

    }

    static void SaniyelikIs(object o, ElapsedEventArgs a)
    {
        Console.WriteLine(DateTime.Now);
    }
}
```

Timer sınıfının işlevini görebilmek için programımızın en az beş on saniye çalışmasına devam etmesi gerekir. Bunun için while(true) şeklinde sonsuz bir döngü kurduk. Eğer bu sonsuz döngü olmasaydı programın icrası bir saniyeden kısa bir sürede biteceği için Elapsed olayı meydana gelmeyecekti. Programın çalışması sırasında elde edilmiş bir ekran görüntüsü aşağıdaki gibidir.



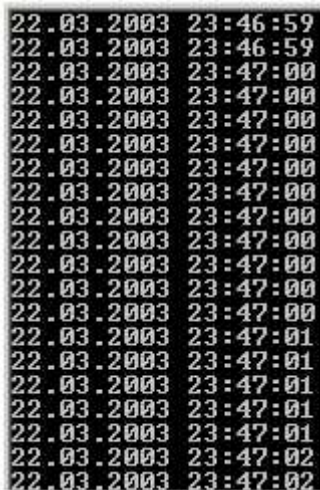
```
22.03.2003 23:33:56
22.03.2003 23:33:57
22.03.2003 23:33:58
22.03.2003 23:33:59
22.03.2003 23:34:00
22.03.2003 23:34:01
22.03.2003 23:34:02
22.03.2003 23:34:03
22.03.2003 23:34:04
22.03.2003 23:34:05
22.03.2003 23:34:06
22.03.2003 23:34:07
22.03.2003 23:34:08
22.03.2003 23:34:09
```

Ekran çıktısından da görüldüğü üzere her saniyede ekrana yeni saat değeri yazıldığı için programımız bir saat gibi çalışmaktadır.

SaniyelikIs() metodunu `ElapsedEventArgs` sınıfından dolayı aşağıdaki gibi de yazabilirdik. `ElapsedEventArgs` sınıfının `SignalTime` özelliği `Elapsed` olayının meydana geldiği zamanı vermektedir.

```
static void SaniyelikIs(object o, ElapsedEventArgs a)
{
    Console.WriteLine(a.SignalTime);
}
```

Timer sınıfını kullanırken karşılaşılabileceğimiz en büyük sorunlardan biri `Elapsed` olayının meydana gelme aralığının olaydan sonra çalıştırılacak kodların icrası için geçen zamandan az olmasında görülür. Bu sorun, ikinci `Elapsed` olayı meydana geldiğinde birinci `Elapsed` olayına ilişkin kodlar halen çalışıyor durumda olmasından kaynaklanır. Örneğimizdeki Timer nesnesinin Tick sayısını 1000 yerine 100 yaptığımızda aynı saat değerinin 10 defa ekrana yazdırılması gerektiğini anlarız. Çünkü `Elapsed` olayı her saniyede 10 defa gerçekleşecektir. Her 10 `Elapsed` olayı gerçekleştiğine bir saniye geçeceği için ekrana aynı saat değeri 10 defa yazmalıdır diye düşünürüz. Ama bunun gerçekte böyle olmadığını görürüz. Tick sayısını 100 yaptığımızda programın ekran çıktısı aşağıdaki gibi olmaktadır.



```
22.03.2003 23:46:59
22.03.2003 23:46:59
22.03.2003 23:47:00
22.03.2003 23:47:00
22.03.2003 23:47:00
22.03.2003 23:47:00
22.03.2003 23:47:00
22.03.2003 23:47:00
22.03.2003 23:47:00
22.03.2003 23:47:00
22.03.2003 23:47:00
22.03.2003 23:47:00
22.03.2003 23:47:00
22.03.2003 23:47:00
22.03.2003 23:47:01
22.03.2003 23:47:01
22.03.2003 23:47:01
22.03.2003 23:47:01
22.03.2003 23:47:01
22.03.2003 23:47:01
22.03.2003 23:47:01
22.03.2003 23:47:01
22.03.2003 23:47:02
22.03.2003 23:47:02
```

Ekran çıktısında bazı saat değerlerinin daha fazla sayıda bazılarının ise daha az sayıda yazıldığını görüyorsunuz. Bu demek oluyorki programın bellek durumuna ve işlemcinin yoğunluğuna göre `Elapsed` olayının meydana gelme sayısı değişmektedir. Bunu önlemenin yolu tick sayısının, `Elapsed` olayından sonra çalışacak metod yada metodların toplam icra süresinde daha fazla olacak şekilde ayarlamaktır.

`Elapsed` olayına birden fazla metotodu ilişkilendirebiliriz. Örneğin `Elapsed` olayı her saniyede bir meydana geldiğinde ekrana ayrıca "Merhaba `Elapsed` olayı!" yazdırmak için aşağıdaki programı yazabilirsiniz.

```
using System;
using System.Timers;

class Zamanlayici
{
    static void Main()
    {
        Timer t = new Timer(1000);
        t.Elapsed += new ElapsedEventHandler(SaniyelikIs);
        t.Elapsed += new ElapsedEventHandler(Selam);
        t.Start();
    }
}
```

```

        while(true)
        {

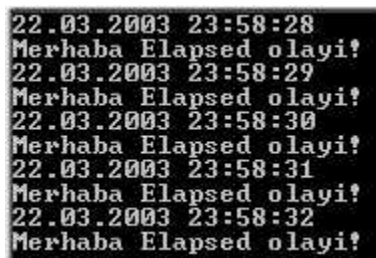
        }
    }

    static void SaniyelikIs(object o, ElapsedEventArgs a)
    {
        Console.WriteLine(DateTime.Now);
    }

    static void Selam(object o, ElapsedEventArgs a)
    {
        Console.WriteLine("Merhaba Elapsed olayı!");
    }
}

```

Bu programda Elapsed olayı meydana geldiğinde ilk önce SaniyelikIs() metodu ardından da Selam() metodu çağrılacaktır. Bunun sebebi olayla ilişkilendirilen metotların olay yöneticisine eklenme sırasıdır. İlk eklenen metot ilk çağrılır. Bu programın ekran çıktısı ise aşağıdaki gibi olacaktır.



```

22.03.2003 23:58:28
Merhaba Elapsed olayı!
22.03.2003 23:58:29
Merhaba Elapsed olayı!
22.03.2003 23:58:30
Merhaba Elapsed olayı!
22.03.2003 23:58:31
Merhaba Elapsed olayı!
22.03.2003 23:58:32
Merhaba Elapsed olayı!

```

Bu yazının sonuna geldik. Size Timer sınıfının kullanımını ve dikkat etmeniz gereken bazı noktaları aktarmaya çalıştım. Yazı ile ilgili düşünceleriniz elektronik posta adresime yazabilirsiniz.

C#'ta Gösterici(Pointer) Kullanmak – II

C#'ta göstericilerin kullanımına yönelik ilk yazıda göstericilere giriş yapmıştık, C#'ta göstericilerin kullanımını 3 yazılıklı bir seri halinde anlatmayı düşündüm. Bu yazıda gösterici aritmetiğini ve fixed anahtar sözcüğünün kullanımını öğreneceğiz.

Göstericilerin adres bileşenlerine sabit tamsayı değerleri ekleyebiliriz, aynı şekilde göstericilerin adres bileşeninden sabit bir tamsayı değerini çıkarabiliriz. Ancak göstericilere uygulanan bu toplama ve çıkarma işlemleri biraz farklıdır. Göstericilere sabit değerlerin eklenmesi yada bir değerın göstericiden çıkarılması göstericideki tür bileşeni ile yakından ilgilidir. Bir göstericinin değerini bir artırmak göstericinin adres bileşenini, göstericinin türünün içerdiği byte sayısı kadar artırmak demektir. Aynı kural çıkarma işlemi içinde geçerlidir. Örneğin int türünden bir gösterici ile 1 sayısını toplamak göstericinin adres bileşenini 4 artırmak anlamına gelir. Çünkü int türü 4 byte büyüklüğündedir. Aynı şekilde int türden bir göstericiden 1 sayısını çıkarmak göstericinin adres bileşenini 4 eksiltmek anlamına gelir. Göstericilerle yapılan bu tür aritmetik işlemlerin tamamına gösterici aritmetiği denilmektedir.

Gösterici aritmetiğini daha yakından görmek için aşağıdaki programı yazın ve sonucunu inceleyin.

```
using System;

class Gosterici
{
    unsafe static void Main()
    {
        int* ptr1 = (int*)500;
        char* ptr2 = (char*)500;
        double* ptr3 = (double*)500;
        byte* ptr4 = (byte*)500;

        ptr1 += 2;
        ptr2 += 5;
        ptr3 += 2;
        ptr4 += 6;

        Console.WriteLine((uint)ptr1);
        Console.WriteLine((uint)ptr2);
        Console.WriteLine((uint)ptr3);
        Console.WriteLine((uint)ptr4);
    }
}
```

Programı **/unsafe** argümanı ile derleyip çalıştırdığımızda aşağıdaki ekran görüntüsünüz elde ederiz. Programda Main() metodunun unsafe olarak işaretlendiğine dikkat edin.

```
508
510
516
506
```

Programın çıktısından da görüldüğü üzere int türden bir göstericiye 2 sayısını eklemek göstericinin adres bileşenini $2 \times 4 = 8$ kadar artırmıştır. Aynı şekilde char türden bir göstericiye 5 değerini eklemek göstericinin adres bileşenini $5 \times 2 = 10$ kadar artırmıştır. Toplama yerine çıkarma işlemi yapılmış olsaydı bu sefer aynı oranda adres bileşeni eksiltilmiş olacaktı.

Göstericiler üzerinde sadece tamsayılarla aritmetik işlemler yapılabilir. Göstericiler ile aşağıdaki aritmetik operatörleri kullanabiliriz.

`+, -, --, ++, -=, +=`

void göstericilerde herhangi bir tür bilgisi olmadığı için bu tür göstericiler üzerinde aritmetik işlemler yapılamaz. Çünkü void türünden bir göstericiye örneğin 1 eklemek istediğimizde göstericinin adres bileşeninin kaç byte öteleneyeceği belli değildir.

Göstericiler üzerinde yapılabilecek diğer önemli işlemde iki göstericinin birbirinden çıkarılmasıdır. Gösterici türleri aynı olmak şartıyla iki göstericiyi birbirinden çıkarabiliriz. Ancak iki göstericinin çıkarılması sonucunda üretilen değer bir gösterici türü değildir. İki gösterici arasındaki fark, adres bileşenlerinin sayısal farkının gösterici türlerinin büyüklüğünden kaç adet byte miktarı edeceğidir. Diğer bir deyişle adres bileşenlerinin sayısal farkı alınıp gösterici türünün byte miktarına göre bir değer belirlenir. İki göstericinin farkı long türden bir değer üretir. İki göstericinin farkına örnek verecek olursak, int türden 5008 adres ile int türden 5000 adresinin farkı $(5008-5000) \% \text{sizeof(int)}$ tir. Yani sonuç long türden 2 dir. Aşağıdaki programı yazarak sonucu görebilirsiniz.

```
using System;

class Gosterici
{
    unsafe static void Main()
    {
        int* ptr1 = (int*)500;
        int* ptr2 = (int*)508;

        long fark=ptr2 - ptr1;

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

Diğer bir ilginç nokta iki göstericinin adres bileşenlerinin farkı gösterici türlerinin büyüklüğünün tam kati olmadığında görülür. Örneğin ptr2 göstericisini tanımlanmasını

```
int * ptr2 = (int*)507;
```

şeklinde değiştirdiğinizde bu sefer ekrana 1 yazdığını görürsünüz. Burdan çıkarmamız gereken sonuç iki göstericinin farkı adres bileşenlerinin sayısal farkının olmamasıdır.

Dikkat: İki göstericinin farkı long türden bir değer üretir. Bu yüzden iki göstericinin farkı açıkça bir tür dönüşümü yapılmadıkça long türünden küçük türden olan değişkenlere atanamaz.

Göstericiler ile kullanılacak diğer operatörler ise `==`, `<` ve `>` gibi karşılaştırma operatörleridir. Bu operatörler iki göstericinin adres bileşenini karşılaştırıp true yada false değeri üretirler. Karşılaştırma operatörleri göstericiler için çok istisnai durumlar dışında anlamlı değildir. Bu istisna durumlardan biri göstericileri kullanarak dizi işlemleri yaptığımızda görülür.

fixed Anahtar Sözcüğü

Bildiginiz gibi C#' ta tanımladığımız referans değişkenleri heap bellek bölgesindeki adresler temsil ederler. Ancak biz adresler yerine nesnenin ismini kullanırız. Gereksiz bilgi toplayicisi(garbage collector) bellek optimizasyonui açısından heap bellek bölgesindeki nesnelerin yerlerini her an değiştirebilir. Bu yer değişiminden bizim haberimiz olmaz, çünkü nesnenin yeri değiştiği anda bu nesneye referans olan stack bellek bölgesindeki değişkenin adres bileşeni de değiştirilir. Dolayısıyla biz aynı referans ile farklı bellek bölgesini istediğimiz dışında kullanmış oluruz. Ancak bazı durumlarda gereksiz nesne toplayicisine bir nesnenin adresini değiştirmemesi için ikna etmek durumunda kalırız. Bu, özellikle sınıf nesnelerinin üye elemanlarından birinin adresi ile işlem yapmamız gerektiği durumlarda karşımıza çıkar. Bir değişkenin adresinin belirlenen bir faaliyet alanı boyunca değişmeden kalması için bunu gereksiz nesne toplayicisine bildirmemiz gerekir. Bunun için `fixed` anahtar sözcüğü kullanılır.

Zaten `fixed` anahtar sözcüğünü kullanmadan referans türünden nesnelerin üye elemanlarının adreslerini elde etmemiz mümkün değildir. Üye elemanlarının adreslerini elde edemediğimiz bu tür nesnelere `managed type`(yönetilen tip) denilmektedir. Buna göre sınıflar `managed type` kapsamına girmektedir.

Aşağıdaki programda `ManagedType` isimli sınıfın `int` türünden olan `x` elemanının adresi bir göstericiye atanmak isteniyor.

```
using System;

class ManagedType
{
    public int x;
    public ManagedType(int x)
    {
        this.x = x;
    }
}

class Gosterici
{
    unsafe static void Main()
    {
        ManagedType mt = new ManagedType(5);

        int* ptr1 = &(mt.x);
    }
}
```

`ManagedType` sınıfının `x` elemanı `değer` tipi olmasına rağmen `mt` nesnesi üzerinden `x` değişkeninin adresi elde edilememektedir. Çünkü `x` değişkeninin adresi gereksiz nesne toplayicisi tarafından her an değiştirilebilir. Eğer yukarıdaki kod geçerli olmuş olsaydı `x` değişkeninin adresi değiştiği anda `ptr1` göstericisi nereye ait olduğu bilinmeyen bir adres bilgisi taşıyor olacaktı. `x` değişkeninin bir blok içerisinde sabit adreste olmasını istiyorsak aşağıdaki gibi `fixed` anahtar sözcüğünü kullanmalıyız.

```
using System;

class ManagedType
{
    public int x;
    public ManagedType(int x)
    {
        this.x = x;
    }
}
```

```

    }
}

class Gosterici
{
    unsafe static void Main()
    {
        ManagedType mt = new ManagedType(5);

        fixed(int* ptr1 = &(mt.x))
        {
            //x'in adresi bu blokta asla degismez.
        }
    }
}

```

Yukarıdaki fixed ile işaretlenmiş blokta x'in adresinin değişmeyeceği garanti altına alınmıştır. Birden fazla değişkeni fixed olarak işaretlemek için aşağıdaki gibi bir kullanım geçerli kılınmıştır.

```

ManagedType mt1 = new ManagedType(5);
ManagedType mt2 = new ManagedType(5);

fixed(int* ptr1 = &(mt1.x))
fixed(int* ptr2 = &(mt2.x))
{
    //x'in adresi bu blokta asla degismez.
}

```

Öte yandan bir fixed bildirimi içinde adreslerinin değişmesini istemediğimiz elemanları virgül ile ayırarak aşağıdaki gibi de bildirebiliriz.

```

ManagedType mt1 = new ManagedType(5);
ManagedType mt2 = new ManagedType(5);

fixed(int* ptr1 = &(mt2.x), ptr2 = &(mt2.x))
{
    //x'in adresi bu blokta asla degismez.
}

```

Göstericilerle ilgili son yazıda, yapı göstericileri, göstericiler ile dizi işlemleri ve **stackalloc** ile dinamik alan tahsisatı yapma gibi konuları inceleyeceğiz.

C# ile Nesne Yönelimli Programlama I

Bu makalede nesne yönelimli programlama tekniğine kadar kullanılan yazılım geliştirmedeki yaklaşımlara göz atacağız. Daha sonra nesne yönelimli programlamanın temel kavramları ve neden böyle bir tekniğin kullanıldığı üzerinde duracağız.

Yazılım Geliştirme ve Bu Alandaki Yaklaşımlar

Kimilerine göre geçtiğimiz yüzyılın en önemli buluşu olarak kabul edilen bilgisayar teknolojisi, baş döndürücü bir hızla gelişmektedir. Bilişim sektöründeki değişimler bazen varolan teknolojilere yenilerinin eklenmesi şeklinde olabilir. Diğer taraftan bir kısım yenilikler vardır ki bu alanda büyük değişimlere ve evrimlere yolaçar. Mesela; kişisel bilgisayarların kullanılmaya başlanması veya internetin belli başlı akademik kurumların ve askeri organizasyonların tekelinden alınıp tüm insanlığın hizmetine sunulması gibi.

Hepimizin bildiği gibi bir bilgisayar sistemi iki ana parçadan oluşur. Bunlar donanım(hardware) ve yazılım(software). Donanımın yazılım ile uyumlu çalışması sonucunda sistemlerimiz sorunsuz bir şekilde bizlere hizmet verirler. Ayrıca donanımın amacımızda uygun hizmet vermesi uygun yazılımın geliştirilip kullanılmasına bağlıdır.

Yazılım sektöründe program geliştirme konusunda günümüze kadar bir çok yaklaşım denenmiştir. Bunların ilki programın baştan aşağıya sırası ile yazılıp çalıştırılmasıdır. Bu yaklaşımla BASIC dili kullanılarak bir çok program yazıldığını biliyoruz. Burda sorun programın akışı sırasında değişik kısımlara **goto** deyimi ile atlanmasıdır. Program kodu bir kaç bin satır olunca, kodu okumak ve yönetmek gerçekten çok büyük sorun oluyordu.

İkinci yaklaşım ise prosedürel yaklaşımdır. Programlarda bir çok işin tekrar tekrar farklı değerleri kullanılarak yapıldığı farkedildi. Mesela herhangi bir programda iki tarih arasında ne kadar gün olduğunu bulmak birçok kez gerek olabilir. Bu durumda başlangıç ve bitiş tarihlerini alıp aradaki gün sayısını veren bir fonksiyon yazılabilir ve bu fonksiyon ihtiyaç duyulduğu yerde uygun parametrelerle çağrılıp istenen sonuç elde edilebilir. Prosedürel yaklaşım Pascal ve C dillerinde uzun yıllar başarı ile kullanılmıştır.

Ama her geçen gün programların daha karmaşık bir hal alması, program kodunun kurumsal uygulama projelerinde onbinlerce satırı bulması ve yazılım geliştirme maliyetinin çok arttığını gören bilim adamları, programcılara yeni bir yaklaşımın kullanılabilineceğini öğretiler. Bu yaklaşımın ismi Nesne Yönelimli Programlama(Object Oriented Programlama)dır.

Nesne yönelimli programlama tekniği, diğer yaklaşımlara nazaran, yazılım geliştiren insanlara büyük avantajlar sağlamaktadır. Birincisi karmaşık yazılım projelerinin üretilmesini ve bakımını kolaylaştırıyor olmasıdır. Diğerisi ise program kodunun tekrar kullanılabilmesine (code-reusability) olanak sağlamasıdır. Bu noktada program kodunun tekrar kullanılabilmesi profesyonel yazılım şirketlerinin maliyetlerini azaltmıştır. Dolayısı ile programların lisans ücretleri düşmüş ve sektörün sürekli olarak canlı kalmasına ve rekabet içinde gelişmesine yardımcı olmuştur.

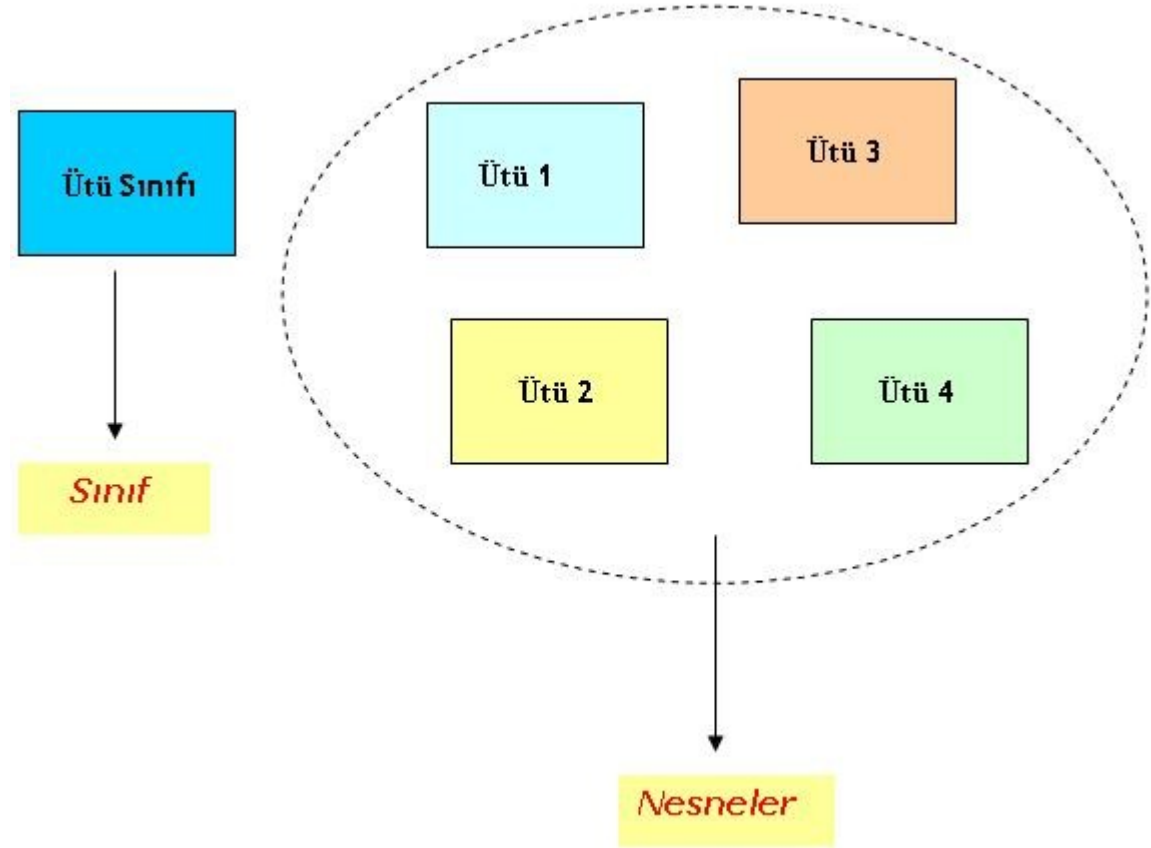
Nesne Yönelimli Programlama Nedir?

Nesne yönelimli programlamada esas olan, gerçek hayatta varolan olguların programlamaya aktarılmasındaki yeni yaklaşımdır. Prosedürel programlamada verilerimiz ve fonksiyonlarımız vardı. Yani veri ve bu veriyi işleyen metodlar etrafında dönüyordu herşey.

Aslında nesne yönelimli programlamada da iki önemli birim veri ve veriyi işleyip mantıklı sonuçlar üreten metodlar bulunur. Ama burdaki fark gerçek hayattaki olguların da daha iyi gözlenip programlama dünyasına aktarılmasındadır.

Mesela elimizde bir ütümüz olsun. Ütünün markası, modeli, rengi, çalıştığı elektrik voltajı, ne tür kumaşları ütüleyebildiği bu ütüye ait özelliklerdir (veri). Aynı zamanda ütümüzü ısıtabiliriz, ütüleme işinde kullanabiliriz ve soğumaya bırakabiliriz. Bunlar ise ütünün fonksiyonlarıdır(metod). Eğer ütü ile ilgili bir program yapmış olsak ve nesne yönelimli programlama tekniğini kullansak hemen bir ütü sınıfı(class) oluştururduk. Bu sınıfta ütüye ait bilgiler (veriler) ve ütü ile yapabileceğimiz işler(metod) bulunurdu. O zaman nesne yönelimli programlama da bir sınıfta, sınıfa ait veriler ve bu verileri işleyip bir takım faydalı sonuçlar üreten fonksiyonlar/metodlar bulunur.

Dahası, biz birtane ütü sınıfı tasarlırsak bu sınıftan istediğimiz sayıda değişik ütüler(object veya instance) yapabiliriz. Aşağıdaki şekilde ütü sınıfı ve bu sınıftan oluşturduğumuz nesnelerin görsel olarak anlatımı bulunmaktadır.



Sınıf Nedir ve Nasıl Tasarlanır?

Az önce de değindiğimiz gibi, sınıf bir yazılım kurgusudur ve gerçek hayattaki herhangi bir olguyu modelleyen ve bu olguya ait özellikleri(veri) ve davranışları(metodlar) tarifleyen yapıdır.

İsterseniz ütü ile ilgili bir örnek üzerinde çalışalım. Aşağıdaki örnek C# program kodunu lütfen dikkatlice inceleyiniz.

```
using System;  
  
class Ütü_Örneği  
{
```

```

static void Main(string[] args)
{
    Ütü üt1= new Ütü("Beyaz", "AyBakır");

    Console.WriteLine(üt1.Isın(70));
    Console.ReadLine();
}

class Ütü
{
    public int sıcaklık;
    public string renk;
    public string marka;

    public Ütü(string renk, string marka)
    {
        sıcaklık=15;
        this.renk=renk;
        this.marka= marka;

        Console.WriteLine(sıcaklık+ "derece sıcaklığında,\n "
            + renk + " renginde ve\n "
            + marka +" markasıyla bir ütü nesnesi oluşturuldu\n\n");
    }

    public string Isın(int derece)
    {
        sıcaklık+=derece;
        return "şu an sıcaklığım: " + sıcaklık+ " derece";
    }
}

```

Yukarıdaki örnek programımızda önce altta bulunan **Ütü** sınıfımızı inceleyelim. Bu sınıfın sıcaklık, renk ve marka olmak üzere üç adet verisi vardır. Ayrıca Ütü sınıfımızın Ütü(string renk, string marka) şeklinde tanımlanmış yapılandırıcısı (constructor) vardır. Yapılandırıcılar bir sınıftan oluşturulan nesnelerin ilk değerlerini atama ve başlangıç işlemlerini yapmak için kullanılırlar. Ütü sınıfımızın yapılandırıcısı, oluşturulan her ütü nesnesinin sıcaklığını varsayılan değer olarak 15 dereceye ayarlıyor. Ayrıca parametre olarak alınan renk ve marka değerlerini de atayıp, ütüye ait özellikleri ekrana yazdırıyor.

Ütü sınıfına ait olan diğer metod ise Isın(int derece) olarak tanımladığımız metoddur. Bu metod ütünün sıcaklığını derece parametresinde verilen değer kadar artırıp sonucu string tipinde geri dönderiyor.

Ütü_Örneği sınıfına geri dönersek, burda sadece Main() metodunun bulunduğunu görürüz. Main() metodu, C# dilinde her programda bulunması gereken zorunda bir metoddur. Çünkü programın çalıştırılması buradan başlar. Ayrıca her C# programında sadece bir tane Main() metodu bulunur. Örneğimizde Main() metodundaki en önemli satır:

```
Ütü üt1= new Ütü("Beyaz", "AyBakır");
```

satırıdır. Bu satırda Ütü sınıfına ait üt1 nesnemizi oluşturuyoruz. Yukarıdaki satırdan da görebileceğimiz gibi herhangi bir sınıfa ait yeni nesneyi oluştururken genel olarak şu yapı kullanılır:

```
SınıfAdı nesneAdı = new SınıfAdı(parametre1, parametre2, ... parameter N)
```

Referans Tipleri ve Nesneler

Nesnelerin en önemli özelliği referans tipinde olmalarıdır. Referans tiplerinde bir nesnenin değeri saklanmaz. Sadece nesnenin hafızadaki (heap memory) yeri saklanır. Bir nesnenin hafızadaki yerini new operatörü geri dönderir. Aşağıdaki program kodu ile referans tiplerine ait bir kaç temel özelliği inceleyelim:

```
using System;

class Ütü_Örneği
{
    static void Main(string[] args)
    {
        Ütü üt1= new Ütü("Beyaz", "AyBakır");

        Console.WriteLine("Ütü1'nin ilk özellikleri: " +
                           üt1.Özellikler());
        Console.WriteLine("Ütü1'i 50 derece ısıtalım: " +
                           üt1.Isin(50)+ "\n\n");

        // üt2 nesnemizi renksiz ve markasız olarak oluşturalım:
        Ütü üt2 = new Ütü("", "");

        //Ütü2 nesnemizin özelliklerine bir göz atalım:
        Console.WriteLine("Ütü2'nin ilk özellikleri: " +
                           üt2.Özellikler());

        üt2= üt1; // üt2 nesnemizi üt1 nesnemize atıyoruz.
        Console.WriteLine("Ütü2'nin özellikleri: " +
                           üt2.Özellikler());

        Console.ReadLine();
    }
}

class Ütü
{
    public int sıcaklık;
    public string renk;
    public string marka;

    public Ütü(string renk, string marka)
    {
        sıcaklık=15;
```

```

        this.renk=renk;
        this.marka= marka;
    }

    public string Isın(int derece)
    {
        sıcaklık+=derece;
        return "şu an sıcaklığım: " + sıcaklık+ " derece";
    }

    public string Özellikler()
    {
        string sonuc= " Sıcaklık: " + sıcaklık+
                      "\n Renk: " + renk+
                      "\n Marka: " + marka+ "\n\n";
        return sonuc;
    }
}

```

isterseniz kodumuzu incelemeye yine Ütü sınıfından başlayalım. Bir önceki örneğe göre sınıfımızda değişiklikler yaptık. Ütü sınıfının özelliklerini gösteren ayrı bir metod yazdık. Bu metod Özellikler() isimli olsun. Doğal olarak Ütü sınıfının yapılandırıcısında, sınıf oluşturulduktan hemen sonra nesnenin özelliklerini gösteren kısım kaldırıldı.

şimdi ise Ütü_Örneği sınıfındaki Main() metodu üzerinde yoğunlaşalım. Yine, birinci örnekte olduğu gibi, ütü1 nesnemizi

```
Ütü ütü1= new Ütü("Beyaz", "AyBakır");
```

ile oluşturuyoruz. Sonra ütü1 nesnesinin özelliklerini ekrana yazdırıyoruz. Bir sonraki satırda ise ütü1 nesnemizi 50 derece ısıtıp sonucu ekrana yazıyoruz.

Artık ikinci bir ütü nesnesinin oluşturmanın zamanı geldiğini düşünüp

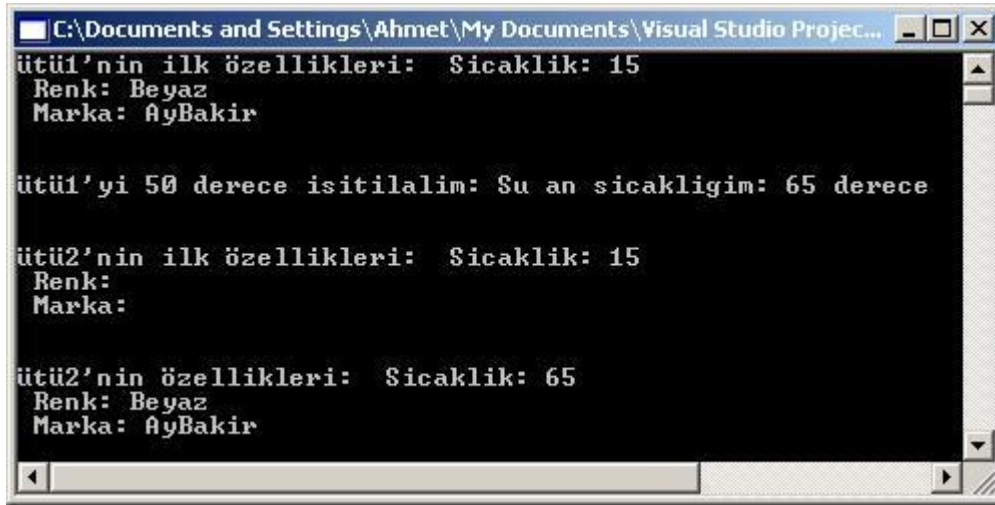
```
Ütü ütü2 = new Ütü("", "");
```

satırında ütü2 nesnemizi oluşturuyoruz. Burada dikkatinizi çektiği gibi ütü2 nesnesinin renk ve marka özelliklerini boş olarak (null) oluşturuyoruz. ütü2 nesnesini oluşturunca özelliklerine göz atmak için

```
Console.WriteLine("ütü2'nin ilk özellikleri: " +
                  ütü2.Özellikler());
```

satırlarını yazıyoruz. Bir sonraki satır programımızın en önemli satırıdır. Bu satırda **ütü2= ütü1** ifadesi ile ütü2 nesnesinin referansını ütü1 nesnesinin referansına atıyoruz. Değer tiplerinde "=" operatörü ile bir değişkenin değeri diğerine kopyalanır. Ama nesneler referans tipleri oldukları için, "=" operatörü bir nesnenin referansı diğerine atar. Bu durumda ütü2 nesnesi artık ütü1'in elemanlarının adreslerini içerir. Bu andan sonra ütü2'nin eskiden gösterdiği sıcaklık, renk ve marka verilerine erişmemiz mümkün değildir. Artık onlar hafızada bir yerde ve biz bilmiyoruz. Bu verilerin hafızada kapladıkları alanı .NET Framework'nun çöp toplayıcısı (garbage collector) uygun gördüğü bir anda temizleyecektir.

ütü2 nesnesinin son durumda özelliklerine bakacak olursak gerçektende ütü1'i referans ettiğini görürüz. Yukarıdaki programımızı çalıştırdığımızda aşağıdaki sonucu elde ederiz:



```
C:\Documents and Settings\Ahmet\My Documents\Visual Studio Projec...
ütü1'nin ilk özellikleri: Sıcaklık: 15
Renk: Beyaz
Marka: AyBakır

ütü1'yi 50 derece ısıtılalım: Şu an sıcaklığım: 65 derece

ütü2'nin ilk özellikleri: Sıcaklık: 15
Renk:
Marka:

ütü2'nin özellikleri: Sıcaklık: 65
Renk: Beyaz
Marka: AyBakır
```

Son olarak şu ilginç soruya cevap vermeye çalışalım. "Bir önceki program kodunun sonuna aşağıdaki kısmı eklersek ne gibi bir değişiklik olur?"

```
// ütü2 nesnemizin özelliklerini değiştiriyoruz:
ütü2.sıcaklık=30;
ütü2.renk="Kırmızı";
ütü2.marka="Dolu";

//ütü1 nesnemizin özelliklerine bir göz atalım:
Console.WriteLine("ütü1'nin son durumdaki özellikleri: " +
    ütü1.Özellikler());

//ütü2 nesnemizin özelliklerine bir göz atalım:
Console.WriteLine("ütü2'nin son durumdaki özellikleri: " +
    ütü2.Özellikler());
```

Yukarıdaki kodun eklenmiş halini derleyip çalıştırmayı deneyelim. Sonuç aşağıdaki gibi olacaktır:

```
C:\Documents and Settings\Ahmet\My Documents\Visual Studio Projects\Üt...
ütü1'nin ilk özellikleri:  Sicaklik: 15
  Renk: Beyaz
  Marka: AyBakir

ütü1'yi 50 derece isitilalım: Su an sicakligim: 65 derece

ütü2'nin ilk özellikleri:  Sicaklik: 15
  Renk:
  Marka:

ütü2'nin özellikleri:  Sicaklik: 65
  Renk: Beyaz
  Marka: AyBakir

ütü1'nin son durumdaki özellikleri:  Sicaklik: 30
  Renk: Kirmizi
  Marka: Dolu

ütü2'nin son durumdaki özellikleri:  Sicaklik: 30
  Renk: Kirmizi
  Marka: Dolu
```

Evet ütü1 ve ütü2'nin son durumdaki özellikleri aynıdır. Bu durumdan şu sonuca ulaşabiliriz: " ütü1 ve ütü2'nin herhangi biri ile özellikleri değiştirsek diğeri de aynı özelliklere sahip olur. Çünkü her iki nesne de hafızada aynı yere referans ediyorlar."

Bu makalede nesne yönelimli programlamaya giriş yaptık. Nesne yönelimli programlamanın iki temel kavramı olan sınıf(class) ve nesne(object)'yi inceledik. Sonraki makalelerde nesne yönelimli programlamanın diğer önemli özelliklerini birlikte incelemek dileğiyle.

C#'ta Gösterici(Pointer) Kullanmak – III

C#'ta göstericilerin kullanımı ile ilgili yazı dizisinin son bölümü olan bu yazıda göstericiler ile dizi işlemlerinin nasıl yapıldığı, stackalloc ile dinamik bellek tahsisatının yapılmasını ve son olarak yapı(struct) göstericilerinin kullanılışını inceleyeceğiz.

Göstericiler ile Dizi İşlemleri

C#'ta tanımladığımız diziler System.Array sınıfı türündendir. Yani bütün diziler **managed type** kapsamına girerler. Bu yüzden tanımladığımız bir dizinin herhangi bir elemanının adresini **fixed** bloğu kullanmadan bir göstericiye atayamayız. Bu yüzden göstericiler ile dizi işlemleri yaparken ya fixed blokları kullanıp gereksiz nesne toplayıcısını uyarmalıyız yada **stackalloc** anahtar sözcüğünü kullanarak kendimiz unmanaged type(yönetilemeyen tip) dizileri oluşturmamız. Her iki durumda birazdan inceleyeceğiz.

Bildiğiniz gibi dizi elemanları bellekte ardışıl bulunur. O halde bir dizinin elemanlarını elde etmek için dizinin ilk elemanının adresini ve dizinin boyutunu bilmemiz yeterlidir. System.Array sınıfı dizilerinin elemanlarına göstericiler yardımıyla rahatlıkla ulaşabiliriz. Bunun için ilk olarak fixed ile işaretlenmiş bir blok içerisinde dizinin ilk elemanının adresini bir göstericiye atamalıyız. Ardından göstericinin değerini bir döngü içerisinde birer birer artırdığımızda her döngüde dizinin bir sonraki elemanına ulaşmış oluruz. Dizilere bu şekilde erişebilmemizi sağlayan ise gösterici aritmetiğidir. Tabi dizilerin elemanlarının bellekte ardışıl bulunması da bu işlemi bu şekilde yapmamızı sağlayan ilk etkidir.

Şimdi yönetilen türden(managed) bir dizinin elemanlarına nasıl eriştiğimizi bir örnek üzerinde inceleyelim.

```
using System;

class Gosterici
{
    unsafe static void Main()
    {
        int[] a = {1,2,3,4};

        fixed(int* ptr = &a[0])
        {
            for(int i=0; i<a.Length; ++i)
                Console.WriteLine(*(ptr+i));
        }
    }
}
```

Programı derlediğinizde dizinin elemanlarının

```
1
2
3
4
```

şeklinde ekrana yazdırıldığını görürsünüz. **(programı derlerken /unsafe argümanını kullanmayı unutmayın)**. Programın kritik noktası

```
fixed(int* ptr = &a[0])
```

satırı ile dizinin ilk elemanının adresinin elde edilmesidir. Zira dizinin diğer elemanlarında sırayla ilk elemandan itibaren her eleman için adres değeri 4 byte artacak şeklindedir. Diğer önemli nokta ise for döngüsü içindeki

```
*(ptr+i)
```

ifadesidir. Bu ifade her döngüde ptr göstericisinin adres bileşeni, döngü değişkeni kadar artırılıyor, sözgelimi döngü değişkeni 1 ise ptr'nin adres bileşeni 4 artırılıyor. Bu da dizinin ikinci elemanının bellekte bulunduğu adrestir. İçerik operatörü ile bu adrese erişildiğinde ise dizinin elemanı elde edilmiş olur.

Gösterici dizileri ile ilgili diğer önemli nokta göstericilerin indeksleyici gibi kullanılabilmesidir. Örneğin yukarıdaki örnekte bulunan

```
*(ptr+i)
```

ifadesini

```
ptr[i]
```

şeklinde değiştirebiliriz. Burdan aşağıdaki eşitlikleri çıkarabiliriz.

```
*(ptr+0) == ptr[0]  
*(ptr+1) == ptr[1]  
*(ptr+2) == ptr[2]  
*(ptr+3) == ptr[3]
```

Dikkat: ptr[i] bir nesne belirtirken (ptr+i) bir adres belirtir.

Dizilerin isimleri aslında dizilerin ilk elemanının adresini temsil etmektedir. Örneğin aşağıdaki programda bir dizinin ilk elemanının adresi ile dizinin ismi int türden bir göstericiye atanıyor. Bu iki göstericinin adres bileşenleri yazdırıldığında sonucun aynı olduğu görülmektedir.

```
using System;  
  
class Gosterici  
{  
    unsafe static void Main()  
    {  
        int[] a = {1,2,3,4};  
  
        fixed(int* ptr1 = a, ptr2 = &a[0])  
        {  
            Console.WriteLine((uint)ptr1);  
            Console.WriteLine((uint)ptr2);  
        }  
    }  
}
```

Programı derleyip çalıştırdığınızda ekrana alt alta iki tane aynı sayının yazıldığını görürsünüz.

Yönetilen(managed) tiplerle çalışmak her ne kadar kolay olsa da bazı performans eksiklikleri vardır. Örneğin bir System.Array dizisinin bir elemanına erişmek ile stack bölgesinde oluşturmamız bir dizinin elemanına ulaşmamız arasında zaman açısından büyük bir fark vardır. Bu yüzden yüksek performanslı dizilerle çalışmak için System.Array sınıfının dışında stack tabanlı diziler oluşturmamız gerekir. Stack tabanlı diziler yönetilemeyen dizilerdir. Bu yüzden bu tür dizileri kullanırken dikkatli olmalıyız. Çünkü her an bize tahsis edilmeyen bir bellek alanı üzerinde işlem yapıyor olabiliriz. Ancak yönetilen dizilerde dizinin sınırlarını aşmak mümkün değildir. Hatırlarsanız bir dizinin sınırları aşılmış çalışırken **IndexOutOfRangeException** istisnai durumu meydana geliyordu. Oysa stack tabanlı dizilerde dizinin sınırları belirli değildir ve tabiki dizinin sınırlarını aşmak kısıtlanmamıştır. Eğer dizinin sınırları aşılmışsa muhtemelen bu işlem bir hata sonucu yapılmıştır. Hiçbir programcı kendisine ait olmayan bir bellek alanında işlem yapmamalıdır. Aksi halde sonuçlarına katlanması gerekir.

Stack tabanlı diziler **stackalloc** anahtar sözcüğü ile yapılır. stackalloc bize istediğimiz miktarda stack bellek bölgesinden alan tahsis eder. Ve tahsis edilen bu alanın başlangıç adresini geri döndürür. Dolayısıyla elimizde olan bu başlangıç adresi ile stackalloc ile bize ayrılmış olan bütün bellek bölgelerine erişebiliriz. stackalloc anahtar sözcüğünün kullanımı aşağıdaki gibidir.

```
int * dizi = stackalloc int[10];
```

Bu deyim ile stack bellek bölgesinde $10 * \text{sizeof}(\text{int}) = 40$ byte'lık bir alan programcının kullanması için tahsis edilir. Bu alan, dizinin faaliyet alanı bitinceye kadar bizim emrimizdedir. Tahsis edilen bu 40 byte büyüklüğündeki bellek alanının ilk byte'ının adresi ise int türden gösterici olan dizi elemanına aktarılır. Dolayısıyla dizi göstericisi ile içerik operatörünü kullandığımızda bize ayrılan 10 int'lik alanın ilk elemanına erişmiş oluruz.

! stackalloc ile tahsis edilen bellek alanlarının ardışıl olması garanti altına alınmıştır.

stackalloc ile alan tahsisatı yapılır. Ancak alan tahsisatı yapılan bellek bölgesi ile ilgili hiçbir işlem yapılmaz. Yani yukarıdaki deyim ile, içinde tamamen rastgele değerlerin bulunduğu 40 byte'lık bir alanımız olur. Bu alandaki değerlerin rastgele değerler olduğunu görmek için aşağıdaki programı yazın.

```
using System;

class Gosterici
{
    unsafe static void Main()
    {
        int * dizi = stackalloc int[10];

        for(int i=0; i<10;++i)
            Console.WriteLine("*(dizi+{0}) = {1}",i,dizi[i]); }
    }
}
```

! Yukarıdaki programda bir gösterici ile bellekteki ardışıl bölgelere indeksleyici operatörü ile nasıl eriştiğimize dikkat edin.

Programı /unsafe argümanı ile beraber derleyip çalıştırdıktan sonra aşağıdaki ekran görüntüsünü elde etmeniz gerekir. Tabi bu değerler rastgele olduğu için sizdeki görüntü tamamen farklı olacaktır. Rastgele değerden kasıt çalışma zamanında Random gibi bir

sınıfın kullanılıp rastgele bir sayı üretilmesi değildir. Burdaki sayılar daha önce çalışmış olan programlardan kalan çöp değerlerdir.

```
C:\Programlar\StackAlloc
*(dizi + 0) = 0
*(dizi + 1) = 1244236
*(dizi + 2) = 1243328
*(dizi + 3) = 1350496
*(dizi + 4) = 124334
*(dizi + 5) = 1287174
*(dizi + 6) = 1243328
*(dizi + 7) = 0
*(dizi + 8) = 0
*(dizi + 9) = 1243404
C:\Programlar\StackAlloc
```

```
Console.WriteLine("*(dizi+{0}) = {1}",i,dizi[i]);
```

satırındaki

`dizi[i]`

yerine

`*(dizi + i)`

yazmamız herhangi birşeyi değiştirmezdi. Daha önce bu iki kullanımın eşdeğer olduğunu belirtmiştik.

Stack tabanlı dizilerle ilgili bilinmesi gereken en önemli nokta dizinin sınırlarının aşılması ile ilgilidir. Daha önceden de denildiği gibi stack tabanlı dizilerin sınırları aşıldığında herhangi bir uyarı verilmez. Elbetteki program başarıyla derlenir ancak çalışma zamanında bize ait olmayan bir adresin içeriğini değiştirmiş oluruz ki bu da bir programcının başına gelebilecek en tehlikeli durumdur. Örneğin aşağıdaki programda stackalloc ile 10 int türünden nesnelik alan tahsis edilmektedir. Buna rağmen istediğimiz kadar alanı kullanabiliyoruz.

```
using System;

class Gosterici
{
    unsafe static void Main()
    {
        int * dizi = stackalloc int[10];

        for(int i=0; i<50;++i)
            *(dizi+i) = i;
    }
}
```

stackalloc ile oluşturacağımız dizilerin boyutu derleme zamanında bilinmek zorunda değildir. Örneğin çalışma zamanında kullanıcının belirlediği sayıda elemana sahip olan bir ardışıl bellek bölgesi aşağıdaki programda olduğu gibi tahsis edilebilir.

```

using System;

class Gosterici
{
    unsafe static void Main()
    {
        Console.Write("Dizi boyutu gir: ");
        uint boyut=0;

        try
        {
            boyut = Convert.ToUInt32(Console.ReadLine());
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }

        int * dizi = stackalloc int[(int)boyut];

        for(int i=0; i<(int)boyut; ++i)
        {
            *(dizi+i) = i;
            Console.WriteLine(dizi[i]);
        }
    }
}

```

Bu program ile çalışma zamanında 7 elemanlı stack tabanlı bir dizinin oluşturulduğunu aşağıdaki ekran görüntüsünden görebilirsiniz.

Yapı(Struct) Türünden Göstericiler

int,char,double gibi veri türleri aslında birer yapıdır. Bu konun başında bütün değer tipleri ile gösterici tanımlayabileceğimizi söylemiştik. C# temel veri türlerinin yanısıra kendi bildirdiğimiz yapılar da değer türündendir. O halde bir yapı göstericisi tanımlayabilmemiz doğal bir durumdur. Yapı göstericilerinin tanımlanması temel veri türlerinden gösterici tanımlama ile aynıdır. Ordaki kuralların tamamı yapılar içinde geçerlidir. Fakat yapı göstericisi tanımlamanın bir şartı vardır, oda yapının üye elemanlarının tamamının değer tipi olma zorunluluğudur. Örneğin aşağıdaki yapı göstericisi tanımlaması geçersizdir.

```

using System;

struct Yapi
{
    int x;
    char c;
    string s;

    public Yapi(int x,char c, string str)
    {
        this.x = x;
        this.c = c;
        this.s = str;
    }
}

```

```

}

class StackAlloc
{
    unsafe static void Main()
    {
        Yapi yapi = new Yapi(2,'a',"Deneme");

        Yapi* pYapi = &yapi;
    }
}

```

Yukarıda 'Yapi' türünden göstericinin tanımlanamamasının sebebi yapının yönetilen türden(managed type) bir üye elemanının bulunmasıdır. Bu üye elemanı da doğal olarak string türüdür. Yapı bildiriminden string türünü çıkarıp aşağıdaki gibi ilgili değişiklikleri yaptığımızda 'Yapi' türünden göstericileri tanımlayabiliriz.

```

using System;

struct Yapi
{
    int x;
    char c;

    public Yapi(int x,char c)
    {
        this.x = x;
        this.c = c;
    }
}

class StackAlloc
{
    unsafe static void Main()
    {
        Yapi yapi = new Yapi(2,'a');

        Yapi* pYapi = &yapi;
    }
}

```

Yapı göstericileri üzerinden yapı göstericisinin adresine ilişkin nesnelerin elemanlarına özel bir operatör olan **-> operatörü** ile erişebiliriz. Örneğin yukarıdaki programın Main() metodunu aşağıdaki gibi değiştirdiğinizde ekrana yapı nesnesinin x ve c elemanları yazdırılacaktır. **Tabi Yapi'nın üye alanlarını public olarak değiştirmeniz gerekecektir. Çünkü ok operatörü ilede olsa ancak public olan elemanlara ulaşabiliriz.**

```

unsafe static void Main()
{
    Yapi yapi = new Yapi(2,'a');

    Yapi* pYapi = &yapi;
}

```



```
Console.WriteLine("yapi.x= " + pYapi->x);
Console.WriteLine("yapi.c= " + pYapi->c);
}
```

Not: ' -> ' operatörüne ok operatörü de denilmektedir.

Yapının public olan elemanlarına ok operatörü yerine yapı nesnesinin içeriğini * operatörü ile elde edip nokta operatörü ile de ulaşabiliriz. Buna göre yukarıdaki Main() metodu ile aşağıdaki Main() metodu eşdeğerdir.

```
unsafe static void Main()
{
    Yapi yapi = new Yapi(2,'a');

    Yapi* pYapi = &yapi;

    Console.WriteLine("yapi.x= " + (*pYapi).x);
    Console.WriteLine("yapi.c= " + (*pYapi).c);
}
```

Her iki Main() metodunun ürettiği çıktı aynıdır.

Göstericilerin en çok kullanıldığı diğer bir uygulama alanı da karakter işlemleridir. Bir yazıyı karakter dizisi olarak temsil edip yazılar ile ilgili işlemler yapılabilir. C#' taki string türünün altında gerçekleşen olaylarda zaten bundan ibarettir. Karakter dizileri ile ilgili en önemli nokta bir yazıyı char türden bir göstericiye atayabilmemizdir. Karakter dizileri olan stringlerdeki her bir karakter bellekte ardışıl bulunmaktadır. Dolayısıyla yazıdaki ilk karakterin adresini bildiğimizde yazıdaki bütün karakterlere erişebiliriz. C#'taki **string** türü yönetilen tip(managed type) olduğu için char türden bir göstericiye bir yazının ilk karakterinin adresini atamak için fixed anahtar sözcüğünü kullanmalıyız. Aşağıdaki programda bir yazının char türden göstericiye nasıl atandığını ve bu gösterici ile yazıdaki her karaktere ne şekilde erişildiğini görüyorsunuz.

```
using System;

class KarakterDizisi
{
    unsafe static void Main()
    {
        fixed(char* ptr = "Sefer Algan")
        {
            for(int i=0; ptr[i] != '\0'; ++i)
                Console.Write(ptr[i]);
        }
    }
}
```

Buradaki en önemli nokta ptr[i]' nin '\0' karakteri ile karşılaştırıldığı yerdir. Göstericiler ile bellekte char türünün büyüklüğü kadar ilerlerken yazının nerede sonlandığını bilemeyiz. Bunun için

```
char* ptr = "Sefer Algan"
```

deyimi ile belleęe yerlestirilen 'n' karakterinden sonra null deęerini ifade eden '\0' karakter yerleřtirilir. Bu karektere rastlanıldıęı zaman yazının sonuna gelmiř bulunuyoruz.

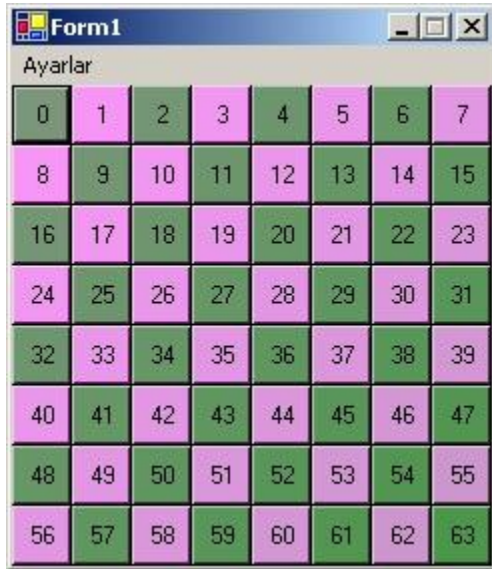
Not: Göstericilerle ilgili yayınlanan bu makale dizisi yazmıř olduęum ve yakında Pusula yayıncılıktan çıkacak olan C# kitabından alınmıřtır.

.NET'te Dinamik Kontrol Oluřturma

Bu yazıda windows ve web uygulamalarında dinamik kontrollerin nasıl oluşturulacağını ve oluşturulan dinamik kontrollere erişim yöntemlerini inceleyeceğiz. Ayrıca bu yazıda dinamik kontrol oluşturma ile ilgili kapsamlı bir örnek kod yer almaktadır.

Bir çok uygulamada(özellikle oyunlarda) kontrollerimizin sayısını tasarım aşamasında bilemeyebiliriz. Örneğin hepimizin bildiği mayın tarlası oyununda, oyun alanı bir çok kareden oluşmaktadır. Üstelik oyun alanı, kullanıcının isteğine göre değişebilmektedir. Eğer tasarım aşamasında bütün kontroller form üzerine yerleştirilmiş olsaydı hem bu kontrollerin yönetimi zorlaşacaktı hemde gereksiz yere bir çok kod yazılmış olacaktı. Buna benzer bir durum web formları ile programlama yaparken de görülebilir. Örneğin bir anket sisteminde anketin seçenekleri kadar CheckBox kontrolünü dinamik olarak web formunun üzerine yerleştirebilmemiz gerekir. Eğer bu imkanımız olmasaydı Web formları çok esnek bir programlama modeli sunmazdı. Neyse ki Microsoft tasarımcıları herşeyi düşünmüş... :)

Dinamik kontrol oluşturmaya geçmeden önce nasıl bir form tasarımı yapmaya çalıştığımıza bakalım. Aşağıdaki formdan da gördüğünüz üzere 8X8'lik bir dama tahtasının her bir karesi aslında bir Button kontrolünden ibarettir. Tasarım aşamasında 64 tane Button kontrolünü form üzerine yerleştirip herbirini tek tek düzenlemek yerine bir döngü vasıtası ile istediğimiz sayıda Button kontrolünü oluşturup Form elemanına ekleyeceğiz.



ToolBox penceresinde gördüğünüz bütün kontrolleri temsil eden sınıflar System.Windows.Forms isim alanında bulunan **Control** isimli sınıftan türetilmiştir. Bazı kontroller ise yapısında değişik kontrolleri barındıracak şekilde tasarlanmıştır. Örneğin Form penceresi üzerine eklenen kontroller bu duruma bir örnektir. Kontrolleri gruplamak için kullanılan GroupBox ta bu şekildedir. Bu özel kontrollere yeni bir kontrol eklemek için bu Control türünden olan **Controls** isimli koleksiyon kullanılmaktadır. Örneğin bir form üzerine yeni bir Button kontrolü eklemek için aşağıdaki deyimleri yazmalıyız.

```
Button tb = new Button();  
Form1.Controls.Add(button)
```

Yukarıdaki deyimleri bir döngü içinde yapıp oluşturduğumuz Button kontrollerinin ilgili özelliklerini değiştirdikten sonra dama tahtasını rahatlıkla oluşturabiliriz.

Şimdi bu yazının en altında bulunan linki kullanarak projeyi bilgisayarınıza yükleyerek projeyi açın. Form1 sınıfının içinde bulunan aşağıdaki TahatCiz() isimli metodu inceleyerek kontrollerin dinamik olarak nasıl yaratıldıklarını inceleyin.

```

private void TahtaCiz(int Boyut1,int Boyut2, int En, int Boy)
{
    int satir =0;
    int sutun =0;

    Button tb = new Button();

    for(int i=0; i < Boyut1*Boyut2; i++)
    {
        if(i % Boyut2 == 0)
        {
            satir++;
            sutun = 0;
        }

        tb = new Button();
        tb.Name = "tb" + i.ToString();
        tb.TabIndex = i;
        tb.Text = i.ToString();
        tb.Size = new System.Drawing.Size(En,Boy);

        Point p = new System.Drawing.Point(tb.Size.Width + (sutun-1)*tb.Width,tb.Size.Height + (satir-2)*tb.Height);

        tb.Location = p;
        tb.FlatStyle = FlatStyle.Standard;
        tb.Click += new System.EventHandler(this.button1_Click);

        Butonlar.Add(tb);
        this.Controls.Add(tb);
        sutun++;
    }

    this.ClientSize = new Size(tb.Width*Boyut2 ,tb.Height*Boyut1);
    satir = 0;
    sutun = 0;
    Butonlar.TrimToSize();
    RenkAyarla();
}

```

TahtaCiz() isimli metodu biraz inceleyelim. Bu metod kendisine gönderilen bilgiler ışığında form üzerine dinamik bir dama tahtası çizmektedir. Ayrıca işe yarar bir kaç işlem daha yapmaktadır. Bu metoda gönderilen Boyut1 ve Boyut2 değişkenleri dama tahtasının boyutlarını belirtmektedir. En ve Boy parametreleri ise tahtadaki her bir karenin enini ve boyunu belirtmektedir. Bu bilgileri parametre olarak almamız istediğimiz boyutta dama tahtasını çizebileceğimiz anlamına gelmektedir. Zaten form üzerine konulan bir menü yardımıyla dama tahtasının boyutu ve karelerin en ve boyu istenildiği gibi değiştirilmektedir. Bu metod sadece butonların form üzerine yerleştirilmesinden sorumludur. RenkAyarla() isimli diğer bir metod ise yerleştirilen bu butonların bir algoritmaya göre renklerini ayarlamayı sağlamaktadır. Bu metod içinde oluşturulan buton kontrollerine başka metodlar içinde erişebilmek için oluşturulan her kontrol global düzeyde tanımlanan Butonlar isimli bir ArrayList koleksiyonuna eklenmektedir. Bu metottaki diğer önemli bir satır ise oluşturulan dama tahtasının boyutlarına göre form nesnesinin yeniden şekillendirilmesidir. Bu işlem

```
this.ClientSize = new Size(tb.Width*Boyut2 ,tb.Height*Boyut1);
```

satırıyla yapılmaktadır.

Not: Her bir Button kontrolünün yanyana ve alt alta nasıl yerleştirildiğini daha iyi anlamak için size tavsiyem kağıt üzerine bir dama tahtası çizip her bir karenin konumuna ilişkin matematiksel bir ifade bulun. Böylece herşey daha açık olacaktır.

Aşağıdaki satırda her bir buton kontrolünün Text özelliği döngü değişkeni olan i'ye atanmaktadır.

```
tb.Text = i.ToString();
```

Dinamik kontrolleri oluşturma ile ilgili diğer bir önemli nokta da kontrollere ilişkin olayların nasıl çağrılacağıdır. Dikkat ederseniz yukarıdaki metotta oluşturulan bütün kontrollerin Click olayına EventHandler temsilcisi yardımıyla button1_Click() metodu iliştilmiştir. Oluşturulan butonlardan herhangi birine tıkladığınızda bu metot işletilecektir. Peki hani butonun tıkladığını nasıl anlayacağız. Bunun için button1_Click() metodunun bildirimine bakalım.

```
private void button1_Click(object sender, System.EventArgs e)
{
    Button b = (Button)sender;

    MessageBox.Show(b.Text);
}
```

Yukarıdaki metodun bir parametresi olan sender değişkeni bu metodun hangi button'un tıklanması sonucu işletildiğini tutmaktadır. Bize bu imkanı sağlayan elbetteki EventHandler temsilcisidir(delegate).

Dönüşüm operatörü kullanılarak **sender** nesnesi Button nesnesine dönüştürülmektedir. (Bu işleme unboxing denildiğini de hatırlatmak isterim)

Bu durumda, örneğin üzerinde 39 yazan Button'a tıkladığınızda gösterilen mesaj kutusuna gibi 39 yazacaktır.

Aşağıdaki ekran görüntüsündeki dama tahtasının biçimi uygulamadaki Ayarlar menüsündeki Renk sekmesi tıklanarak oluşturulabilir.

Not: Dama tahtası biçiminde bir form oluşması için Renk Seçimi penceresindeki Gradyan Katsayısı ortalımalıdır. Renk ayarlama işleminin nasıl yapıldığı bu yazı kapsamında olmadığı için detaylarını anlatmayacağım. RenkAyarla() isimli metodu incelemenizi tavsiye ediyorum.

Web formları ile dinamik kontrol oluşturmak yukarıda anlattıklarımın farklı değil. Yeni bir ASP.NET uygulaması açarak aşağıdaki gibi bir form tasarlayın. Dinamik olarak oluşturacağımız kontrolleri Page sınıfına ekleyebileceğimiz gibi Placeholder kontrolünü de kullanabiliriz. Ben bu iş için Placeholder kontrolünü tercih ettim.

"Oluştur" isimli butonun Click olayına ilişkin metodunu aşağıdaki gibi değiştirip sonucu inceleyelim.

```
private void Button1_Click(object sender, System.EventArgs e)
{
    int Adet = Convert.ToInt32(TextBox1.Text);

    for(int i=0; i<Adet; ++i)
    {
        TextBox yeni = new TextBox();
        yeni.ID = "Text" + i.ToString();
    }
}
```

```
yeni.Text = "TextBox" + i.ToString();  
  
Placeholder1.Controls.Add(yeni);  
Placeholder1.Controls.Add(new LiteralControl("<br/>"));  
}  
}
```

Projeyi derleyip çalıştırın ve TextBox kutusuna bir tamsayı girip "Oluştur" butonuna tıklayın. İşlemleri başarı ile yaptıysanız aşağıdaki gibi "Placeholder" kontrolü içinde 4 adet TextBox kontrolü dinamik olarak yerleştirilecektir.



Not : Sunucu kontrolü olmayan
 etiketinin LiteralControl olarak tanımlandığına dikkat edin.

Dinamik kontrolleri oluşturmayı öğrendiğinize göre artık web tabanlı bir mayın tarlası oyunu yapmanın zamanı geldi sanırım :) Biraz çaba ile bu oyunu rahatlıkla yapabileceğinizi düşünüyorum.

Visual C#.NET 2003'teki Yenilikler

Bu yazıda Visual Studio.NET 2003 ile birlikte gelen Visual C# dilinde yapılan önemli değişikliklere ve Visual Stdio.NET geliştirme ortamının(IDE) yeni özelliklerine değinilecektir. Bu yeni özellikler ve değişiklikler 3 ana başlık altında incelenecektir : Derleyici ve dildeki değişiklikler, proje geliştirme ile ilgili değişiklikler ve VS.NET 2003 IDE'sindeki değişiklikler.

Giriş

Bildiğiniz gibi 2001 yılından beri C# dili ECMA(European Computer Manufacturer's Association) tarafından standart hale getirilmiştir. Bu herhangi bir kurum veya kuruluşun istediği takdirde ECMA-334 standartlarına uymak koşulu ile kendi C# derleyicisini üretebileceği anlamına gelmektedir. Nitekim ECMA-334 standartları çerçevesi içerisinde şu anda farklı platformlar için geliştirilmiş C# derleyicileri bulunmaktadır. Bu derleyiciler içerisinde en çok bilinen ve kullanılanı Microsoft'un Visual C# derleyicisidir. Microsoft, ECMA standartlarına bağlı kalmak koşulu ile kendi derleyicisini istediği şekilde biçimlendirebilmektedir. Bu yazıda Visual C# derleyicisine ve dilin yapısına 2003 versiyonu ile birlikte gelen yenilikleri inceleyeceğiz. Ayrıca Microsoft'un Visual Studio.NET yazılım geliştirme platformu IDE'sine yeni eklediği bir takım özellikleri inceleyeceğiz.

Visual C# Dili ve Derleyicisindeki Değişiklikler

Visual C# 2003 versiyonunda iki ana değişiklik yapılmıştır. Bu değişiklikler eski kodların çalışmamasına sağlayacak nitelikte değildir. Bu değişikliklerin ilki **#line önışlemci komutudur**. #line önışlemci komutu eski versiyonda zaten bulunmaktaydı, yeni versiyonda #line önışlemci komutuna yeni bir anlam daha yüklenmiştir. Yeni kullanımda, iki #line komutu arasında kalan kod satırı Debugger programı için bilgi sağlamaz. Yani Debugger için bu kodlar önemsizdir. Program debugger ile çalıştığında ilgili kod satırlara dikkate alınmaz. Dikkat etmemiz gereken nokta derleme işleminin normal biçimde yapılmasıdır. Zira #line önışlemci komutuna yüklenen bu anlam sadece Debugger(hata ayıklayıcı) programını ilgilendirmektedir. #line önışlemci komutunun bu anlamda kullanılması için **"hidden"** parametresinin kullanılması gerekmektedir. Aşağıda #line önışlemci komutunun kullanılması ve sonuçlarının görülmesi adım adım anlatılmaktadır.

```
using System;

class MyClass
{
    public static void Main()
    {
        Console.WriteLine("www"); // Buraya "Breakpoint" koyun.

        #line hidden

        Console.WriteLine("csharpnedir");

        #line default

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


Kırmızı yazılı satıra bir breakpoint ekledikten sonra Debugger ile birlikte programı derleyin. Bunun için yapmanız gereken F5 tuşuna basmak ya da Debug menüsünden Start sekmesini seçmektir. Program çalıştığında programdaki her 3 satırın da ekrana yazdırıldığını göreceksiniz, ancak kontrol Debugger programına geçtiğinde Debug menüsündeki StepOver(F10) komutu ile program içinde satır satır ilerlerken Debugger programının ikinci satırıyla ilgilenmediğini göreceksiniz.

Diğer bir değişiklik ise XML yorum satırları ile ilgilidir. Bildiğiniz gibi C# kodu içerisinde XML formatında yorum satırları eklenebilmektedir. Bu yorum satırları metotlar, sınıflar ve diğer tipler hakkında dokümantasyon sağlamak için yazılmaktadır. Derleme işleminde istenirse bu yorum satırları C# derleyicisi tarafından ayıklanarak rapor halinde sunulabilir. Eski versiyonda XML yorum satırları 3 tane ters bölü karakterinden sonra yazılırdı. Örneğin sıklıkla gördüğümüz "summary" elementi aşağıdaki gibi yazılabilir

```
/// <summary>
///
/// </summary>
```

Yeni versiyonda XML yorum satırları /** ve */ karakterleri arasında da yazılabilir hale getirilmiştir. Bu yeni kurala göre aşağıdaki yazım biçimleri ile XML yorum satırları oluşturulabilir.

```
/**
<summary>yorum</summary>
*/

/** <summary>yorum</summary> */

/**
* <summary>
* yorum</summary>*/
```

Yukarıdaki her üç kullanımda eşdeğerdir.

Bunların dışında bir önceki versiyonla yeni versiyondaki kullanımları birleştirip aynı kaynak kod içinde aşağıdaki gibi de kullanabiliriz.

```
/**
<summary>Yorum
        satiri
*/
/// <summary>
```

Yapısal Değişiklikler

Visual C# 2003 derleyicisinin ürettiği bazı kodlarda ve içsel mekanizmalarda da değişiklik olmuştur. Bu değişikliklerden en önemlileri özellik(property) bildiriminde ve foreach döngü yapısının çalışma mantığında görülmektedir.

Sınıfların bir üye elemanı olan özellik bildirimi set ve get anahtar sözcükleri ile yapılır. Bu yüzden bu özelliklere genellikle "setter" ve "getter" da denilmektedir. Özellik bildirimi C# derleyicisi tarafından özel metotlara çevrilir. Bu metotlar sırasıyla get_OzellikIsmi() ve set_OzellikIsmi() metotlarıdır. Dolayısıyla bir nesne üzerinden herhangi bir özelliği

çağırdığımızda aslında bir metot çalıştırmış oluyoruz. Zaten C++ dilinde bu tür işlemleri yapmak için Get ve Set ile başlayan çeşitli metotlar tanımlanırdı. C# bu işlemleri biraz daha soyutlaştırarak kullanıcının daha rahat çalışmasını sağlamıştır.

Visual C# 2002'de önemli bir bug vardı. Bu bug özellik bildirimlerinin get ve set bloklarının varlığından kaynaklanmaktaydı. Örneğin aşağıdaki kodda gördüğünüz sınıf bildirimi herhangi bir hata vermiyordu.

```
class Deneme
{
    public int a
    {
        get
        {
            return 5;
        }
        set
        {
            value = 5;
        }
    }

    public int get_a()
    {
        return 3;
    }

    public int set_a()
    {
        return 3;
    }
}
```

Bu sınıf bildirimini içeren bir kaynak kodu delendiğinde

```
Deneme d = new Deneme();
d.a = 5;
```

satırlarındaki **d.a;** ile a özelliğinin set bloğunun mu çalıştırılacağı yoksa set_a() metodunun mu çalıştırılacağı belli değildir. Bu tür bir bildirimin derleme hatasına yol açması gerekir. Çünkü a özelliğinin bildirimindeki set ve get blokları sayesinde zaten get_a() ve set_a() metot bildirimleri yapılmıştır. Visual C# 2003 derleyicisi ile bu sınıf bildirimini içeren bir kaynak kod derlenemeyecektir.

Not: Arayüzlerde bildirim yapılmış özellikler için derleme zamanında get_Ozellik() ve set_Ozellik() gibi iki metot tanımlı yapılmaz. Bu yüzden aşağıdaki kodda IArayüz arayüzünden türemiş olan sınıf içinde a özelliğinin get ve set blokları tanımlanırken set_a() ve get_a() metotları kullanılamaz.

```
interface Deneme : IArayuz
{
    public int a
    {
        get;
        set;
    }
}
```

```

}

class Deneme : IArayuz
{
    //Geçerli Özellik Bildirimi
    public int a
    {
        get
        {
            return 5;
        }
        set
        {
            value = 5;
        }
    }

    //Geçersiz Bildirim
    public int IArayuz.get_a()
    {
    }

    //Geçersiz Bildirim
    public int IArayuz.set_a(int deger)
    {
    }
}

```

Derleyicinin yapısındaki diğer bir değişiklik foreach döngüsünün işletilmesi sırasında görülmektedir. Bildiğiniz gibi foreach ile iteratif bir şekilde türlere ait dizilerin elemanlarına erişmemiz mümkün, ancak bütün türler için foreach döngüsünü kullanamayız. Bunun için foreach ile kullanılacak türlerin bazı metotları ve özellikleri sağlaması gerekir. Bu metotlar ve özellikler IEnumerator arayüzü içinde bildirilmiştir. foreach döngüsünün bitiminde C# derleyicisi döngüde kullanılan türün IDisposable arayüzünü uygulayıp uygulamadığını kontrol etmiyordu. foreach ile kullanılan tür IEnumerator arayüzünü uygulasın yada uygulamassın Visual C# 2003 derleyicisi IDisposable arayüzünün uygulanmış olup olmadığını kontrol eder ve bu sayede IDisposable arayüzündeki Dispose() metodu çağrılır.

Göze çarpan diğer bir değişiklik ise niteliklerin kullanımında görülmektedir. private elemanı olan nitelikler eski versiyonda kullanılabiliyordu ancak yeni derleyicide sadece public nitelik elemanları kullanılabilmektedir. Örneğin Deneme isimli nitelik sınıfı ve bu sınıfın private üye elemanı olan Ozellik bildirilmiş olsun. Buna göre aşağıdaki metot bildirimi yeni derleyici için geçersizdir.

```

[Deneme(Ozellik = "deneme özellik")]
public int Metot()
{
}

```

Not: Yukarıdaki bildirim Visual C# 2002 derleyicisi için geçerlidir.

Diğer bir değişiklik enum sabitlerinin artık char türüne dönüştürülebilmesidir. Örneğin

aşağıdaki programı derleyip çalıştırdığınızda ekrana 'L' karakteri yazdırılacaktır. (L karakterinin unicode karşılığı 76'dır)

```
using System;

enum Harfler
{
    A,B = 76,C
}

class Class1
{
    static void Main()
    {
        char c = (char)Sefer.B;
        Console.WriteLine(c);
    }
}
```

Yukarıda anlatılan değişikliklerin tamamı uygulama performansını artırmak ve ECMA standartlarına daha sıkı bağlılık amacıyla yapılmıştır. Bir önceki versiyonda ECMA standartlarına uymayan kurallar ve bazı bug'lar yeni versiyonda düzeltilmiştir.

Geliştirme Ortamındaki Değişiklikler

1- Geliştirdiğimiz uygulamaları derlerken çeşitli uyarılar yada hatalar alabiliriz. Bu yazılım geliştirmenin doğal bir sürecidir. Bazı durumlarda derleyicinin verdiği uyarılar gerçekten can sıkıcı olabilir. Bazende bu uyarılar programcının olası bir hataya karşı önlem alması için olabilir. VS.NET 2003 ortamında proje derlenirken bazı uyarıların verilmemesini sağlayabiliriz. Komut satırı derleyicisinde bu işlem **/nowarn** argümanı ile yapılır. Örneğin derleme işlemi sırasında CS0050 ve CS0060 kodlu uyarının verilmemesi için csc derleyicisi aşağıdaki gibi çalıştırılır.

csc dosya.cs /nowarn:50,60

VS.NET kullanarak bu uyarı engelleme işi proje özelliklerinden ayarlanır. Solution Explorer penceresinde projeye sağ tıklayıp proje özellikleri penceresinden "Configuration Properties" sekmesini ardından "Build" seçeneğini seçin. Aşağıdaki ekran görüntüsündeki alana uyarı kodlarını noktalı virgül ile birbirinden ayrılacak şekilde yazın.

Errors and Warnings	
Warning Level	Warning level 4
Treat Warnings As Errors	False
Suppress Specific Warnings	50;60

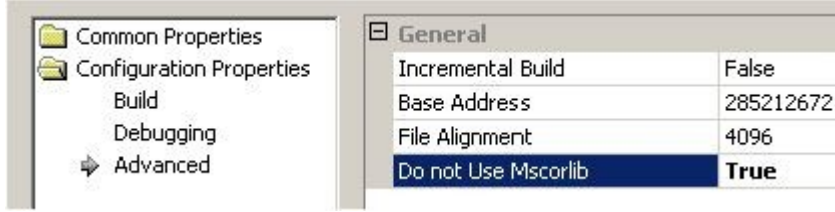
2- Yeni versiyon ile birlikte istersek **/nostdlib** argümanını kullanarak mscorlib.dll kütüphanesini programlarımıza eklemeyebiliriz. Bildiğiniz gibi bütün System isim alanı ve bu isim alanında bildirilmiş türler mscorlib.dll kütüphanesinde bulunmaktadır. Komut satırından

csc dosya.cs /nostdlib-

şeklindeki bir derleme ile standart kütüphane olan mscorlib.dll uygulamamıza eklenmez. Bu işlemi kendi System isim alanımızı bildirmek için kullanabiliriz.

/nostdlib+ şeklindeki kullanım varsayılan kullanım ile eşdeğerdir, yani mscorlib.dll kütüphanesi uygulamaya eklenir.

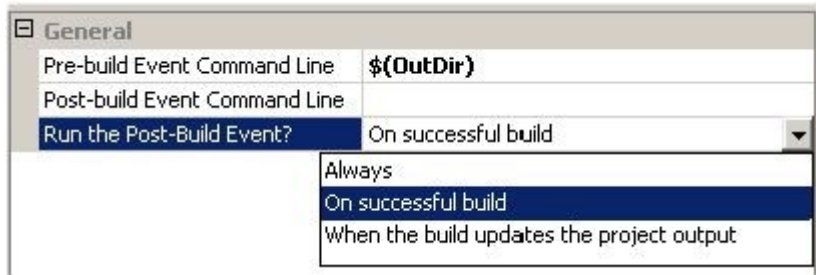
VS.NET ortamında bu değişikliği yapmak için proje özelliklerinden "Configuration Properties" sekmesini ardından "Advanced" seçeneğini seçin. Bu pencereden "Do not use Mscorlib" seçeneğini aşağıdaki gibi true yapın.



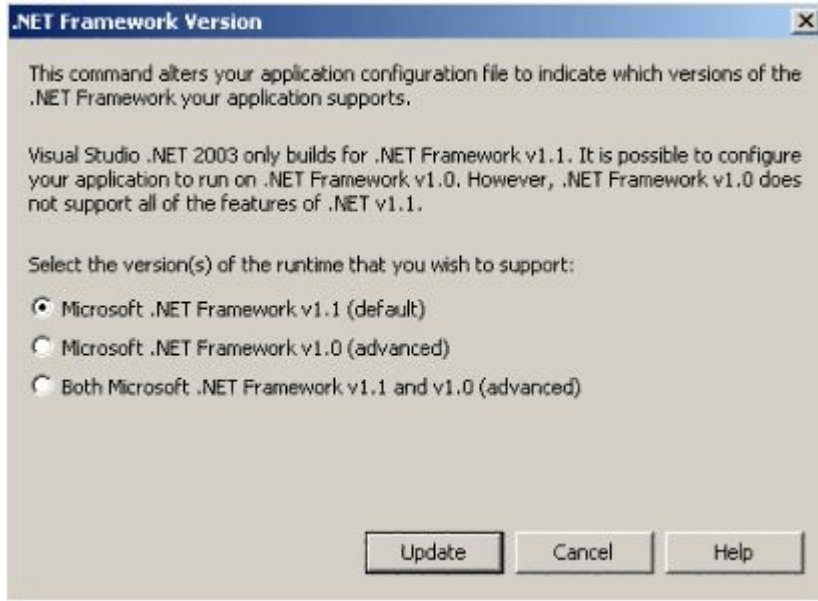
3- Proje özellikleri penceresinde "Common Properties" seçeneğindeki "Build Events" kısmındaki alanları doldurarak projenin oluşturulması sırasında ve proje oluşturulduktan sonra çalıştırılacak batch komutları yazılabilir. Önceden tanımlanmış bir kaç makroyu da kullanmak mümkündür. "Post-Build Event Command Line" seçeneği ile ilgili 3 durum söz konusudur. Bunlar batch komutlarının

- a) Her zaman çalıştırılması
- b) Yalnızca başarılı oluşturmalar sırasında çalıştırılması
- c) Oluşturma işlemi çıktı dosyalarını güncellediği zaman çalıştırılmasıdır.

Bu seçeneklerin ayarlandığı pencere aşağıdaki gibidir.



4- Web ve Windows uygulamaları için birden fazla çalışma zamanı desteği sağlamak mümkündür. Örneğin .NET Framework 1.0 ve .NET Framework 1.1 çalışma zamanını destekleyecek ve bu ortamlarda çalışabilecek uygulama geliştirmek mümkündür. Bu ayarı yapmak için proje özellikleri penceresindeki "Common Properties" sekmesindeki "General" sayfasından "Supported Runtimes" özelliğini değiştirmek gerekir. Bu özelliğe tıklanıldığında aşağıdaki pencere ile karşılaşılır.



Bu ayar yapılırken dikkat edilmesi gereken nokta .NET Framework 1.1 'deki bazı özelliklerin 1.0 versiyonunda bulunmamasıdır. Bu yüzden 1.1 versiyonunda geliştirilen projelerin 1.0 versiyonunda çalışabilmesi için ortak özelliklerin bulunması gerekir.

Yeni Eklenen "Intellisense" Özellikleri

VS.NET IDE'si geliştirici için büyük kolaylıklar sağlamaktadır. Bu kolaylıkların en bilineni ve en işe yarayanı IDE'nin akıllı olmasıdır. IDE, dilin kurallarına göre legal olan bir çok olayı bizim için otomatik yapmaktadır. Örneğin bir nesnenin metotlarını ve özelliklerini '.' operatöründen sonra görebilmemiz gibi. VS.NET 2003 IDE sine hoşunuza gidecek yeni akıllı özellikler eklenmiştir. Bu kısımda bu yeni özellikleri göreceğiz.

1- Göze çarpan ilk değişiklik olaylara yeni bir metot ekleme sırasında görülmektedir. VS.NET tasarım ekranında bir kontrole ait olayı işlemek istediğimizde Properties ekranında ilgili olayı seçip metot ismini yazıyorduk. Yeni versiyonda bu işlemler kod editöründen de yapılabilir. Bir olaya += operatörü ile bir metot eklemek istediğimizde "TAB" tuşuna basarsak olayı yönetecek temsilci new operatörü ile eklenir. Tekrar "TAB" tuşuna basıldığında olayı yönetecek temsilcinin temsil edeceği metodun prototipine uygun bir "event handler" metodunun bildirimi otomatik olarak yapılır.

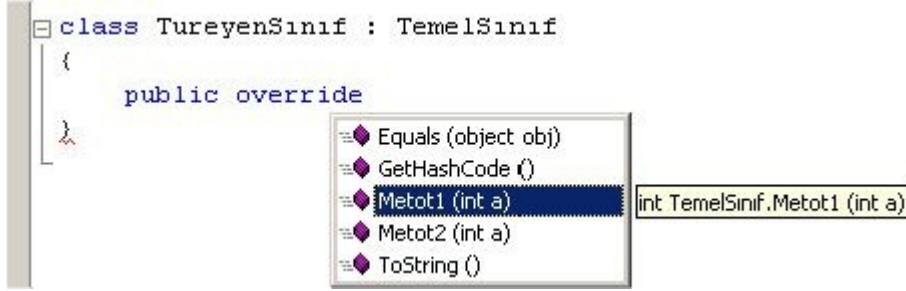
2- Diğer bir yeni özellik arayüzlerin türetilmesi sırasında görülür. Bildiğiniz gibi bir sınıf bir arayüzden türetiliyorsa arayüzde bildirilmiş olan bütün özellik ve metotların türeyen sınıfta tanımlanması yani uygulanması gerekir. VS.NET IDE si türetilen arayüzdeki eleman bildirimlerini türeyen sınıfta otomatik olarak gerçekleştirir. Aşağıdaki ekran görüntüsünde bu işlemin nasıl yapıldığı gösterilmektedir.

```
class Deneme : IDisposable|
Press TAB to implement stubs for interface 'System.IDisposable'
```

Türetilen arayüz ismi yazıldıktan sonra TAB tuşuna basılırsa arayüzdeki eleman bildirimleri Deneme sınıfına otomatik olarak eklenecektir. Bu işlemten sonra Dispose() metodunun bildirimi Deneme sınıfına otomatik olarak eklenmiş olacaktır.

3- Türetme sırasında yeniden yüklenebilecek(override) metotlar override anahtar

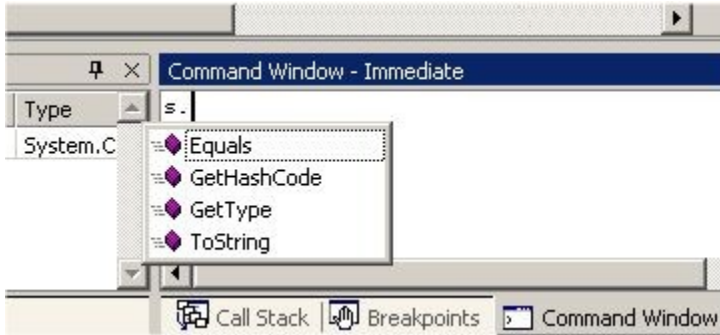
sözcüğü yazıldıktan sonra otomatik olarak gösterilir.Örneğin Metot1 ve Metot2 adında 2 tane sanal(virtual) metodu olan TemelSınıf'tan türeyen sınıf içinde override anahtar sözcüğü kullanıldıktan sonra aşağıdaki ekran görüntüsü elde edilir.



Yukarıdaki ekranda Object sınıfının metotlarının da gösterildiğine dikkat edin. Ayrıca TemelSınıf taki sanal olmayan metotların da gösterilmediğine dikkat edin.

4- VS.NET IDE'sinde nesneler ile '.' operatörü kullanıldığı anda nesnenin türüne ait elemanlar listelenir. Listeleme yapılırken ilk eleman her zaman en başta olur. Yani sıralama işlemi eleman isimlerinin alfabetik sıraya göre dizilmesiyle yapılır. Yeni IDE ile birlikte kullandığınız elemanlar "sık kullanılan elemanlar" bölümüne eklenerek bir sonraki kullanımda en son kullanmış olduğunuz elemanın seçili olması sağlanır. En çok kullandığımız Console sınıfının WriteLine() metodunu örnek verelim. Listede Write() metodu WriteLine() metodundan önce gelmektedir. Dolayısıyla WriteLine() metodunu seçebilmek için "Writel" yazmak gerekecektir. Oysa yeni kullanımda WriteLine() metodunu bir kere seçtikten sonra bir sonraki kullanımda '.' operatörüne basıp "W" yazıldığı anda WriteLine() metodu seçilecektir.

5- Diğer bir yeni özellik ise Debug işlemi sırasında kullanılan "Immediate Window" penceresinin kullanımında görülür. Artık "Immediate" penceresinde de kod editöründe olduğu gibi nesnelerin özelliklerini ve metotlarını görebilmekteyiz. Object türünden olan s nesnesinin üye metotlarının "Immediate" penceresinden ne şekilde görüldüğü aşağıdaki ekran görüntüsünde gösterilmiştir.



Not: "Immediate" penceresi ile çalışabilmek için kaynak kodda herhangi bir satıra "Breakpoint" yerleştirip programı "Debugger" ile birlikte derlemeniz gerekir.

Sonuç

Visual C# ve VS.NET IDE'sinde yapılan değişiklikler yukarıda anlatılanlar ile sınırlı değildir. Ancak göze çarpan yeni özellikler bunlardır diyebiliriz. VS.NET 2003'teki diğer göze çarpan özellik ise "MMIT" ve "Smart Device" eklentilerinin varsayılan olarak yüklenmesidir.

C# Dilinde Özellikler – 1

Nesne yönelimli programlamanın günümüzde ne kadar yaygın olduğunu programlama ile ilgilenen herkes bilmektedir. Nesne Yönelimli Programlama (NYP) yaklaşımında temel olan prensiplerden birisi bilgi gizleme (information hiding)'dir. Bu prensibi projelerimizde uygulamak için C#'in sunduğu en önemli araçlardan biri olan sınıf özellikleri (class properties) konusunu inceleyeceğiz.

Bildiğiniz gibi, C# dilinde tasarlanmış bir sınıfta iki temel unsur bulunur. Birincisi sınıfın özellikleri (fields), ikincisi ise sınıfın metodları (methods)'dır. Herhangi bir sınıfın özelliği sınıfta tutulan ilişkili verilerlerdir. Diğer taraftan sınıfın bizim için değişik işleri yapmasını metodları vasıtasıyla sağlarız. Sınıf tasarımı çok önemli bir iş olup; deneyim, konsantrasyon ve dikkat ister. Sınıfımızın özelliklerini tutan veriler, program akışı sırasında sınıf dışında değiştirilebilir veya bu değerlere ulaşmak istenebilir.

Bu durumda akla ilk gelen çözüm sınıfın verilerinin hepsinin dışarıdan ulaşılabilmesini ve değiştirilebilmesine olanak sağlayan **public** anahtarı ile tanımlamaktır. Aşağıdaki programda bu tür bir çözümün uygun olabileceği düşünülmüştür:

```
using System;

namespace Property_Makale
{
    class Otomobil
    {
        public int model;
        public string marka;
        public string renk;

        public Otomobil(int model, string marka, string renk)
        {
            if(model>DateTime.Now.Year)
                this.model=DateTime.Now.Year;
            else this.model=model;

            this.marka=marka;
            this.renk=renk;
        }

        public void OzellikleriGoster()
        {
            Console.WriteLine("\nOtomobilimizin Özellikleri: ");
            Console.WriteLine("\t Marka: "+ marka);
            Console.WriteLine("\t Model: "+ model);
            Console.WriteLine("\t Renk: "+ renk+"\n" );
        }
    }

    class OtomobilTest
    {
        static void Main(string[] args)
        {
            Otomobil oto1 = new Otomobil(2000, "BMW" , "Siyah");
        }
    }
}
```



```

        oto1.OzellikleriGoster();

        oto1.model=300;

        oto1.OzellikleriGoster();
    }
}

```

Yukarıdaki kod örneğimizde iki tane sınıf bulunmaktadır. **Otomobil** sınıfı ile otomobil nesnelerimizi oluşturabiliriz. Ayrıca bu sınıfın **OzellikleriGoster()** metodu ile herhangi bir otomobil nesnemizin özelliklerini görmek için ekrana yazdırıyoruz. İkinci sınıfımızda (**OtomobilTest**) ise Otomobil sınıfımızdan nesneler oluşturmak ve onların özelliklerini çağırmak için kullanacağız. Şimdi isterseniz Main() fonksiyonunu incelemeye başlayalım. Metodun hemen ilk başında **oto1** isimli nesnemizi oluşturuyoruz. **oto1** nesnemizin özellikleri 2000 model, siyah ve BMW olsun. Bir sonraki satırda **oto1** nesnemizin modelini 300 yapıyoruz. İşte burda büyük bir hata yapılıyor! Çünkü 300 yılında henüz otomobil üretilmemiştir. Böyle bir hatayı nasıl önleriz? Çözüm olarak otomobil nesnemizin herhangi bir özelliğini değiştirmek için ayrı bir metod yazmamız gerekir. O zaman programızı şu şekilde değiştirmemiz gerekiyor:

```

using System;

namespace Property_Makale
{
    class Otomobil
    {
        private int model;
        public string marka;
        public string renk;

        public Otomobil(int model, string marka, string renk)
        {
            if(model>DateTime.Now.Year)
                this.model=DateTime.Now.Year;
            else this.model=model;

            this.marka=marka;
            this.renk=renk;
        }

        public void OzellikleriGoster()
        {
            Console.WriteLine("\nOtomobilimizin Özellikleri: ");
            Console.WriteLine("\t Marka: "+ marka);
            Console.WriteLine("\t Model: "+ model);
            Console.WriteLine("\t Renk: "+ renk+"\n" );
        }

        public void ModelDegistir(int yeniModel)
        {
            if( (yeniModel>DateTime.Now.Year)|| (yeniModel<1900) )

```

```

        Console.WriteLine("Otomobilin modeli su an ki yıldan büyük veya 1900'den
küçük olamaz ! ");
        else this.model=yeniModel;
    }
}

class OtomobilTest
{
    static void Main(string[] args)
    {
        Otomobil oto1 = new Otomobil(2000, "BMW" , "Siyah");
        oto1.OzellikleriGoster();

        oto1.ModelDegistir(300);
        oto1.OzellikleriGoster();
    }
}

```

Yukarıdaki programda Otomobil sınıfına **ModelDegistir(int yeniModel)** metodunu ekledik. Bu metod ile modeli değiştirilecek nesnenin modelinin şu anda bulunulan yıldan sonra ve 1900'den önce yapılmak istendiğinde hata mesajı veriyor ve modelini değiştirmiyor. Ayrıca sınıf içindeki model değişkenin tanımlanmasında **private** anahtarını da kullandığımıza dikkat ediniz. Bu şekildeki bir yaklaşım ile hem sınıfın iç işleyişini sınıf dışından saklamış oluyoruz hem de sınıfa ait verilerin değiştirilmesini sırasındaki hataları en az seviyede tutmayı sağlıyoruz.

Fakat yukarıdaki yöntemi genelde C++ programcıları kullanır(dı). Bizler C# programcıları olarak daha gelişmiş bir yola sahibiz. Sınıf içindeki değerleri değiştirmek ve ulaşmak için özellik (Property) aracını kullanırız. Aşağıdaki program ise C#'ın özellikleri nasıl kullandığına bir örnektir:

```

using System;

namespace Property_Makale
{
    class Otomobil
    {
        private int model;
        public string marka;
        public string renk;

        public Otomobil(int model, string marka, string renk)
        {
            if(model>DateTime.Now.Year)
                this.model=DateTime.Now.Year;
            else this.model=model;

            this.marka=marka;

            this.renk=renk;
        }
    }
}

```

```

    }

    public void OzellikleriGoster()
    {
        Console.WriteLine("\nOtomobilimizin Özellikleri: ");
        Console.WriteLine("\t Marka: "+ marka);
        Console.WriteLine("\t Model: "+ model);
        Console.WriteLine("\t Renk: "+ renk+"\n" );
    }

    public int Model
    {
        get
        {
            return model ;
        }

        set
        {
            if((value>DateTime.Now.Year)|| (value<1900) )
            {
                Console.WriteLine("Otomobilin modeli su an ki yildan büyük veya
1900'den küçük olamaz ! \n");
            }
            else this.model=value;
        }
    }
}

class OtomobilTest
{
    static void Main(string[] args)
    {
        Otomobil oto = new Otomobil(2000, "BMW" , "Siyah");
        Console.WriteLine("Otomobilimizin modeli: "+ oto.Model);

        oto.Model=300;
        oto.OzellikleriGoster();
    }
}

```

Yukarıdaki programı çalıştırdığımızda aşağıdaki sonucu alırız:



```

C:\Documents and Settings\root\My Documents\Visual Studio Projects\Property_Makale\bin\Debug...
Otomobilimizin modeli: 2000
Otomobilin modeli su an ki yildan büyük veya 1900'den küçük olamaz !

Otomobilimizin Özellikleri:
    Marka: BMW
    Model: 2000
    Renk: Siyah

```

Çıktısını gördüğünüz program kodunda C#'ın özellik tanımlama ve kullanma yöntemini kullandık. Özellikler ile herhangi bir nesneye ait değişkenin değerini öğrenebilir ve değiştirebiliriz. Yukarıdaki örnek kodda yeralan aşağıdaki kısımda bir özellik(Property) tanımlıyoruz. Genellikle bir özelliğin ismi üzerinde iş yaptığı değişkenin ismi ile aynı olup sadece ilk harfi büyük olur. Aslında bu bir zorunluluk değil. Sadece C#'da bu bir gelenektir. Bu şekilde bir kullanım bizim kodumuzu okuyanların kodu daha kolay anlaması ve bizim başkalarının kodlarını daha kolay anlamamıza yardımcı olur. Bir özelliğin tanımında özellik isminden önce ne tür bir değer dönderecekse onun tipini belirtmeliyiz. Bu genelde özelliğin ilgili olduğu değişkenin tipidir :-)

```
public int Model
{
    get
    {
        return model ;
    }

    set
    {
        if((value>DateTime.Now.Year)|| (value<1900) )
        {
            Console.WriteLine("Otomobilin modeli su an ki yıldan büyük veya
1900'den küçük olamaz ! \n");
        }
        else this.model=value;
    }
}
```

Özellikler içinde **get** ve **set** olmak üzere iki ayrı blok kod olur. İstersek sadece get veya sadece set blokları olan özellikler de yazabiliriz. get bloğunda ilgili değişkenimizin değerini dışarıya döndeririz. set bloğunda ise değişkenimizin değerini değiştiririz. Burda gereken kontrolleri yapıp daha sonra uygunsa girilen değeri kabul edebiliriz. Eminimki get bloğu içinde dikkattinizi değişkeni ismi yerine **value** şeklinde çağırmamız çekmiştir. Bu sayede kod içinde karışıklık olmaz. Zaten sadece bir tane değişken üzerinde çalışıyorsunuz.

Bu makalemizde C# dilinde yeralan özellik (Property) kavramını inceledik. Bir ilerleyen makalelerimizde sadece yazılabilir (read-only) ve sadece okunabilir (write-only) özellik yazmayı ve kullanmayı inceleyeceğiz.

WinAPI Fonksiyonlarının C#'ta Kullanımı

C# ve dotNET ile birlikte yazılım geliştirmeye yeni bir soluk gelmiş olsada C# ile eskiden yazılmış COM komponentlerine erişebilmek mümkündür. Daha önceki iki makalede .NET ve COM ilişkisini detaylı bir şekilde incelemiştik. Bu makalede .NET'in Win 32 API ile nasıl entegre edildiği anlatılacaktır. .NET ve C#'ın yeni imkanlarının yanısıra eski bir teknoloji olan COM ve yönetilmeyen(unmanaged) kodlarla uyumlu bir şekilde çalışması belkide C# ve .NET'i diğer yazılım geliştirme platformlarından ayıran en önemli özelliktir.

Bildiğiniz gibi C#'ta gösterici kullanımı tamamen serbesttir. Bu yüzden eskiden(.NET öncesi) yazılmış ve parametre olarak gösterici alan COM komponentleri ve Windows API fonksiyonları C# ile sorunsuz bir şekilde çalıştırılabilmektedir. Bu yazıda Win API fonksiyonlarının .NET ortamında ne şekilde ele alındığı incelenecektir.

Win32 sistem fonksiyonları kullanıldığında, kod CLR tarafından yönetilmekten çıkar. .NET ortamında geliştirilen bir uygulamada yönetilmeyen kod segmenti ile karşılaşılırsa ilgi kod segmenti CLR tarafından yönetilmekten çıkar. Dolayısıyla "garbage collection" mekanizması ve .NET'e özgü diğer servisler kullanım dışı olur.

CLR tarafından yönetilmeyen kodlara erişebilmek için C#'ta **System.Runtime.InteropServices** isim alanında bulunan ve DllImportAttribute sınıfını temsil eden **DllImport** niteliği kullanılmaktadır. DllImport niteliği ile harici bir kaynaktan bulunan metoda referans vermek için **external** anahtar sözcüğü kullanılır. Bir sınıf bildiriminin en başında external anahtar sözcüğü ve DllImport niteliği kullanılarak CLR tarafından yönetilmeyen bir metod bildirimi yapılır. Tabi metodun gövdesi harici bir kaynaktan zaten var olduğu için bizim metodun gövdesini yazmamızın bir anlamı yoktur. Ardından bu metod sınıfın istenildiği yerde kullanılabilir. İsterseniz basit bir örnekle DllImport niteliğinin kullanımını gösterelim.

Win API windows sistemlerinin programlanabilir arayüzünü içermektedir. Windows uygulamalarının tamamı bu arayüzdeki fonksiyonları ve diğer yapıları kullanmaktadır. Aşağıdaki programda Win32 sistemlerinde bulunan MessageBox() fonksiyonunun kullanımına bir örnek verilmiştir.

Not : Bu yazıda C#'ta niteliklerin(Attributes) nasıl kullanıldığını bildiğiniz varsayılmıştır.

```
using System;
using System.Runtime.InteropServices;

class Class1
{
    [DllImport("user32.dll")]
    public static extern int MessageBox(int tip, string mesaj, string baslik, int secenek);

    static void Main()
    {
        MessageBox(0, "Mesaj", "Win API MessageBox", 2);
    }
}
```

DllImportAttribute sınıfının bir tane yapıcı metodu bulunmaktadır. Bu metod parametre olarak harici kaynağın adı belirtmektedir. Yukarıdaki kaynak kodda MessageBox

fonksiyonunun bulunduğu "user32.dll" isimli dosya DllImport niteliğine parametre olarak verilmiştir. Bu örnekte dikkat edilmesi gereken diğer nokta ise **extern** anahtar sözcüğünün kullanımıdır. Bu anahtar sözcük ile bildirimi yapılan metodun harici bir dosyada olduğu belirtilmektedir. Dolayısıyla C# derleyicisi metodun gövdesini kaynak kodda aramayacaktır.

Programın çalışma şeklini açıklamadan önce ekran çıktısına bakalım :



Yukarıdaki çıktıdan ve kaynak koddan da görüldüğü üzere Win API deki bir fonksiyonun çağırımı klasik metod çağırımından farklı değildir. Değişen tek şey metodun bildirim şeklidir.

Not : Gösterilen mesaj kutusunun farklı formlarını görmek için MessageBox metodunun parametreleri ile oynayın.

Şimdi kısaca yukarıdaki programın çalışma zamanındaki durumunu inceleyelim. Program çalıştırıldığında, CLR tarafından yönetilmeyen bir metod çağırımı yapıldığında ilgili kaynaktan metod belleğe yüklenir ve belleğe yüklenen metodun başlangıç adresi saklanır. Ardından bizim parametre olarak geçtiğimiz değişkenler DLL' deki fonksiyona uyumlu hale getirilir ve parametre olarak geçirilir. Eğer bir geri dönüş değeri bekleniyorsa yönetilmeyen kod bölümünden gelen değer uygun .NET türüne dönüştürülerek işlemlere devam edilir. Bu işlemler tamamen .NET'in alt yapısını ilgilendirmektedir. Dolayısıyla programcının yapacağı iş sadece metodun bildirimini doğru bir şekilde gerçekleştirmektir.

DllImportAttribute sınıfının bir kaç önemli özelliği daha vardır. Bunlardan en önemlisi harici kaynaktaki bir fonksiyona takma isim verebilmemizi sağlayan string türünde olan **EntryPoint** özelliğidir. EntryPoint özelliğini kullanarak MessageBox fonksiyonuna takma isim verebiliriz. Fonksiyon çağırımı bu takma isim ile gerçekleştirilebilmektedir. Örneğin MessageBox fonksiyonuna TebrikMesajiVer şeklinde bir takma isim vermek için aşağıdaki gibi bir bildirim yapılmalıdır.

```
using System;
using System.Runtime.InteropServices;

class Class1
{
    [DllImport("user32.dll", EntryPoint = "MessageBox")]
    public static extern int TebrikMesajiVer(int tip, string mesaj, string baslik, int secenek);

    static void Main()
    {
        TebrikMesajiVer(0, "Tebrikler", "Takma İsim Verme", 0);
    }
}
```

Şimdi de DllImport niteliği ile ilgili diğer özelliklere ve önemli noktalara bakalım.

1 - DllImport niteliği yalnızca metotlara uygulanabilir.

2 - DllImport niteliği ile işaretlenmiş metotlar mutlaka **extern** anahtar sözcüğü ile bildirilmelidir.

3 - DllImport niteliğinin EntryPoint'in haricinde 4 tane isimli parametresi(named parameter) daha vardır. Bunlar : **CallingConvention, CharSet, ExactSpelling, PreserveSig ve SetLastError** parametreleridir. Bu parametrelerden önemli olanlar aşağıda açıklanmıştır.

4 - CallingConvention : Bu parametre CallingConvention numaralandırması türündendir. Bu parametre ile harici metodun ne şekilde çağrılacağı ile ilgili bilgi verilir. CallingConvention numaralandırması 5 tane sembol içerir. Varsayılan olarak bu sembol Winapi olacak şekilde ayarlanmıştır. Yani CallingConvention parametresini DllImport ile kullanmıyorsa varsayılan olarak bu Winapi'dir. Diğer semboller ise **Cdecl, FastCall, ThisCall, StdCall** şeklindedir. En çok Winapi sembolu kullanıldığı için diğer sembollerin ne anlama geldiği anlatılmayacaktır. Diğer sembollerin ne anlama geldiklerini MSDN kütüphanesinden detaylı bir şekilde öğrenebilirsiniz.

5 - CharSet : Harici fonksiyonun çağrımında kullanılacak karakter setini belirler. Bu parametre CharSet numaralandırması ile belirtilir. Varsayılan olarak **CharSet.Auto** şeklindedir. CharSet numaralandırmasının diğer sembolleri **Ansi, Unicode ve None** şeklindedir.

6 - ExactSpelling : EntryPoint ile belirtilen ismin ilgili fonksiyon ismine yazım biçimi bakımından tam uyumlu olup olmayacağını belirtir. Bu özellik bool türündendir ve varsayılan olarak **false** değeridir.

DllImport metodunun varsayılan çağrım biçimi olan Winapi sadece Windows sistemlerine özgün olduğu için sistemler arası taşınabilirliğin yüksek olması gereken projelerde bu niteliğin kullanımından kaçınmak gerekir. Bu yüzden DllImport özelliğini kullanmadan önce ilgili fonksiyonun .NET Framework içinde olup olmadığını kontrol etmek gerekir. Örneğin MessageBox fonksiyonu zaten System.Windows.Forms isim alanında bulunduğu için API kullanarak bu fonksiyondan yararlanmak mantıklı değildir. Zira ileride programınızın Linux ortamında yada diğer farklı ortamlarda da çalışmasını istiyorsanız programınızı değiştirip yeniden derlemeniz gerekecektir. Oysa .NET Framework içinde bulunan standart sınıfları ve onların metotlarını kullanırsanız böyle bir derdiniz olmayacaktır.

C#'da Sıra (Queue) Sınıfı ve kullanımı

Bir önceki yazımızda genel olarak yığın (Stack) veri yapısının çalışma modeline ve C#'ta kullanabileceğimiz yığın sınıfını ve bu sınıfın metodları üzerinde durmuştuk. Şimdi burada ise, diğer önemli veri yapısı olan sıra (queue) veri yapısını inceleyeceğiz. Queue veri yapısının mantığını anladıktan sonra .NET sınıf kitaplıklarında bulunan Queue sınıfını öğreneceğiz.

Queue veri tipi ve .NET'in sınıf kütüphanesinde bulunan Queue sınıfı, yığın (stack) sınıfına çok benziyor. Bu veri tipleri çalışma mantığı olarak tam tersi gibi görünmelerine rağmen bir çok metodları ve özellikleri birebir örtüşüyor. Bu durumda birazdan okayacağınız makalenin formatı yığın makalemizdekine çok benzediğini göreceksiniz.

1. Queue(sıra) veri Yapısının Çalışma Şekli

Sıralar (queue) **İlk Giren İlk Çıkar** (FIFO) prensibi ile çalışan veri yapısı şeklinde bilinirler. Bir sıraya ilk giren eleman ilk olarak çıkar. Sıralara örnek olarak bir markette alışverişini yapan müşterilerin, aldıkları ürünlerin ücretlerini ödemek için kasada sıraya geçmeleri verilebilir. Marketteki sırada sıraya ilk giren müşterinin işi ilk önce biter. Daha sonra ikinci ve üçüncü müşterilerin işleri yapılır.

Sıralar bilgisayar programlamada sık sık başvurulanan veri yapılarıdır. Mesela, işletim sisteminde yapılması gereken işleri bir sıra veri yapısı ile tutarız. Herhangi bir anda yeni bir iş geldiği zaman bu iş sıraya (Queue) girer. Sırası gelen iş yapılır ve sonraki işe geçilir gibi. Queue veri yapıları ayrıca simülasyonlarda da sık sık kullanılır.

2. .NET Sınıf Kütüphanesi Sıra Sınıfı (Queue)

.NET sınıf kütüphanesinde sıra veri yapısını kullanmak için **Queue** sınıfını kullanırız. NET'in yığın (**Stack**) sınıfını kullanmak için program kodunun baş tarafına **using System.Collections;** eklememiz gerekir. Yani sıra sınıfı **System.Collections** isim alanında bulunuyor.

C# veya herhangi bir dilde yazılan yığın veri yapılarında **Enqueue(), Dequeue, Peek(), Clear()** fonksiyonları ve **Count**, özelliği vardır. Bunların yanında **Clone(), CopyTo(), ToArray(), Contains() ve Equals()** metodları .NET'in yığın sınıfında yer alır.

Sıra sınıfının **Enqueue()** metodu sıraya yeni bir eleman ekler. **Dequeue()** metodu ile yığının en öndeki elemanı sıradan siler ve silinen elemanı geriye dönderir. Eğer sıranın tepesindeki elemanı öğrenmek istersek **Peek()** metodu ile öğrenebiliriz. Bu metod sıranın başındaki nesneyi döndürür ama bu nesneyi sıradan silmez.

```
using System;
using System.Collections; // Queue sınıfı bu isim alanı içinde bulunur.

class Sira_Ornek1
```



```

{

public static void Main()
{

    // Queue sınıfından bir nesne oluşturalım:
    Queue sıra = new Queue();

    // Nesnemize Enqueue metodu ile değerler girelim:
    sıra.Enqueue("Ahmet");
    sıra.Enqueue("Ferit");
    sıra.Enqueue("Hasan");
    sıra.Enqueue("Hüseyin");

    // sıra isimli nesnemizin eleman sayısı:
    Console.WriteLine( "\n sıra nesnemizin eleman sayısı: " + sıra.Count);

    // sıra isimli nesnemizin elemanları:
    Console.WriteLine( "\n sıra nesnemizin elemanları: " );
    DegerleriYaz( sıra );

    //sıra isimli nesnemizden bir eleman alalım:
    string eleman= (string)sıra.Dequeue();
    Console.WriteLine(" \n Sıramızın başından şunu aldık: " + eleman);

    //şimdi ise sıranın en başındaki nesneyi öğrenelim.
    // Ama onu sıradan çıkartmayacağız:
    eleman= (string)sıra.Peek();
    Console.WriteLine(" \n Sıramızın başındaki eleman " + eleman);

}

public static void DegerleriYaz( IEnumerable koleksiyon )
{
    System.Collections.IEnumerator Enum = koleksiyon.GetEnumerator();

    while ( Enum.MoveNext() )
        Console.Write( "\t{0}", Enum.Current );

    Console.WriteLine();
}
}

```

Yukarıdaki örnek programda önce Queue sınıfından **sıra** isimli bir nesne oluşturuyoruz. Sonraki altı satırda sıramıza "Ahmet", "Ferit", "Hasan", ve "Hüseyin" değerlerini **Enqueue** metodu ile ekliyoruz. **Degerleri()** ismini verdiğimiz static fonksiyonumuz ile sıramızdaki eleman sayısını ve elemanları ekrana yazdırıyoruz. Daha sonra sıramızdan bir tane elemanı **Deuque()** metodu yardımıyla alıyor ekrana yazdırıyoruz. Programın son kısmında ise **Peek()** metodunu kullanarak sıranın en üstündeki elemanın ne olduğunu öğreniyoruz ve bu eleman sırada kalıyor.

Sıra sınıflarında bulunan diğer iki temel fonksiyonlar olan **Count** özelliği ve **Clear()** metodlarıdır. Bunlardan **Count**, sıra nesnesinde bulunan elemanların sayısını geriye dönderen bir özelliktir. *Özellikler C# dilinde sınıflarda bulunan üye değişkenlerin*

değerlerini öğrenmemize ve onların değerlerini değiştirmemize yarayan bir tür fonksiyonlardır. **Count** özelliği eleman sayısını int tipinde dönderir ve sadece okunabilen (readonly) yapıdadır. Özellikler program içinde çağrılırken parantezleri kullanmayız. Eğer sırayı boşaltmak/temizlemek istersek **Clean()** metodu işimizi yarayacaktır. **Clean()** metodu hiçbir parametre almaz ve hiçbirşey döndermez. Herhangi bir sıra nesnesinin içinde bir elemanın olup olmadığını anlamak için **Contains()** metodu kullanılır. Bu metod aranacak nesneyi alır ve geriye **true** veya **false** değerlerini dönderir. İsterseniz aşağıdaki programda **Contains()** ve **Clear()** metodları ile **Count** özelliklerini nasıl kullanabileceğimizi görelim:

```
using System;
using System.Collections; // Queue sınıfı bu isim alanı içinde bulunur.

class Sira_Ornek2
{
    public static void Main()
    {
        // Queue sınıfından bir nesne oluşturalım:
        Queue sıra = new Queue();

        // Nesnemize Enqueue metodu ile değerler girelim:
        sıra.Enqueue(12);
        sıra.Enqueue(3);
        sıra.Enqueue(18);
        sıra.Enqueue(7);
        sıra.Enqueue(20);

        // sıra isimli nesnemizin eleman sayısı:
        Console.WriteLine( "\n sıra nesnemizin eleman sayısı: " + sıra.Count);

        // sıra isimli nesnemizin elemanları:
        Console.WriteLine( "\n sıra nesnemizin elemanları: " );
        DegerleriYaz( sıra );

        //Contains metodunun kullanımı:
        int sayi=7;

        if(sıra.Contains(7))
            Console.WriteLine("Sıramızda " + sayi + " var.");
        else
            Console.WriteLine("Sıramızda " + sayi + " yok.");

        // sıramızın içindeki elemanları silelim:
        sıra.Clear();

        // Sıramızı temizledikten sonra kaç tane
        //eleman bulunduğunu bulup yazalım.
        int elemanSayisi= sıra.Count;

        Console.WriteLine("\n Sıramızda şu anda {0} tane eleman vardır.", elemanSayisi);
    }
}
```

```
}

public static void DegerleriYaz( IEnumerable kolleksiyon )
{
    System.Collections.IEnumerator Enum = kolleksiyon.GetEnumerator();

    while ( Enum.MoveNext() )
        Console.Write( "\\t{0}", Enum.Current );

    Console.WriteLine();
}
}
```

Yığında makalemiz **Clone()**, **ToArray()** ve **Equals()** metodlarının aynıları Queue sınıfı içinde bulabiliriz. **Sizlerin yığın makalesindeki en son örneğini sıra veri tipi için de yazıp kendinizi denemenizi tavsiye ederim.** Herkese iyi çalışmalar.

Visual C# ile Basit Bir Not Defteri Uygulaması

Bu yazımızda konu olarak Windows Form'u seçtim;çünkü bu konuda Türkçe kaynak neredeyse yok, doğru dürüst bir programın yapımını gösteren bir yazı, bir site bulamadım, tabi ki Türkçe bir çok makale var sitelerde, bende onlardan yararlanarak ve deneyerek bir şeyler yaptım ve şimdi bu yaptıklarımı sizinle paylaşıyorum.

Eğer bu yazıyı sonuna kadar okursanız ve kodları sizde yazarsanız, yazının sonuna geldiğiniz Basit Not Defteri adında bir uygulamanız olacak. Önce bu programdan biraz bahsedelim. Adı üstünde bir Not Defteri uygulaması ancak basit hem de çok basit. Yapabildiği şeyler: Yeni dosya yaratmak, var olan dosyaları açmak, dosya kaydetmek... Böyle bir program yapmamın sebebi tabi ki metin editörleri konusunda alternatif oluşturma isteği falan değil, tek sebep benim ilk başlarda çok zorlandığım SaveFileDialog, OpenFileDialog gibi kontroller konusunda örneklemeler yapmak..

Lafı daha fazla uzatmadan artık uygulamaya geçelim. Önce aşağıdaki programı Visual Studio .Net'in Designer'ında oluşturun...



Ben bu resime kullandığım kontrollerin isimlerini de yazdım ki kodları incelerken zorluk çıkmasın. Yalnız burada görünmeyen 2 kontrol daha var. Biri SaveFileDialog (objSave), diğeri OpenFileDialog (objOpen). Bu kontrolleri de ekleyip adlarını parantez içlerindeki gibi yaparsanız sorun çıkmaz...

Şimdi kodlarımıza geçebiliriz. Bu bölümde adım adım ilerleyeceğiz. Menülerdeki tüm başlıkların olaylarını yazacağız.

1)Genel değişkenimizi tanımlama

Bu programda Degisim adında, "bool" yani sadece true ve false değerleri alabilen bir değişken tanımladım. Bu değişken sayesinde kullanıcıyı metnin değişip değişmediği

konusunda uyaracağız, böylece isterse değişen metni kayıt imkanı vereceğiz...

```
private bool Degisim;
```

2)Form1'in onLoad Olayı

Bu olay programımızın açılışında yürütülen olaydır. Burada objSave ve objOpen için bazı ayarlar yapıyoruz ve göstermesini istediğimiz dosyaların uzantılarını giriyoruz.

```
private void Form1_Load(object sender, System.EventArgs e)
{
    objOpen.Filter = "Text Dosyaları(*.txt)|*.txt|Tüm Dosyalar(*.*)|*.*" ;
    objOpen.FilterIndex = 1 ;
    objSave.Filter = "Text Dosyaları(*.txt)|*.txt|Tüm Dosyalar(*.*)|*.*" ;
    objSave.FilterIndex = 1 ;
}
```

3)Kullanılacak metotların tanımlanması...

Ben bu programda sadece 2 tane metot tanımladım. Bunlardan birincisi KayitMekanizmasi, diğeri DegisimUyari . KayitMekanizmasi adı üstünde yazdıklarımızı kaydedecek olan mekanizma, DegisimUyari ise objText içindeki metnin değişimi durumunda programı kapatırken falan bize haber verecek olan kod.

```
public void KayitMekanizmasi(string strVeri)
{
    if (objSave.ShowDialog() == DialogResult.OK)
    {
        StreamWriter Kayitci = new
        StreamWriter(Environment.GetEnvironmentVariable("mydocuments")
        +objSave.FileName.ToString(),false,System.Text.Encoding.Unicode);
        Kayitci.Write(strVeri);
        Kayitci.Close();
        Degisim = false;
    }
}
```

Önce yukarıdaki kodu biraz inceleyelim. Burada önce bi if kontrolü görüyorsunuz. Bu kontrolün amacı, Kayıt ekranı açıldığı zaman kullanıcı "OK" düğmesine tıklayıp tıklamadığını kontrol etmek. Eğer "OK"e tıkladı ise programımız bir adet StreamWriter oluşturuyor. Kayitci adındaki bu Writer Environment.GetEnvironmentVariable("mydocuments") bu kod ile ayarlı olan Belgelerim klasörüne gidiyor otomatik olarak. objSave.FileName ise bizim Kayıt Ekranın da dosyaya verdiğimiz ismi bize döndürüyor. Son olarak ise bu satırda Unicode bir kodlama yaptığımız gösteriyoruz. Bunu yazmazsanız Türkçe karakterlerinizin yerinde yeller estiğini görürsünüz.

Kayitci.Write(strVeri) satırı ile gelen veriyi kaydediyor ve StreamWriter nesnesini kapatıyor. Degisim değerini ise true olarak atıyor. Bunun nedeni değişim oldu ve ben bunu gördüm demek. Kullanıcıya haber vermeye gerek yok anlamına gelecek.

Şimdi devam edelim.

```
public bool DegisimUyari()
{
    if (MessageBox.Show("Dosyanızda bir değişiklik oldu kaydetmek ister misiniz?", "Değişiklik Var", MessageBoxButtons.YesNo, MessageBoxIcon.Exclamation) == DialogResult.Yes)
    {
        return true;
    }
    else
    {
        Degisim = false;
        return false;
    }
}
```

Yukarıdaki kodda ise tipik bir MessageBox kullanımı görüyorsunuz. Buradaki metodumuz birde değer döndürüyor. Bir bool değeri döndürüyor. Bu dönen değer ile biz az sonra kullanıcının çıkan mesaj kutusunda dosyayı kaydetmek isteyip istemediğini anlayacağız.

```
MessageBox.Show("Dosyanızda bir değişiklik oldu kaydetmek ister misiniz?", "Değişiklik Var", MessageBoxButtons.YesNo, MessageBoxIcon.Exclamation) == DialogResult.Yes)
```

Bu satırı biraz incelemek lazım. Burada ilk overload (Overload metodlara parantezler içinde yollanan veri demek.) mesaj kutusunda görünecek olan yazı, ikincisi bu mesaj kutusunun başlığı, üçüncüsü mesaj kutusu üzerinde ki "Evet", "Hayır" düğmeleri ve son olarak mesaj kutusundaki simge. Ancak kodlara bakmaya devam ettiğimizde bir karşılaştırma görüyoruz ("==" ifadesi) DialogResult.Yes , aslında açıklamaya bile gerek yok. Eğer kullanıcı "Evet"e tıkladı ise demek. Asıl kodlarda bu durumda bir "true" ifadesi döndürüldüğünü görebilirsiniz. Biz daha sonra bunu kontrol ederek KayitMekanizmasi metodumuzu çağıracağız.

4) Yeni düğmesi

Menümüzdeki "Yeni" düğmesine tıkladığımızda olacak olayları gireceğiz. Bunun için bu düğmeye Designer'dan çift tıklayınız.

```
private void menuItem2_Click(object sender, System.EventArgs e)
{
    if (Degisim == false)
    {
        objText.Clear();
    }
    else
    {
        if (DegisimUyari())
        {
            KayitMekanizmasi(objText.Text);
        }
    }
}
```

```

        objText.Clear();
        Degisim = false;
    }
    else
    {
        objText.Clear();
        Degisim = false;
    }
}
}

```

Burada önce Degisim değerini kontrol ediyoruz. Eğer değer "false" ise yani değişim yoksa ya bu dosya önceden kaydedilmiştir ya da yeni açılmıştır. O zaman içeriğinin temizlenmesinde bir sorun yok.

Eğer değer "true" ise biraz karışıyor ortalık. Önce kullanıcıyı uyarmak için DegisimUyari() çalıştırılıyor. Eğer kullanıcı kayıt etmek istiyorsa, KayitMekanizmasi() çalıştırılıyor, ekran temizleniyor ve Degisim değeri false oluyor. Eğer kullanıcı kayıt etmek istemiyorsa içerik temizleniyor ve Degisim değeri yine false oluyor. Böylece yeni bir dosya açma işlemlerini hallettik.

5) Varolan dosyayı açma

Metin editörünüz ile daha önce var olan bir dosyayı açmak istersiniz diye böyle bir özellik ekledik birde. Menümüzde "Aç"a çift tıklayın ve tıklama olayına aşağıdaki kodları girin.

```

private void menuItem3_Click(object sender, System.EventArgs e)
{
    if (Degisim == true)
    {
        if (DegisimUyari())
        {
            KayitMekanizmasi(objText.Text);
        }
    }
    if (objOpen.ShowDialog() == DialogResult.OK)
    {
        FileInfo strKaynak = new
        FileInfo(Environment.GetEnvironmentVariable("mydocuments")
        +objOpen.FileName.ToString());
        StreamReader Okuyucu = strKaynak.OpenText();

        objText.Text = Okuyucu.ReadToEnd();
        Degisim = false;
        Okuyucu.Close();
    }
}

```

Bu kodlarda da önce değişim var mı diye bakıyoruz. Yani amacımız kullanıcının yazdığı metni yanlışlıkla bastığı bir düğme yüzünden kaybetmesini engellemek. Eğer değişim varsa ve uyarıdan "true" değeri dönerse kaydediyoruz, aksi halde herhangi bir şey

yapmıyoruz.

Bundan sonra yukarıda SaveFileDialog için yaptığımız benzer şeyleri yapıyoruz. Yani .ShowDialog() metodunu çağırıyoruz. Kullanıcı OK'e tıklayınca kodlarımız devam ediyor. Ancak burada yukarıdakinden farklı kodlar var. Dosya okumak için çok farklı yöntemler var. Yazmak içinde tabi ki. Mesela StreamWriter'ın StreamReader'ı da var ve ben burada bunu kullandım. Eğer kodları incellerseniz biraz farklı olduğunu göreceksiniz. Çünkü burada FileInfo diye de bir şey var. FileInfo bu tür dosya işlemcilerine yardımcı olur. strKaynak değişkenine atadığımız nesnemizde StreamWriter daki gibi path gösterip dosyamızı açıyoruz. Burada objOpen.FileName'den gelen veri, kullanıcının açmak istediği dosya.

StreamReader nesnesini de oluşturup strKaynak.OpenText() ile metin dosyamızı açıyoruz. Yalnız burada bir noktaya dikkat çekmek istiyorum. Ben burada açılacak dosyanın bir .txt dosyası olduğunu bildiğim için .OpenText'i kullandım. Yoksa başka versiyonları da mevcut. Bu nesneyi de oluşturduktan sonra objText'e Okuyucu.ReadToEnd ile baştan sonra tüm veriyi okuyup aktarıyoruz. Değişimden haberimiz olduğunu programa bildirip, nesnelerimizi kapatıyoruz...

6) Kaydet düğmesi

Kullanıcı çalışmasını kaydetmek istediği zaman bu düğmeye tıklayabilir. Çok kısa bir kodu var. Zaten asıl işi yapan KayitMekanizmasi(), biz sadece onu çağıracağız şimdi.

```
private void menuItem4_Click(object sender, System.EventArgs e)
{
    KayitMekanizmasi(objText.Text);
    Degisim = false;
}
```

Burada açıklanacak bir kod yok. Gördüğünüz gibi ...

7) Kapat Düğmesi

Kullanıcı programı kapatmak isteyebilir ve bunun için Dosya menüsündeki Kapat düğmesini kullanabilir. O zaman bu düğmeye de bir olay atamamız lazım. Şimdi çift tıklayın ve aşağıdaki kodları yazın.

```
private void menuItem6_Click(object sender, System.EventArgs e)
{
    Close();
}
```

Bu kod çok basit. Sadece Close() metodunu çağırıyor. Bu özel tanımlı bir metodur ve o form penceresinin kapanmasını sağlar. Şimdi aklınıza gelebilir ya içeride kaydedilmemiş veri varsa hiç kontrol etmedik. O zaman biraz sabır, ona da bakacağız...

8) Kapanmadan önce kontrol

Kullanıcımız programı çok farklı şekillerde kapatabilir. Alt + F4 kombinasyonu, köşedeki X ile kapatabilir ya da Kapat düğmemize tıklar; ancak az önce de dediğimiz gibi ya içeride veri varsa. O zaman bu veri için bi kontrol yapmamız lazım. Form'ların "Closing" adında

olayları vardır. Bu form kapatılmadan hemen önce yapılacakları belirler. Biz buna bazı olaylar atıyoruz şimdi.

```
private void Form1_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
{
    if (Degisim == true)
    {
        if (DegisimUyari())
        {
            KayitMekanizmasi(objText.Text);
            Close();
        }
    }
    else
    {
        Close();
    }
}
```

Burada yapılanlardan farklı olan hiç bir şey yok. Degisim değerini kontrol ediyoruz ve ona göre işlem yapıyoruz..

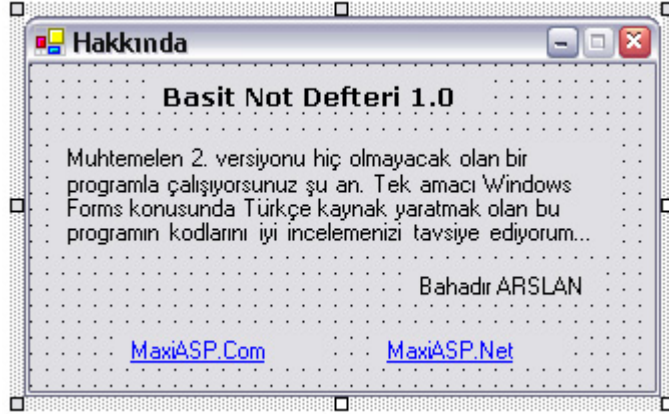
9) Son bir metod...

Asıl en önemli şeyi yapmadık sanıyorum. Örneğin kullanıcı programa bir veri girdiğinde yani herhangi bi yazı yazdığında Degisim değeri değişmedi. O zaman bunu halledelim. objText'in TextChanged adında bir olayı var. Şimdi o olay kodları içine aşağıdaki tek satırlık kodu yazıyoruz.

```
private void objText_TextChanged(object sender, System.EventArgs e)
{
    Degisim = true;
}
```

Evet, böylece olayın iş yapan kısmı bitti..

Ancak menülerimiz arasında hiç ilgilenmediğimiz bir düğme var. Hakkında. Bu aslında en gereksiz şey belki ama bir programcının en çok önemseddiği bölüm :). Bunun için basit bir form yaratınız. Ben aşağıdaki formu oluşturdum ve adını "hakkında" yaptım.



Burada altta iki tane de link var. Biri MaxiASP.Com 'a biri MaxiASP.Net'e yönlendirilmiş durumda. Bunlara tıklandığında tarayıcımızın açılıp sitelere gitmemizi sağlayacak kodlarda aşağıda.

```
private void linkLabel1_LinkClicked(object sender,  
System.Windows.Forms.LinkLabelLinkClickedEventArgs e)  
{  
    System.Diagnostics.Process.Start("http://www.maxiasp.com");  
}  
  
private void linkLabel2_LinkClicked(object sender,  
System.Windows.Forms.LinkLabelLinkClickedEventArgs e)  
{  
    System.Diagnostics.Process.Start("http://www.maxiasp.net");  
}  
}
```

Gerçekten çok uzun bir yazı oldu. Eğer her şey yolunda gitti ise şu an canavar gibi çalışan bir "Basit Not Defteriniz" var. Başka yazılarda görüşmek üzere.

C#'ın Gelecekteki Özellikleri

Bildiğiniz gibi C# dili 2001 yılında Microsoft tarafından çıkarılan ve nesne yönelimli programlama tekniğine %100 destek veren bir programlama dilidir. C#, programcılara sunulduğundan beri bir çok programcının dikkatini çekmiştir. Bu ilgide en önemli neden herhalde C# dilinin kendinden önce çıkarılmış olan JAVA ve C++ dillerini örnek almasıdır. Evet C# modern çağın gerektirdiği bütün yazılım bileşenlerini içermekle beraber eski programlama dillerinde bulunan iyi özellikleri de yapısında barındırmaktadır. Microsoft ve C# dil tasarımcıları her geçen gün yeni piyasa araştırmaları yaparak dile katabilecekleri özellikleri tartışmaktadırlar. Bu amaçla C# dilinin tasarımcıları yakın bir zaman içinde C# diline eklemeyi düşündükleri yeni özellikleri bildirmişlerdir. Bu yazıda muhtemelen "VS.NET for Yukon(VS.NET Everett'ten sonraki versiyon)" ile birlikte uygulamaya konulacak C# dilinin muhtemel özelliklerini özetlemeye çalışacağım. Bu bildirinin tamamını C# topluluğunun resmi sitesi olan www.csharp.net adresinden okuyabilirsiniz.

C# diline yakın bir zamanda eklenilmesi düşünülen özellikler 4 ana başlık altında toplanmıştır. Bu özellikler temel olarak aşağıdaki gibidir.

- 1 - Generics (Soysal Türler)
- 2 - Iterators
- 3 - Anonymous Methods (İsimsiz-Anonim- Metotlar)
- 4 - Partial Types (Kısmi Türler)

Bu yazıda yukarıda başlıklar halinde verilen her bir konuyu ayrıntılı olarak inceleyip, programcıya ne gibi faydalar sağlayabileceğini ve programların performansına nasıl etki edeceğine değineceğim.

1 - Generics

Profesyonel programlamada, türden bağımsız algoritma geliştirme önemli bir tekniktir. Türden bağımsız algoritmalar geliştirici için büyük kolaylıklar sağlamaktadır. Söz gelimi iki int türden sayının toplanmasının sağlayan bir fonksiyonu yazdıktan sonra aynı işlemi iki double türden sayı için tekrarlamak zaman kaybına sebep olacaktır. C++ dilinde türden bağımsız algoritma kurabilmek için şablon(template) fonksiyonları ve şablon sınıfları kullanılmaktadır. C#, türden bağımsız algoritma geliştirmeye doğrudan destek vermiyor olsada dolaylı yollardan türden bağımsız işlemler yapabilmek mümkündür. Bu işlemler

C#'ta "Her şey bir Object'tir" cümlesinin altında yatan gerçeğe halledilmektedir. C#'ta her şeyin bir nesne olması ve her nesnenin ortak bir atasının olması ve bu atanın da Object sınıfı olması bu cümlemin altında yatan gerçektir. Dolayısıyla herhangi bir türe ait referansı Object referanslarına atayabiliriz. Yani bir bakıma türden bağımsız bir işlem gerçekleştirmiş oluyoruz. Söze gelimi Object türünden bir parametre alan bir fonksiyonu dilediğimiz bir nesne referansı geçebiliriz. Temel(base) sınıfa ait referanslara türeyen(inherited) sınıf referanslarını atayabilmek nesne yönelimli programlama tekniğinin sunduğu bir imkandır.

C#'ta Object referanslarına istenilen türden referanslar atanabilir. Bu, büyük bir imkan gibi görülsede aslında bazı dezavantajları da beraberinde getiriyor. Çünkü çalışma zamanında Object türüne atanmış referanslar orjinal türe tekrar geri dönüştürülmektedir. Kısaca unboxing olarak bilinen bu işlem özellikle değer(value) ve referans(reference) türleri arasında yapıldığında önemsenerek büyüklükte bir performans kaybı meydana gelmektedir. Çünkü değer ve referans türleri belleğin farklı bölgelerinde saklanmaktadır. Bu durum boxing ve unboxing işlemlerinin çalışma zamanında farklı bellek bölgeleri arasında uzun sürebilecek veri transferlerine sebep olur. Bu tür bir performans kaybını bazı veri yapıları için önlemek için C# dil tasarımcıları Generics isimli bir kavramın dile eklenmesini öngörmüşlerdir. Bu sayede bazı veri yapılarında özellikle .NET sınıf kütüphanesindeki System.Collections isim alanında bulunan veri yapılarında epeyce performans kazancı elde edilecektir.

İsterseniz basit bir yığın(stack) sınıfı üzerinden "generics" kavramının sağlayacağı yararları ve boxing/unboxing işlemlerinin etkisini inceleyelim.

.NET sınıf kütüphanesinde de bulunan Stack sınıfı içinde her türden veri bulunduran ve istenildiğinde bu verilere FIFO(İlk giren ilk çıkar) algoritmasına göre veri çekilebilen bir veri yapısıdır. .NET'teki Stack sınıfı ile bütün veri türlerine ait işlemleri yapabilmek için Stack sınıfındaki veri yapısı Object olarak seçilmiştir. Eğer bu böyle olmasaydı Stack sınıfının her bir tür için ayrı ayrı yazılması gerekecekti. Bu mümkün olsa bile her şey bitmiş olmayacaktı. Çünkü Stack sınıfı kullanıcıyı tanımlayacağı türleri barındıracak duruma gelmez. İşte bütün bu sebeplerden dolayı Stack veri yapısında saklanan veriler Object olarak seçilmiştir. Buna göre Stack sınıfının arayüzü aşağıdaki gibidir.

```
class Stack
{
    private int current_index;
    private object[] elemanlar = new object[100];

    public void Push(object veri)
    {
        .
        .
        elemanlar[current_index] = veri;
        .
        .
    }

    public object Pop()
    {
        .
        .
        return elemanlar[current_index];
        .
        .
    }
}
```

```
}
```

Bildirilen bu Stack sınıfının elemanı Object türünden olduğu için Push() metodu ile istediğimiz türden veriyi saklayabiliriz. Aynı şekilde Pop() metodu ile bir veri çekileceği zaman veri Object türünden olacaktır. Pop() metodu ile elde edilen verinin gerçek türü belli olmadığı için tür dönüştürme operatörü kullanılır. Örneğin,

```
Push(3);
```

şeklinde yığına eklenen veriyi tekrar elde etmek için

```
int a = (int)Pop();
```

biçiminde bir tür dönüşümü yapmamız gerekir. Bu işlemler kendi tanımlayacağımız özel sınıflar içinde geçerlidir. Ancak int ve double gibi temel veri türlerindeki performans kaybı daha fazladır. Çünkü Push(3) şeklindeki bir çağrıda boxing işlemi gerçekleşirken Pop() metodunun çağrılmasında unboxing işlemi gerçekleşir. Üstelik bu durumda Pop() metodunun geri dönüş değerini byte türüne dönüştürmeye çalışırsak derleme zamanında herhangi bir hata almayız. Bu da çalışma zamanında haberimiz olmadan bazı veri kayıplarının olabileceğini gösterir. Kullanıcı tanımlı sınıflar ilgili bir yığın kullanıyorsak Pop() metodunun geri dönüş değerini farklı bir kullanıcı tanımlı sınıfa çeviriyorsak bu sefer de derleme zamanında hata alınmaz, ancak çalışma zamanında "invalid cast operation" istisnai durumu meydana gelir.

Bütün eksi durumlardan kurtulmak için generics(soysal tür)'lerden faydalanılabilir. Soysal türler C++ dilindeki şablon sınıflarının bildirimi ile benzerdir. Bu tür sınıf bildirimlerine **parametrelili tip** de denilmektedir. Parametrelili tipler aşağıdaki gibi bildirilir.

```
class Stack
{
    private int current_index;
    private Veri türü[] elemanlar;

    public void Push(Veri türü veri)
    {
        .
        .
        elemanlar[current_index] = veri;
        .
        .
    }

    public Veri türü Pop()
    {
        .
        .
        return elemanlar[current_index];
        .
        .
    }
}
```

ile stack sınıfının hangi türden verileri tutacağı stack nesnesini oluşturacak programcıya bırakılmıştır. Örneğin int türden verileri saklayacak bir yığın aşağıdaki gibi oluşturulur.

```
Stack<int> yığın = new Stack<int>;
```

Yukarıdaki şekilde bir yığın oluşturulduğunda Stack sınıfının bildirimindeki **Veri türü** ifadeleri int türü olarak ele alınacaktır. Dolayısıyla Pop() metodu ile yığından bir eleman çıkarılıp aşağıdaki gibi başka bir değişkene atanmak istendiğinde tür dönüştürme operatörünü kullanmaya gerek yoktur. Bu da boxing ve unboxing işlemlerinin gerçekleşmediği anlamına gelir ki istediğimiz de buydu zaten.

```
Stack<int> yığın = new Stack<int>;
```

```
yığın.Push(3); // Boxing işlemi gerçekleşmez.
```

```
int a = yığın.Pop(); //Unboxing işlemi gerçekleşmez.
```

Aynı şekilde yığınınızın double türden verileri saklamasını istiyorsak int yerine double kullanmalıyız. Bu durumda çalışma zamanında hem int hem de double verileri tutan yığın sınıfları oluşturulacaktır. Biz tek bir yığın sınıfı bildirmiş olmamıza rağmen çalışma zamanı bizim için ayrı iki yığın sınıfı oluşturur.

Soysal türleri kendi tanımladığımız sınıflar içinde oluşturabiliriz. Örneğin Musteri isimli bir sınıfın verilerini yığında tutmak için yığın sınıfını aşağıdaki gibi oluşturmalıyız.

```
Stack yığın = new Stack;
```

Bu durumda yığına sadece Musteri nesneleri eklenebilir. Yani yığın.Push(3) şeklindeki bir kullanım derleme aşamasında hata verecektir. Aynı zamanda yığından çekilecek veriler de Musteri türündendir. Dolayısıyla tür dönüşümü uygun türler arasında olmalıdır.

Yığın sınıfı yukarıda anlatılan şekilde kullanıldığında yığındaki elemanların belirli bir türden olduğu garanti altına alınır. Böylece Musteri türünden nesneleri tutan bir yığına "3" gibi bir sayıyı ekleyemeyeceğimiz için daha gerçekçi programlar yazılır.

Stack örneğinde sadece bir tane parametre türü kullandık. Soysal türlerde istenilen sayıda parametrelili tür kullanılabilir. Örneğin Hashtable sınıfındaki Deger ve Anahtar ikilisi aşağıdaki gibi parametrelili tür olarak bildirilebilir.

```
public class Hashtable
{
    public void Add(AnahtarTuru anahtar, DegerTuru deger)
    {
        ....
    }

    public DegerTuru this[AnahtarTuru anahtar]
    {
        ....
    }
}
```

Yani bir Hashtable nesnesi oluşturulacağı zaman her iki parametre türü de belirtilmelidir. Örneğin Anahtar türü int olan ve değer türü Musteri sınıfı olan bir Hashtable nesnesi aşağıdaki gibi oluşturulabilir.

```
Hashtable<int,Musteri> hashtable = new Hashtable<int,Musteri>;
```

Not : Parametre sayısını aralarına virgül koyarak dilediğimiz kadar artırabiliriz.

Soysal türlerin saydığımız avantajlarının yanında bu haliyle bazı dezavantajları ve kısıtlamaları da vardır. Söz gelimi Hashtable sınıfının bildirimi içinde AnahtarTuru verisinin bazı elemanlarını bir ifade de kullanmak istiyoruz; derleyici hangi AnahtarTuru parametreleri türünün hangi türden olduğunu bilmediği için bu durumda sadece Object sınıfının ait metotlar ve özellikler kullanılabilir. Mesela Hashtable sınıfının Add metodu içinde anahtar parametresi ile CompareTo() metodunu kullanmak istiyorsak CompareTo metodunun bildirildiği IComparable arayüzünü kullanarak aşağıdaki gibi tür dönüşümü yapmalıyız.

```
public class Hashtable
{
    public void Add(AnahtarTuru anahtar, DegerTuru deger)
    {
        switch(((IComparable)anahtar).CompareTo(x))
        {
            }
        }
    }
}
```

Hashtable sınıfının Add() metodu yularındaki şekilde bildirilse bile hala eksik noktalar var. Mesela AnahtarTuru parametresi eğer gerçekten IComparable arayüzünü uygulamıyorsa switch ifadesi içinde yapılan tür dönüşümü geçersiz olacaktır ve çalışma zamanında hata oluşacaktır. Çalışma zamanında meydana gelebilecek bu tür hataları önlemek için yapılabilecek tek şey AnahtarTuru parametresinin IComparable arayüzünü uyguluyor olmasını zorlamaktır. Bu işlemi yapmak için AnahtarTuru parametresine çeşitli **kısıtlar(constraints)** getirilir. Aşağıdaki Hashtable sınıfında AnahtarTuru parametresinin IComparable arayüzünü uygulaması gerektiği söylenmektedir. Bu kısıt için **where** anahtar sözcüğü kullanılır.

```
public class Hashtable<AnahtarTuru, DegerTuru> where AnahtarTuru : IComparable
{
    public void Add(AnahtarTuru anahtar, DegerTuru deger)
    {
        switch(anahtar.CompareTo(x))
        {
            }
        }
    }
}
```

Dikkat ettiyseniz uygulanan kısıttan sonra switch ifadesi içinde anahtar değişkeni üzerinde tür dönüşümü işlemi yapmaya gerek kalmamıştır. Üstelik kaynak kodun herhangi bir noktasında Hashtable nesnesini IComparable arayüzünü uygulamayan bir AnahtarTuru parametresi ile oluşturursak bu sefer ki hata derleme zamanında oluşacaktır.

Not : parametrelili türler üzerindeki kısıt sadece arayüz olmak zorunda değildir. Arayüz yerine sınıflar da kısıt olarak kullanılabilir.

Bir parametrelili türe birden fazla arayüz kısıtı konabileceği gibi aynı sınıftaki diğer parametreleri türler için de kısıt konulabilir. Ancak bir parametrelili tür için ancak sadece bir tane sınıf kısıt olabilir. Örneğin aşağıdaki Hashtable sınıfında DegerTuru Musteri sınıfından tremiş olması gerekirken, AnahtarTuru hem IComparable hemde IEnumerable arayüzünü uygulamış olması gerekir.

```
public class Hashtable<AnahtarTuru, DegerTuru> where
AnahtarTuru : IComparable
AnahtarTuru : IEnumerable
DegerTuru : Musteri
{
    public void Add(AnahtarTuru anahtar, DegerTuru deger)
    {
        switch(anahtar.CompareTo(x))
        {
            }
        }
    }
}
```

2 - Iterators

Bir dizinin elemanları üzerinde tek tek dolaşma işlemine iterasyon denilmektedir. Koleksiyon tabanlı nesnelerin elemanları arasında tek yönlü dolaşmayı sağlayan foreach döngü yapısının bizim tanımlayacağımız sınıflar için de kullanılabilmesi için sınıfımızın bazı arayüzleri uyguluyor olması gerekir. foreach döngüsü derleme işlemi sırasında while döngüsüne dönüştürülür. Bu dönüştürme işlemi için IEnumerator arayüzündeki metotlardan ve özelliklerden faydalanılmaktadır. Bu dönüştürme işleminin nasıl yapıldığına bakacak olursak :

```
ArrayList alist = new ArrayList();

foreach(object o in alist)
{
    BiseylerYap(o);
}

// Yukarıdaki foreach bloğunun karşılığı aşağıdaki gibidir.

Enumerator e = alist.GetEnumerator();

while(e.MoveNext())
{
    object o = e.Current
    BiseylerYap(o);
}
```

foreach döngüsü yapısı için gerekli olan arayüzlerin uygulanması özellikle ağaç yapısı şeklindeki veri türleri için oldukça zordur. Bu yüzden C# sınıfların foreach yapısı ile nasıl kullanılacağına karar vermek için yeni bir yapı kullanacaktır.

Sınıflarda, foreach anahtar kelimesi bir metot ismi gibi kullanılarak sınıfın foreach döngüsünde nasıl davranacağını bildirebiliriz. Her bir iterasyon sonucu geri döndürülecek değeri ise **yield** anahtar sözcüğü ile belirtilir. Örneğin her bir iterasyonda farklı bir tamsayı değeri elde etmek için sınıf bildirimi aşağıdaki gibi yapılabilir.

```
public class Sınıf
{
    public int foreach()
    {
        yield 3;
        yield 4;
        yield 5;
    }
}
```

Yukarıda bildirilen Sınıf türünden nesneler üzerinde foreach döngüsü kullanıldığında iterasyonlarda sırasıyla 3,4 ve 5 sayıları elde edilecektir. Buna göre aşağıdaki kod parçası ekrana 345 yazacaktır.

```
Sınıf deneme = new Sınıf();

foreach(int eleman in deneme)
{
    Console.Write(eleman);
}
```

Çoğu durumda foreach yapısı ile sınıfımızın içindeki bir dizi üzerinde iteratif bir şekilde dolaşmak isteyeceğiz. Bu durumda foreach bildirimi içinde ayrı bir foreach döngüsü aşağıdaki gibi kullanılabilir.

```
public class Sınıf
{
    private int[] elemanlar;

    public int foreach()
    {
        foreach(int eleman in elemanlar)
        {
            yield eleman;
        }
    }
}
```

Yukarıdaki Sınıf nesneler ile foreach döngüsü kullanıldığında her bir iterasyonda elemanlar dizisinin bir sonraki elemanına ulaşılır.

Gördüğümüz gibi programcının bildireceği sınıflar da foreach döngüs yapısını kullanabilmek için eskiden olduğu gibi IEnumerator arayüzün uygulamaya gerek kalmamıştır. Bu işlemi derleyici bizim yerimize yapar.

3 - Anonymous Metotlar(İsimsiz Metotlar)

İsimsiz metotlar, bir temsilciye ilişkin kod bloklarını emsil eder. Bildiğiniz gibi temsilciler yapısında metot referansı tutan veri yapılarıdır. Bir temsilci çağırımı yapıldığında

temsilcinin temsil ettiği metot çalıştırılır. Özellikle görsel arayüzlü programlar yazarken event tabanlı programlama tekniği kullanılırken temsilcilerin kullanımına sıkça rastlanır. Örneğin bir Button nesnesine tıklandığında belirli bir kod kümesinin(metot) çalıştırılması için temsilci veri yapısından faydalanılır. Sözelimi Button nesnesinin tıklanma olayı meydana geldiğinde Click isimli temsilcisine yeni bir temsilci atanır. Ne zaman button nesnesinin Click olayı gerçekleşse ardından hemen temsilcinin temsil ettiği metot çağrılır. Buna bir örnek verecek olursak;

```
public class Form
{
    Button dugme;

    public Form
    {
        dugme = new Button();
        dugme.Click += new EventHandler(OnClick);
    }

    void OnClick(object sender, EventArgs e)
    {
        ....
    }
}
```

Yukarıdaki koddan da görüldüğü üzere temsilci ile temsilcinin temsil ettiği metotlar ayrı yerlerdedir. İsimsiz metotlarla bu işlemi biraz daha basitleştirmek mümkündür. Temsilci oluşturulduktan sonra açılan ve kapanan parantezler arasına temsilci çağrıldığında çalıştırılacak kodlar yazılabilir. Yukarıdaki örneği isimsiz metot ile yapacak olursak :

```
public class Form
{
    Button dugme;

    public Form
    {
        dugme = new Button();
        dugme.Click += new EventHandler(object sender, EventArgs e)
        {
            //çalıştırılacak kodlar.
        };
    }
}
```

Tanımlanan kod bloğundan sonra noktalı vürgünün eklenmiş olduğuna dikkat edin. Temsilci bloğundaki kodlar normal metotlardan biraz farklıdır. Normal kod blokları ile benzer özellikler taşır. Yukarıdaki temsilci kod bloğunda, blok dışında tanımlanan değişkenlere erişebilmek mümkündür. Ayrıca olay argümanlarının da(sender,e) EventHandler türünün parantezleri içinde yazıldığına dikkat edin. Bir önceki versiyonda olay argümanlarının yerine temsil edilen metodun ismi yazılmıştı.

Peki isimsiz metotlar nasıl çalıştırılmaktadır? İsimsiz metot tanımı ile karşılaşan derleyici tekil isme sahip bir sınıf içinde tekil isme sahip bir metot oluşturur ve isimsiz metot gövdesindeki kodlara bu tekil metot içinden erişilir. Temsilci nesnesi çağrıldığında, derleyicinin ürettiği bu metot ile isimsiz metodun bloğundaki kodlar çalıştırılır.

4 - Partial Types (Kısmi Türler)

Kısmi türler yardımıyla bir sınıfın elemanlarını farklı dosyalarda saklamak mümkündür. Örneğin Dosya1.cs ve Dosya2.cs aşağıdaki gibi olsun.

```
//Dosya1.cs
```

```
public partial class deneme
{
    public void Metot1
    {
        ...
    }
}
```

```
//Dosya2.cs
```

```
public partial class deneme
{
    public void Metot2
    {
        ...
    }
}
```

Yukarıdaki iki dosyayı aynı anda derlediğimizde eğer kısmi türler kavramı olmasaydı derleme zamanında hata alırdık. Çünkü aynı isim alanında birden fazla aynı isimli sınıf bildirimi yapılmış. Halbuki kısmi türler ile bu iki sınıf bildirimi aynı sınıf olarak ele alınır, ve birleştirilir. Yani deneme isimli sınıfın Metot1() ve Metot2() adında iki tane metodu olmuş olur.

Bir türe ait elemanları tek bir dosya içinde toplamak Nesne Yönelimli Programlama açısından her ne kadar önemli olsada bazen farklı dosyalarla çalışmak kodlarımızın yönetilebilirliğini artırabilmektedir.

Not : Bu yazı "MSDN Magazine" deki "Future Features of C#" başlıkla bildiri baz alınarak hazırlanmıştır.

Kaynak Kodunuzu XML ile Süsleyin

Büyük yazılım projelerinde en önemli aktivitelerden birisi proje bazında iyi bir dökümantasyon yapmaktır; proje analiz sürecindeki dökümantasyon genellikle standart olan UML diyagramları ile yapılmaktadır, tabi işin bir de geliştiriciye bakan tarafı vardır. Projenin en nihayi sonu kod yazmak olduğuna göre kodların dökümantasyonu da en az analiz sürecindeki dökümantasyon kadar önemlidir. Bu yazıda .NET ile birlikte ön plana çıkan XML yorum formatı ile kodlarımızı nasıl dökümente edebileceğimizi inceleyeceğiz.

Bildiğiniz gibi kodlarımıza yorum satırı koymamızdaki en büyük amaç kodların başkası tarafından kolaylıkla anlaşılabilir hale gelmesini sağlamaktır. Bazen bu işlemi kendimiz içinde yapmak durumunda kalabiliriz, zira bir çok karmaşık uygulamada yazdığımız kaynak koda yıllar sonra belkide aylar sonra baktığımızda vakt-i zamanında neler düşündüğümüz hemen aklımıza gelmeyebilir. Bu durumda en büyük yardımcımız o zamanlar tembellik etmeden yazdığımız yorum satırları olacaktır. Eğer kendinize yorum satırı ekleme alışkanlığını kazandırırsanız bunun getirisini ileride mutlaka göreceksiniz. Peki ne kadar yorum satırı gereklidir? Bu sorunun cevabı size ve yazdığınız kodların karmaşıklığına göre değişir. Eğer şiir gibi kod yazıyorum diyorsanız belkide bir cümlelik yorum satırı işinizi görebilir, yok arap saçı gibi kod yazıyorum diyorsanız belkide yazdığınız kod satırı sayısından daha fazla yorum yazmak zorunda kalabilirsiniz.

.NET bir çok meselede olduğu gibi yorum ekleme mekanizmasını da estetik bir şekilde çözmüştür. Çok değil daha bir kaç yıl öncesine kadar kaynak kodlarımızdaki yorum satırları // ve /* */karakterleri ile belirtiliyordu. .NET bu eski yorum yazma şeklini desteklemekle birlikte XML formatındaki yorum ekleme mekanizmasıyla ön plana çıkmaktadır. XML sayesinde artık kodlarımızdaki yorumlar standart hale getirilmiştir. Böylece bir XML yorumunda belirli bir etiketi gördüğümüzde o etiketin içindeki açıklamanın neyi ifade ettiğini anlarız. Aynı zamanda VS.NET kodlarımızdaki XML yorumlarını ayrıştırarak saf bir XML dosyası da üretebilmektedir. Bu sayede XML formatındaki yorum dosyasını istediğimiz sistem ile rahatlıkla entegre edebiliriz, söz gelimi proje yöneticisine XML dosyasındaki yorum bilgilerini HTML formatında sunabiliriz.

XML Yorum Satırları

C#'ta XML yorum satırları " /// (3 adet slash karakteri) " ile başlayan satırlarda yazılır. Önceden belirlenmiş bir takım standart XML etiketleri vardır, öyleki bu etiketler aynı zamanda ECMA tarafından da standart olarak kabul edilmiştir. Ancak XML etiketlerini programcı istediği şekilde şirketin ihtiyaçlarına yada kendi ihtiyaçlarına göre genişletebilir. XML yorum yazmadaki belkide tek kısıt XML sözdizimine uyma şartıdır. XML sözdizimine göre açılan bütün etiketler kapanmalıdır.

En çok kullanılan önceden tanımlı XML etiketleri **<param>**, **<remarks>**, **<summary>** ve **<returns>** etiketleridir. Bu etiketlerin bazıları intellisense ve "Object Browser" programı tarafından kullanılmaktadır. Örneğin VS.NET editöründe bir nesne yaratıldığında

nesne türüne ait yapıcı metottaki parametrelerin kısa açıklaması editör penceresinde gösterilir. Aynı şekilde kendi yazdığımız sınıflar içinde bu açıklamaların çıkmasını istiyorsak XML yorum etiketlerini kullanmamız gerekir.

XML yorum etiketleri tür bazında, metot bazında ve parametre bazında olabilir. Tür bazında yorum ekleme işlemi sınıflar, temsilciler, indeksleyiciler, olaylar, numaralandırmalar(enums) ve yapılar(struct) için uygulanabilir. Metot bazındaki yorumlar herhangi bir metodun bildiriminden önce yapılır. Parametre bazındaki yorumlar ise bir metodun parametreleri ve geri dönüş değerleri ile ilgilidir.

Açıklayıcı olması açısından örnek bir sınıf üzerinde XML yorumlarını ve VS.NET gibi akıllı editörlerde bu yorumların ne gibi etkilerinin olduğunu inceleyelim.

Şimdi yeni bir "Class Library" projesi açıp aşağıdaki sınıf bildirimini yazın.

```
using System;

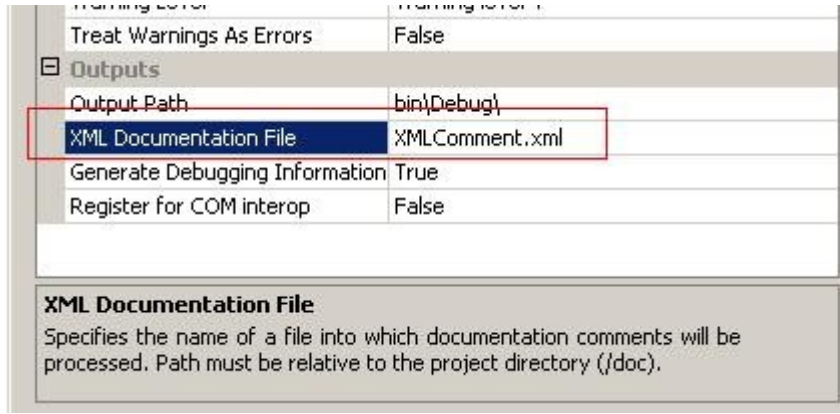
namespace XMLYorum
{
    ///<summary>
    /// Cebir sinifi bazı özel matematiksel işlemleri
    /// yapmak için çeşitli statik metotlar sunar.
    ///</summary>
    public class Cebir
    {
        /// <remarks>
        /// Mutlak Deger Alma Islemi
        ///</remarks>
        ///<summary>
        /// Parametre olarak gelen sayinin
        /// Mutlak Degerini alır.
        ///</summary>
        ///<param name="Deger">Mutlak Degeri alınacak sayi.</param>
        ///<returns>Parametre olarak gelen sayinin mutlak degeri.</returns>
        public static int MutlakDeger(int Deger)
        {
            if(Deger < 0)
                return -Deger;
            else
                return Deger;
        }

        /// <remarks>
        /// Kare Alma Islemi
        ///</remarks>
        ///<summary>
        /// Parametre olarak gelen sayinin
        /// karesini almak için kullanilir.
        ///</summary>
        ///<param name="Deger">Karesi alınacak sayi.</param>
        ///<returns>Parametre olarak gelen sayinin karesi.</returns>
        public static double KareAl(double Deger)
        {
            return Deger * Deger;
        }
    }
}
```

Yukarıdaki örnek koddan görüldüğü üzere sınıf ve metod bildirimlerinden önce /// ile başlayan satırlarda XML formatında yorumlar yazılmıştır. VS.NET kod editörü /// karakterinden sonra <summary>, <param name="Deger"> ve <returns> etiketlerini otomatik oluşturdu. <remarks> etiketini ise kendimiz yazmalıyız. XML yorum etiketleri de intellisense özelliklerinden faydalanır. Otomatik tamamlama işlemi etiketler içinde geçerlidir.

Yukarıdaki örnekte <remarks> etiketi ile sınıf yada metod ile ilgili kısa bir açıklama yapılır. <summary> etiketi içinde daha ayrıntılı bilgi verilir. Gerekirse çeşitli teknik bilgiler de bu etiket içinde belirtilir.

Şimdi C# derleyicisinin XML formatındaki yorumları kaynak koddan ne şekilde ayırdığını görmek için projeyi derleyelim. Projeyi derlemeden önce eğer VS.NET kullanıyorsanız Proje özellikleri(Solution Explorer'a sağ tıklayarak görebilirsiniz) penceresinden "Configuration Properties/Build" sekmesinin altındaki "XML Documentation File" kutusuna oluşturulacak XML dosyasının ismini aşağıdaki gibi yazmalısınız. Tabi VS.NET editörünün intellisense özelliklerinden faydalanmak istiyorsak XML dosyasının ismini oluşturulacak DLL ismiyle aynı verilmelidir.



Eğer VS.NET gibi akıllı bir editörünüz yoksa .NET Framework ile birlikte ücretsiz olarak dağıtılan ve komut satırından çalıştırılabilen C# derleyicisini kullanarak ta XML yorum dosyalarını oluşturabilirsiniz. Bunun için yapmanız gereken csc derleyicisini komut satırından aşağıdaki gibi çalıştırmaktır.

```
csc /t:library /doc:XmlYorum.xml XmlYorum.cs
```

VS.NET yada C# komut satırı ile oluşturulan XML dosyasının yapısı aşağıdaki gibidir.

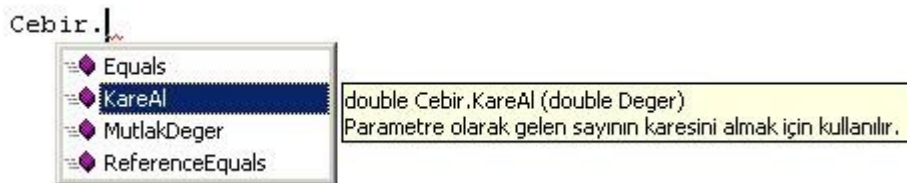
```

<?xml version="1.0" ?>
- <doc>
  - <assembly>
    <name>sefer</name>
  </assembly>
  - <members>
    + <member name="T:XMLYorum.Cebir">
      - <member name="M:XMLYorum.Cebir.MutlakDeger
        (System.Int32)">
        <remarks>Mutlak Deger Alma Islemi</remarks>
        <summary>Parametre olarak gelen sayinin Mutlak Degerini
          alir.</summary>
        <param name="Deger">Mutlak Degeri alınacak sayi.</param>
        <returns>Parametre olarak gelen sayinin mutlak
          degeri.</returns>
        </member>
      - <member name="M:XMLYorum.Cebir.KareAl(System.Double)">
        <remarks>Kare Alma Islemi</remarks>
        <summary>Parametre olarak gelen sayinin karesini almak
          için kullanilir.</summary>
        <param name="Deger">Karesi alınacak sayi.</param>
        <returns>Parametre olarak gelen sayinin karesi.</returns>
        </member>
      </members>
    </doc>

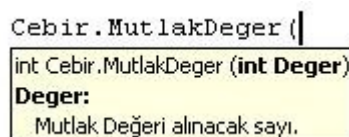
```

Şimdi birde oluşturduğumuz Cebir isimli bir sınıfa farklı bir projeden referans verip metodlarını kullanalım. Yeni bir Console uygulaması açın ve oluşturduğumuz Cebir sınıfına ait assembly dosyasına referans verin. Eğer komut satırı derleyicisi ile çalışıyorsanız " /r:Cebir.dll " parametresini ekleyin. Tabi bu durumda XML yorumlarının etkisini göremeyeceksiniz. Çünkü XML yorumlarını göstermek VS.NET teki akıllı editörün bir yeteneğidir. Notepad'in yada başka text editörlerinden bu tür imkanlar beklememek lazım!

Cebir sınıfının KareAl() metodunu kullanmak istediğimizde VS.NET'teki kod editörü bize KareAl() metoduna ilişkin XML formatındaki açıklamayı sarı kutucuk içinde aşağıdaki gibi gösterecektir. Böylece kullanacağımız metodun veya sınıfın bildirimine bakmamıza gerek kalmamıştır.



Aynı durum parametreler içinde geçerlidir. Örneğin KareAl() metodunun çağrım parantezini yazdığımız anda aktif parametre ile ilgili XML açıklaması aşağıdaki gibi gösterilir.



Aynı XML yorumları VS.NET ile entegre çalışan "Object Browser" arayüzü ile de aşağıdaki gibi gösterilmektedir.

<pre>public static System.Int32 MutlakDeger (System.Int32 Deger)</pre> <p>Member of XMLYorum.Cebir</p> <p>Summary: Parametre olarak gelen sayının Mutlak Değerini alır.</p> <p>Parameters: <i>Deger:</i> Mutlak Değeri alınacak sayı.</p> <p>Returns: Parametre olarak gelen sayının mutlak değeri.</p> <p>Remarks: Mutlak Değer Alma İşlemi</p>

Not : "Object Browser" penceresine erişmek için (Ctrl + Alt + J) tuş kombinasyonunu kullanabilirsiniz.

Diğer XML Yorum Etiketleri

XML yorum etiketleri yukarıda anlatılanlar ile sınırlı değildir. Aşağıda kullanılabilecek standart etiketler alfabetik sıraya göre toplu bir şekilde tablo halinde açıklamalarıyla birlikte verilmiştir.

<c>	<p>Açıklama içindeki bir bölümün "kod fontu" şeklinde olduğunu vurgulamak için kullanılır.</p> <p>Örnek :</p> <pre>/// <summary> /// Bu sınıf <c>Stream</c> sınıfından türemiştir. /// </summary> public class YeniSınıf { ... }</pre>
<code>	<p>XML açıklaması içinde uzun bir kod bloğu örneği verilecekse diğer yazılardan ayırmak için kod bloğu bu etiket arasında yazılır.</p> <p>Örnek :</p> <pre>/// <summary> /// Bu metod iki Kompleks türeden sayıyı toplar. Örneğin /// <code> /// Kompleks sayi1 = new Kompleks(5,6); /// Kompleks sayi2 = new Kompleks(0,1); /// /// Kompleks sayi3 = sayi1 + sayi2; /// </code> /// </summary> public Kompleks operator+(Kompleks sayi1, Kompleks sayi2) { ... }</pre>
<example>	<p>Bir metodun yada sınıfın ne şekilde kullanılacağını açıklayan blok bu</p>

	<p>etiket içinde yazılır. <code> etiketinin kullanımı ile hemen hemen eşdeğerdedir. <code> etiketini verilen örneğin aynısı <example> etiketi içinde geçerli olduğu ayrıca bir örnek vermeye gerek yoktur.</p>
<excepton>	<p>Bir metodun fırlatabileceği istisnai durumlarla ilgili bilgi vermek için kullanılır. <exception> etiketi "cref" niteliği ile birlikte kullanılır. "cref" niteliği ile fırlatılacak istisnai durum(exception) sınıfının türü belirtilir.</p> <p>Örnek :</p> <pre> /// <summary> /// Bu metod iki Kompleks türeden sayıyı toplar. Örneğin /// </summary> /// /// <exception cref="IndexOutOfRangeException"> /// </exception> /// <exception cref="OzelIstisnaiDurum"> /// </exception> public Kompleks operator+(Kompleks sayi1, Kompleks sayi2) { ... } </pre>
<list>	<p>HTML kodlarındaki etiketine benzer bir amacı vardır. Liste şeklinde bir yapı oluşması gerektiği bildirilir. <listheader> listedeki başlık bilgisini, <item> listedeki her elemanı, <term> her elemandaki terimi ve <description> bu eleman hakkındaki detaylı bilgiyi bildirir.</p> <p>Örnek :</p> <pre> /// <remarks>Bu bir liste örneğidir. /// /// <list type="number"> /// /// <listheader> /// <item> /// <term>term 1</term> /// <description>Açıklama 1</description> /// </item> /// </listheader> /// /// <item> /// <term>term 2</term> /// <description>Açıklama 2</description> /// </item> /// /// <item> /// <term>term 3</term> /// <description>Açıklama 3</description> /// </item> /// /// </list> /// </remarks> public class YeniSınıf </pre>

	<pre>{ ... }</pre> <p>Not : Liste tipi(<list type="number">) "number" olabileceği gibi "bullet" ve "table" da olabilir.</p>
<para>	<p><summary> gibi uzun açıklama bloklarında bir paragrafı belirtmek için kullanılır.</p> <p>Örnek :</p> <pre>/// <summary> /// Bu sınıf Stream sınıfından türemiştir. /// <para> /// Bu sınıf aynı zamanda IDisposable arayüzünü uygulamıştır. /// </para> /// </summary> public class YeniSınıf { ... }</pre>
<param>	<p>Bir metodun parametreleri ile ilgili bilgi vermek için kullanılır.</p> <p>Örnek :</p> <pre>///<param name="Sayi1">Toplanacak birinci sayı</param> ///<param name="Sayi2">Toplanacak ikinci sayı</param> public static double Topla(int Sayi1, int Sayi2) { return Sayi1 + Sayi2; }</pre>
<paramref>	<p><paramref> etiketleri içerisine alınan yerde aslında metodun ilgili parametresinin olduğu bildirilir. Böylece oluşacak XML yorum dosyasındaki bu etiketi farklı bir biçimde yorumlama şansına sahip oluruz.</p> <p>Örnek :</p> <pre>/// <summary> /// Bu sınıfın <paramref name="Sayi1"/> , ve /// <paramref name="Sayi2"/> ve biçiminde iki parametresi vardır. /// </summary> public static double Topla(int Sayi1, int Sayi2) { return Sayi1 + Sayi2; }</pre>
<permission>	<p>Üye elemanla ilgili güvenlik bilgisi vermektedir. Örneğin bir metoda yada sınıfa kimlerin erişeceği ve ne şekilde erişeceği bu etiket kullanılarak belirtilebilir.</p> <p>Örnek :</p> <pre>///<permission cref="Private">Herkes bu metoda erişebilir.</pre>

	<pre> ///</permission> public static double Topla(int Sayi1, int Sayi2) { return Sayi1 + Sayi2; } </pre>
<remarks>	<p>Bir tür hakkında kısa bir açıklama vermek için kullanılır.</p> <p>Örnek :</p> <pre> ///<remarks> /// Özel cebirsel işlemleri tanımlar. ///</remarks> public class Cebir { } </pre>
<returns>	<p>Bir metodun geri dönüş değeri ilgili bilgi vermek için kullanılır.</p> <p>Örnek :</p> <pre> /// <remarks> /// Kare Alma Islemi ///</remarks> ///<summary> /// Parametre olarak gelen sayinin /// karesini almak için kullanilir. ///</summary> ///<param name="Deger">Karesi alinacak sayi.</param> ///<returns>Parametre olarak gelen sayinin karesi.</returns> public static double KareAl(double Deger) { return Deger * Deger; } </pre>
<see>	<p>Yazı içinde bir bağlantının(link) olacağını belirtir. "cref" niteliği ile birlikte kullanılır. "cref" niteliği bağlantının olacağı üye elemanı simgeler.</p> <p>Örnek :</p> <pre> /// <summary> /// Bu metod iki Kompleks türeden sayıyı toplar. Örneğin /// </summary> /// /// <see cref="KompleksAlgoritmalar"/> public Kompleks operator+(Kompleks sayi1, Kompleks sayi2) { ... } </pre>
<seealso>	<p>Yazı içinde ilgili elemanla yakından ilişkili olan diğer elemanlara bağlantı vermek için kullanılır. Kullanımı <see> etiketi ile aynıdır.</p> <p>Örnek :</p>

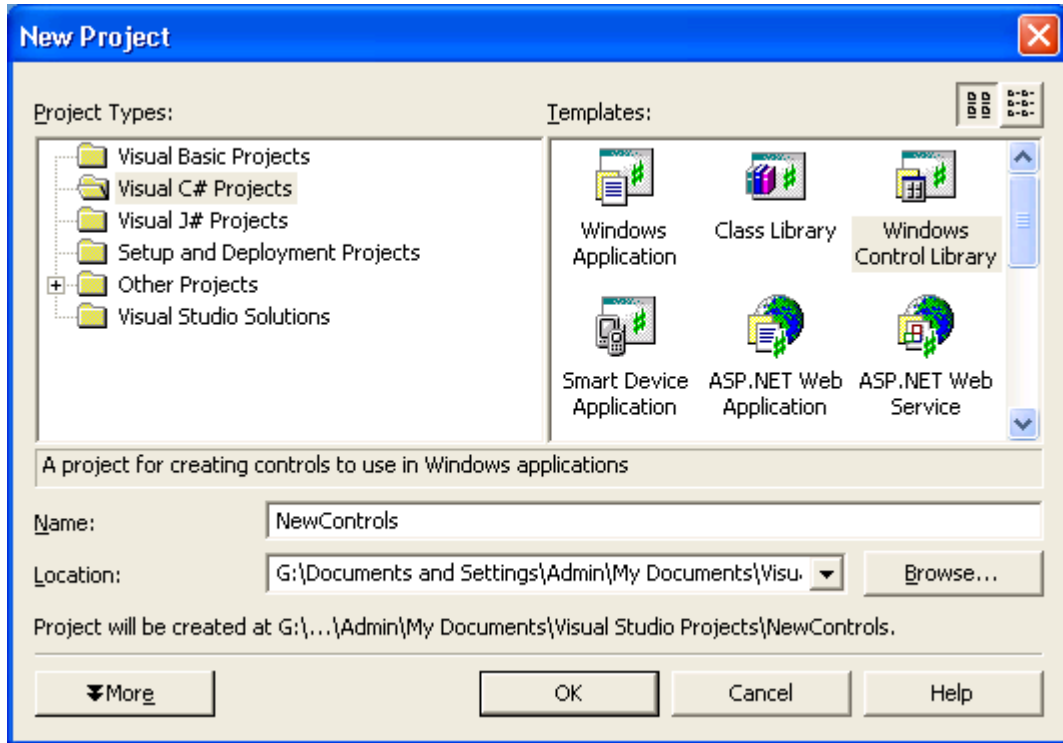
	<pre> /// <summary> /// Bu metod iki Kompleks türeden sayıyı toplar. Örneğin /// </summary> /// /// <seealso cref="operator-"/> /// <seealso cref="operator"/> /// <seealso cref="operator*"/> public Kompleks operator+(Kompleks sayi1, Kompleks sayi2) { ... } </pre>
<summary>	<p>Üye elemanla ilgili geniş açıklama yazmak için kullanılan bir etikettir.</p> <p>Örnek :</p> <pre> ///<summary> /// Cebir sinifi bazı özel matematiksel işlemleri /// yapmak için çeşitli statik metotlar sunar. ///</summary> public class Cebir { ... } </pre>
<value>	<p>Sınıfın bir üye elemanı olan özellikler (Property) hakkında bilgi vermek için kullanılır.</p> <p>Örnek :</p> <pre> ///<value> /// Kontrolün rengini belirtir. ///</value> public int Renk { get { return a;} set { a = value;} } </pre>

Kodlarınızı XML yorumları ile süslemeyi unutmayın !...

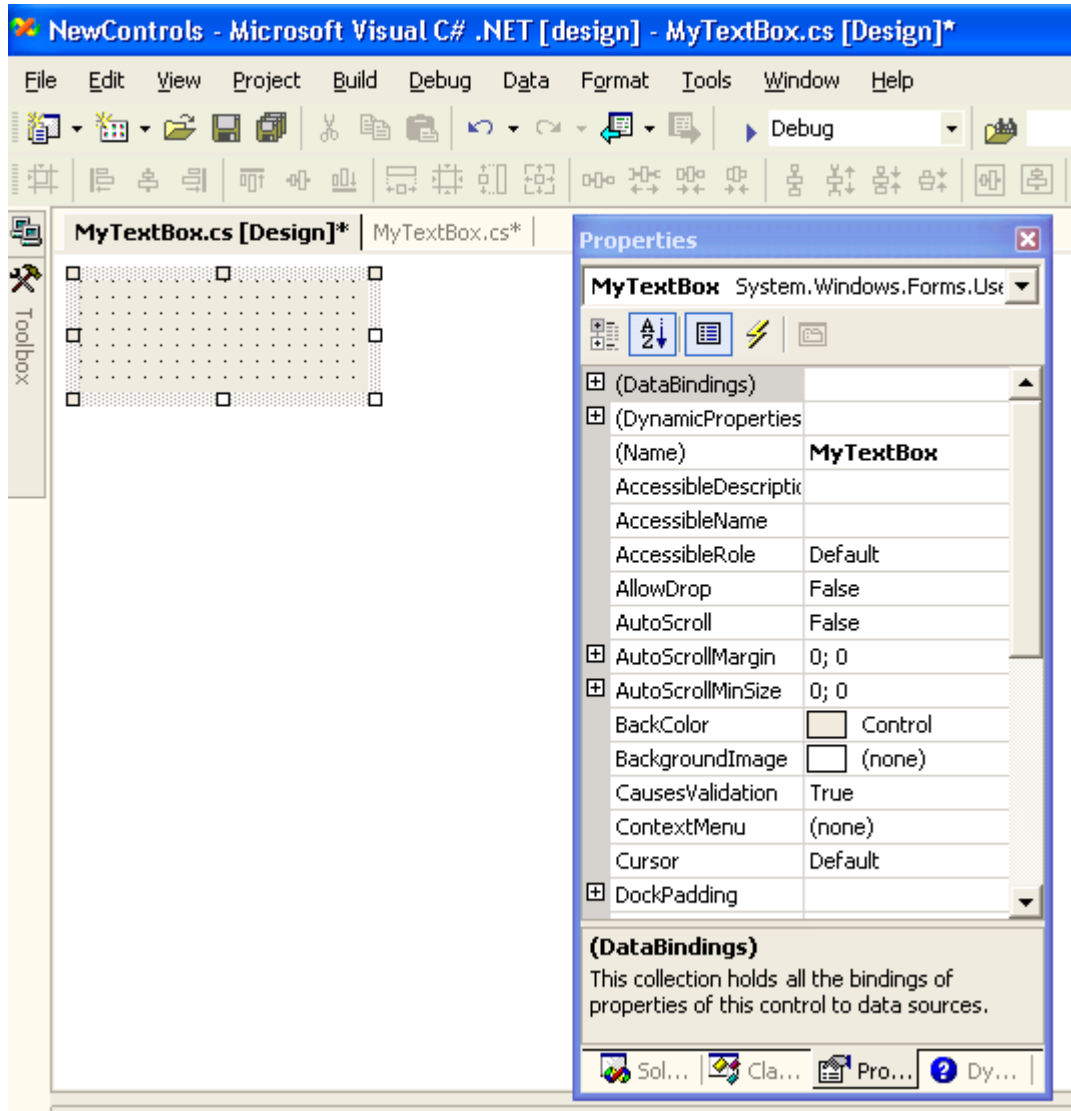
Visual C# ile Windows Kontrolü Hazırlama

Simdi sizlere Visual C# NET'te bir Windows Control nasıl yapılır ve bu Windows Control'ü programlarımızda nasıl kullanırsınız onu göstereceğim. Göstereceğim örneği çok basit seçtim, bunun nedeni de yaratıcılığı siz arkadaşlarıma bırakmayı uygun görmemdir. Şimdi örneğimizi adım adım inceleyelim.

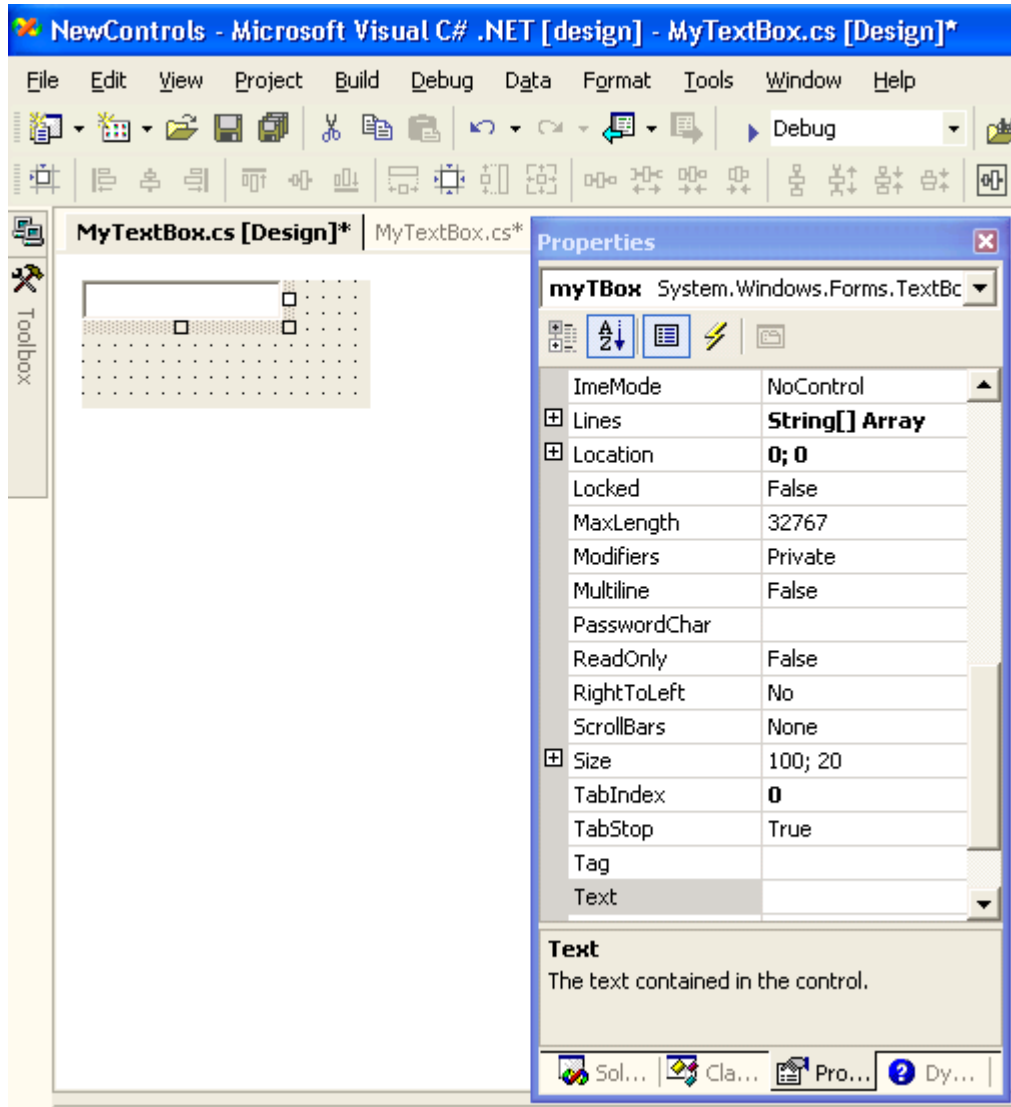
I. Visual Studio .NET'te yeni bir proje açalım ve Windows Control Library'yi seçelim ve adını değiştirelim(Ben burada NewControls adını verdim siz istediğiniz adı verebilirsiniz.)



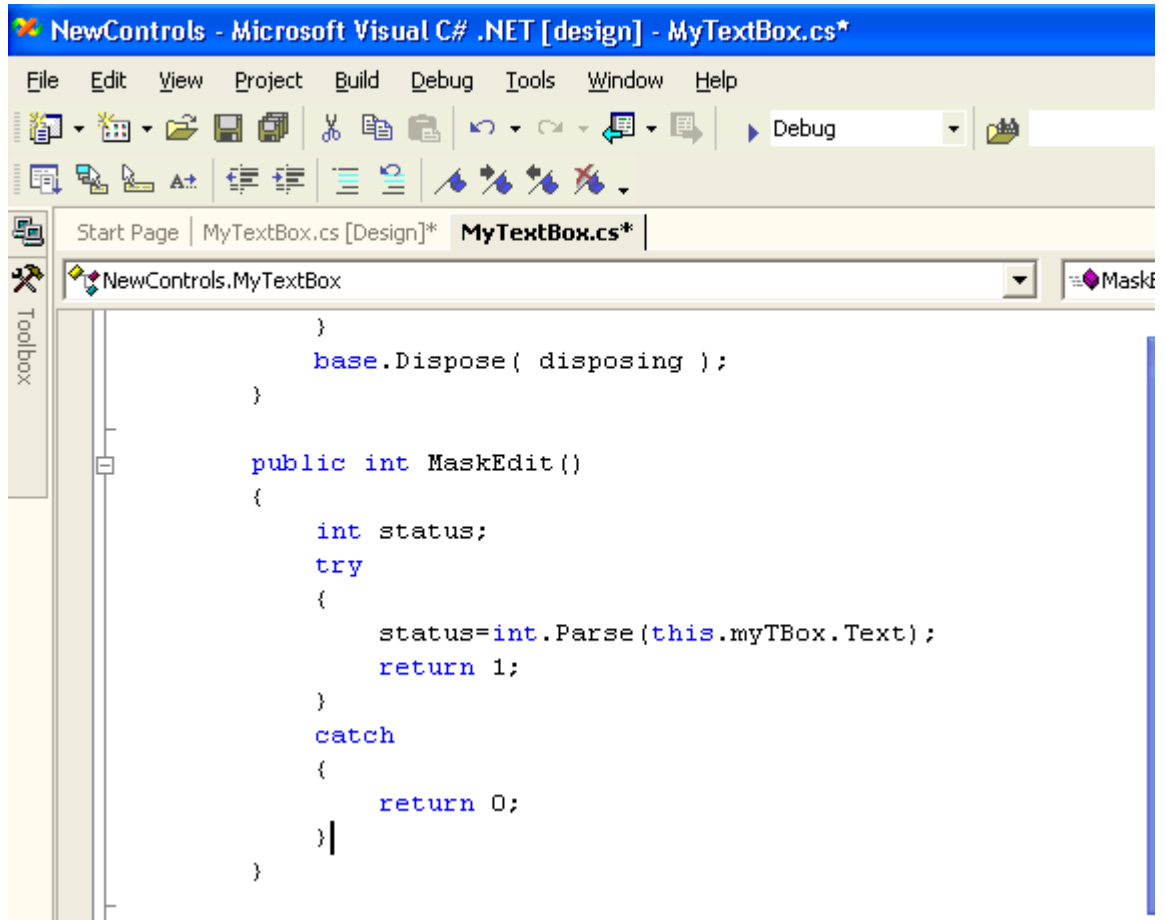
II. Daha sonra Anlamlı bir isim olması için UserControl1'in adını MyTextBox olarak değiştirelim.



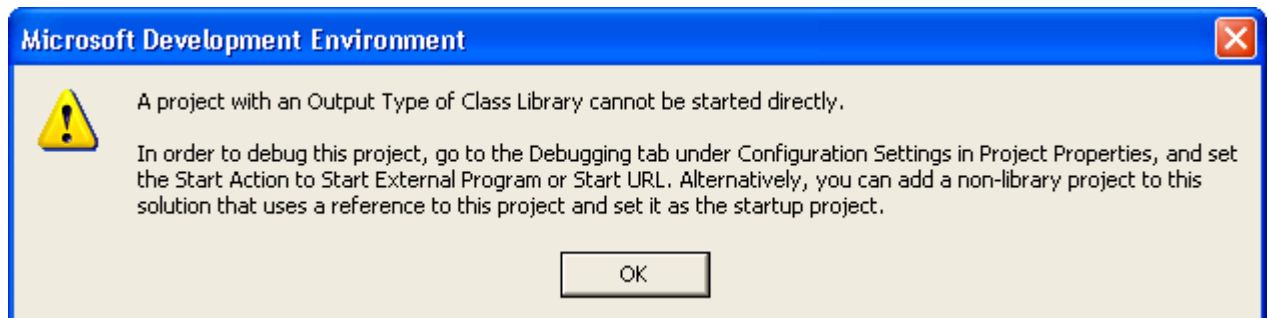
III. Bu değişiklikleri yaptıktan sonra MyTextBox'ın üzerine bir TextBox yerleştirelim ve onun adını da değiştirelim. Ben burada adını myTBox olarak değiştirdim.



IV. Evet şimdi MyTextBox'ın kodunu açalım ve bir TextBox'ın yapması gerektiğini düşündüğümüz özellikleri de eklemek için istediğimiz metodu buraya yazalım. Burada örnek olarak myTBox üzerindeki bilginin integer olup olmadığını kontrol eden bir metod yazalım ve metodun dönüş değeri eğer integer değilse 0 (sıfır) olsun. Eğer dönüş değeri 1 olursa integer olsun.

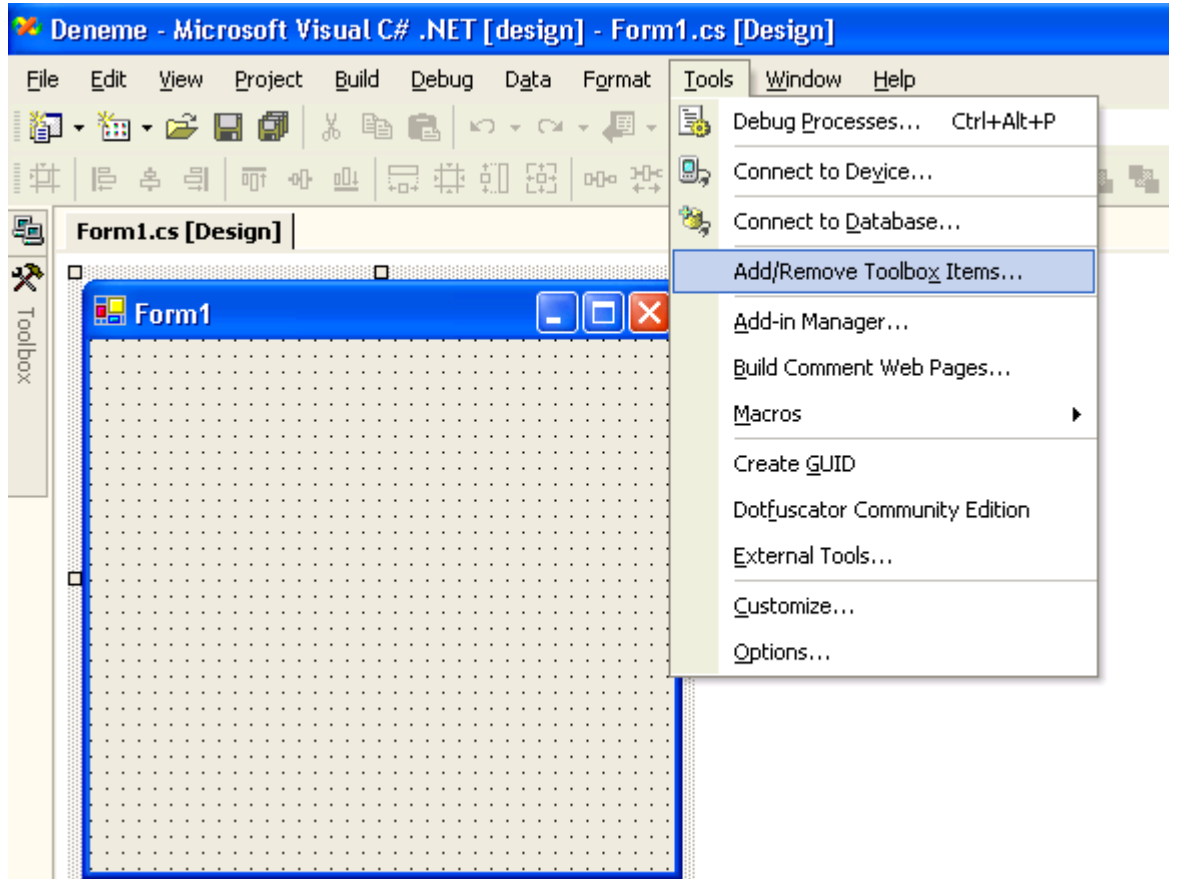


V. Kod yazımını tamamladıktan sonra derleyin, eğer derlemek yerine direk çalıştırırsanız(run) aşağıdaki uyarıyı alırsınız(Kısaca verdiği uyarı : "Bu Windows Control tek başına çalışamaz. Bunu baska projelerde kullanmalısınız.") Ama sorun değil çünkü yaptığımız Windows control'ü zaten diğer projelerde kullanmak üzere tasarladık.

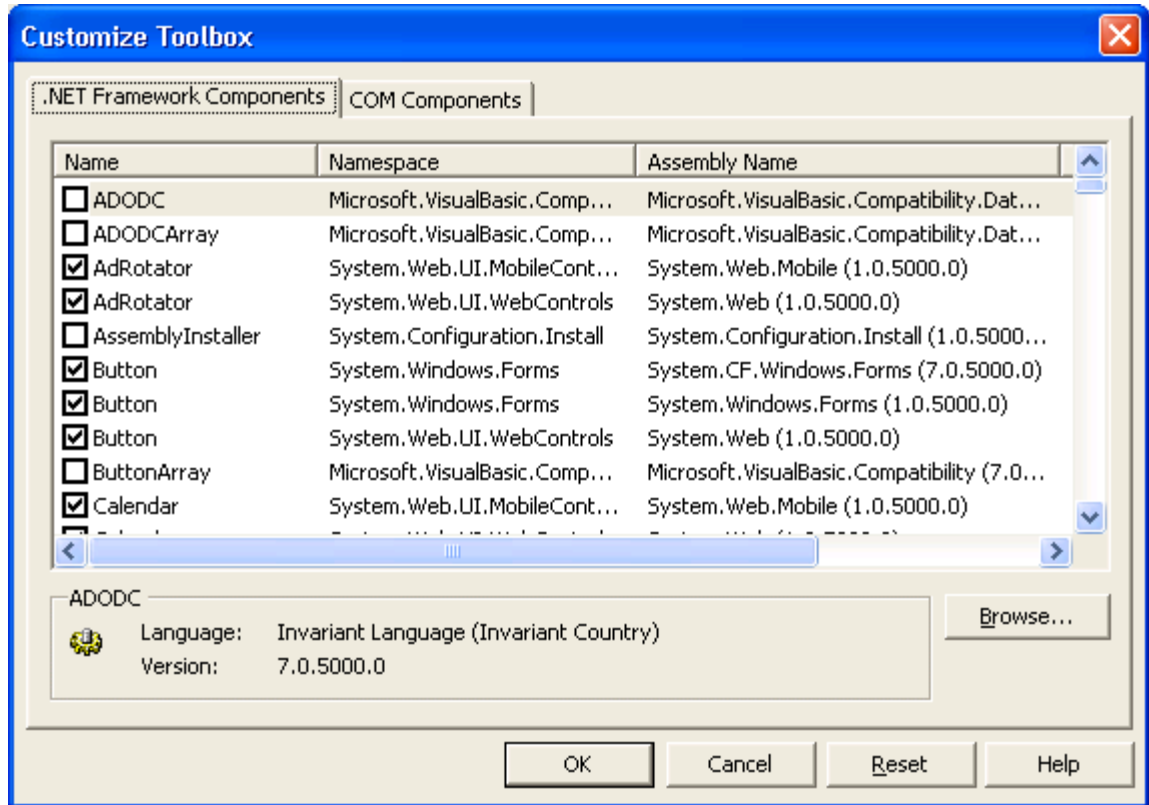


Şimdi sorabilirsiniz bu Windows Control'ü projelerimizde nasıl kullanacağız? Yine adım adım anlatalım.

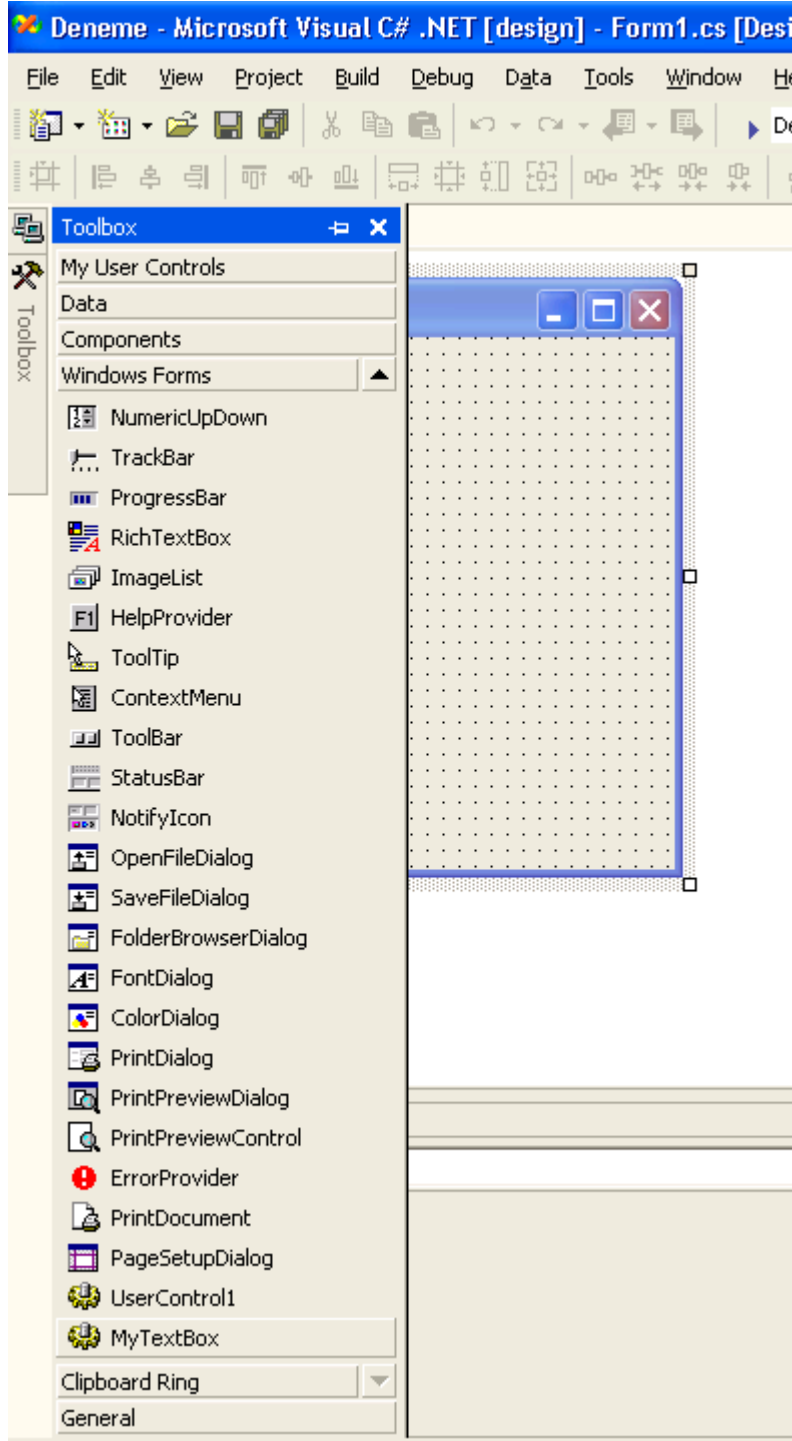
- I. Yeni bir proje açın veya önceden var olan bir projeyi açın. Ben burada Deneme adında yeni bir proje açtım.
- II. Daha sonra .Net'in Ana Proje Penceresindeki MenuBar'dan **Tools**'ı tıklayın açılan menuItem'lardan **Add/Remove ToolBoxItems**'ı tıklayın.



- III. Daha sonra açılan pencereden **.NET Framework Components**'in seçili olmasına dikkat edin. Eğer seçili değilse onu seçin. Ve Browse diyerek daha önce kaydetmiş olduğumuz NewControls projesinin içine girilelim oradan bin'e oradan da Debug'ın içine girelim. Daha sonra karşımıza çıkan **NewControls.dll** adlı dosyayı seçip OK tuşuna basalım. Daha sonra tekrar **NET Framework Components**'in seçili olduğu pencerede OK tuşuna basalım.



IV. Şimdi ToolBox'a bakalım. İşte karşımızda **MyTextBox**'imizin yazılı olduğu bir ToolBox'ımız oldu. Artık **MyTextBox**'imizi diğer **Tool**'ları kullandığımız gibi kullanabiliriz.



Artık bundan sonrasını siz uygulama geliştirici arkadaşlarıma bırakıyorum. Unutmadan; artık bu hazırlamış olduğumuz Tool'u diğer projelerimizde de kullanabiliriz. Anlaşılmayan herhangi birşey olursa aytacozy@msakademik.net adresine mail atabilirsiniz.

Not: Eğer kendi Tool'larınızı kullanarak bir proje yapıyorsanız ve yaptığınız projeyi başka makinelerde çalıştırmak isterseniz kullandığınız Tool'ları o makineye yukarıda bahsettiğim şekilde yüklemeniz gerekir.

C#'ta Inheritance(Miras) Kavramı

Bu yazıda inheritance'ın programlamada ne anlama geldiğinden bahsedeceğim. Inheritance aslında Object Oriented Programming'in (Nesne Yönelimli Programlama) üç prensibinden bir tanesidir. Diğer iki prensip ise encapsulation ve polymorphism'dir. Tabii ki diğer iki prensibe bu yazıda değinmeyeceğim. En sade şekliyle: inheritance sayesinde bir sınıfın metodlarını kullanan başka sınıflar türetilebilmesine yarar diyebiliriz. Ancak ayrıntılarına birazdan ineceğim. Eğer daha önce nesne tabanlı bir programlama dili kullandıysanız, (Java ve C++ gibi) C#'ta inheritance'a çok çabuk adapte olursunuz. Aslında şu ana kadar bahsettiklerim genel kültürden ibaretti ve eminim çoğunuz da bunları biliyordunuz. (Nesne Tabanlı Programlama geçmişi olmayanları da düşünerek böyle bir giriş yaptım.)

Evet şimdi ana kısma yani programın nasıl yazılacağına geliyoruz. Bunun için basit bir örnek vereceğim. Düşünün ki student adında bir sınıfımız(class) olsun. Ayrıca bir de teacher adında bir sınıfımız olsun. Bunların ortak özellikleri nedir? Tabii ki insan olmaları diyeceksiniz ve ana sınıfımıza yani person sınıfına ulaşmış olacaksınız. Şimdi basitçe özetlersek person sınıfından teacher ve student adında iki sınıf türetmiş olduk. Sırada bunun kodunu nasıl yazacağımız var. Alışkanlıklara devam edip adım adım kodu yazalım. (Bunu program yazarken de ilke edinirseniz faydalı olacağına inanıyorum. Önce ne yapacağınızı adım adım belirleyin sonra belirlediklerinizi adım adım uygulamaya geçirin.)

I. İlk önce person sınıfını yazalım.

```
using System;

using System.Windows.Forms;

namespace Miras
{
    public abstract class Person
    {
        //sınıfın sadece türetileceğini
        //belirtmek için sınıfı abstaract keyword'ünü kullanarak soyutladık
        //Ancak burada abstaract keyword'ünün kullanılmasındaki temel

        //faktör bu sınıfın abstract metod içermesidir.

    }

    //Türetilen sınıflarda kullanılmak üzere 3 tane değişken tanımladık.

    protected string Name;

    protected int Age;

    protected string Gender;

    //Türetilen sınıflarda metodun içi doldurulması için
```

```
//abstract olarak makeAction metodu tanımladık

public abstract void makeAction();

public Person()

{

}

}

}
```

II. Şimdi de Student sınıfını yazalım.

```
using System;

using System.Windows.Forms;

namespace Miras

{

    //Student class'ı Person class'ından miras aldığını belirtiyoruz.

    public class Student:Person

    {

        //Person sınıfında tanımlanan abstract metodu override ederek

        //metodun içini istediğimiz gibi doldurduk.

        public override void makeAction()

        {

            MessageBox.Show("Ben bir öğrenciyim");

        }

        public Student(string name,int age,string gender)

        {

        }

    }

}
```



```
        this.Name=name;

        this.Age = age;

        this.Gender=gender;

    }

}

}
```

III. Sıra Teacher sınıfını yazmaya geldi.

```
using System;

using System.Windows.Forms;

namespace Miras
{

    public class Teacher:Person //Teacher class'ı Person class'ından
    // miras alıyor

    {

        private string Unvan; //Teacher sınıfında kullanılmak üzere
        //Unvan adında bir değişken tanımladık.

        //Person sınıfında tanımlanan abstract metodu override ederek
        //metodun içini istediğimiz gibi doldurduk.

        public override void makeAction()

        {

            MessageBox.Show("Ben bir öğretmenim");

        }

    }

}
```


C#'a Kısa Bir Giriş

Ben C' dilini öğrenmeye 1 yıl önce Üniversite'de MS DOS ortamında yaptığımız basit matematiksel işlemlerle başladım. Gerçekte her programda alışla gelmemiş bir çok komut vardı. Günümüzde kullanılan C# diline göre çok gelişmemiş olan bu dile o kadar ısınmıştık ki artık uygulamalara yetişemez olduk. C çok eskiden çıkan bir dil fakat gelişimi ve insanların ona yetişmesi çok hızlı idi. Sırasıyla C, C++, C# ben buna 3D diyorum yani 3 dev demekle yetiniyorum. C dili bir çok dilin temeli veya üstünde bir dil. Hemen hemen her alanda kullanılmaktadır. Mesela Javascript, ActionScript... En önemli olan uygulama alanı ise Windows ve Linux gibi güçlü bir işletim sistemlerinin C de yazılmasıdır. C dilinin uygulama alanları sadece saydıklarımla sınırlı değildir ama bu alanların hepsini burda listelemem mümkün değildir.

.NET Framework, programcılara aşina olduğu kod dilini kullanma özgürlüğü tanıyarak bir devrim gerçekleştirdi. Ve, belli belirtilmelere sadık kalındığı sürece, farklı dillerle yazılmış uygulamaların birbiriyle etkileşebileceğinin de teminatını verdi.

Evet, .NET diller arası etkileşime olanak tanıyan, bir çok dile destek veren bir platform. Üçüncü parti derleyiciler yazılarak .NET için her an yeni bir dil daha yazılabilir. Ama herşeyden önce, .NET'in beraberinde sunduğu dillere bakmak gerekiyor. Bu diller temel olarak 4 tane: C++, Visual Basic .NET, C# ve J#.NET. Dikkat edilirse bu listede, "ben yeniyim" diye göz kırpan bir tanesi var : C#. Yazımızda, bu yeni dili tanımaya çalışacağız.

C# (si şarp) herkesin dile getirmiş olduğu gibi C++ ve Java 'nın birleşmesiyle oluşmuştur. Henüz nasıl bir birleşme şekli olduğuna dair tam bir fikrim yok ama C# mükemmel bir kütüphaneye sahip. Bu kütüphaneye ufak bir göz aşinalığımız olacak ama ilerideki yazılarımızda diğer dillerden büyük bir farkı olan esnek bir yapıya sahip olmasını inceleyeceğiz. Nedir bu esneklik? Yani Program yazarken "of be bu dilin de bu özelliği yokmuş" dediğimiz anlar olmuştur. C ile de şüphesiz nesnel programlama yapabiliriz. Fakat bunu yapabilmek oldukça zordur. C++ ise Nesne yönelimli programlamaya imkan vermektten öte zaten bu paradigmaya göre tasarlanmıştır ve yapısındaki araçlar sayesinde bunu kolaylaştırmıştır. İşte C- C++ arasındaki fark bu peki C#'ın özelliği nedir?

Nesne yönelimli programlamanın günümüzde ne kadar yaygın olduğunu programlama ile ilgilenen herkes bilmektedir. Nesne Yönelimli Programlama (NYP) yaklaşımında temel olan prensiplerden birisi bilgi gizleme (information hiding)'dir. Bu prensibi projelerimizde uygulamak için C#'in sunduğu en önemli araçlardan biri olan sınıf özellikleri (class properties) konusunu inceleyeceğiz.

Bildiğiniz gibi, C# dilinde tasarlanmış bir sınıfta iki temel unsur bulunur. Birincisi sınıfın özellikleri (fields), ikincisi ise sınıfın metodları (methods)'dir. Herhangi bir sınıfın özellikleri sınıfta tutulan ilişkili verilerlerdir. Diğer taraftan sınıfın bizim için değişik işleri yapmasını metodları vasıtasıyla sağlarız. Sınıf tasarımı çok önemli bir iş olup; deneyim, konsantrasyon ve dikkat ister. Sınıfımızın özelliklerini tutan veriler, program akışı sırasında sınıf dışında değiştirilebilir veya bu değerlere ulaşmak istenilebilir.

Elbetteki C# hakkında bilinmesi gerekenler bu kadarla sınırlı değildir. Bundan sonraki yazılarımda herşeyi daha ayrıntılarıyla aktarmaya çalışacağım.

.NET İin Tavsiye Edilen İsimlendirme Konvansiyonları – 1

Merhaba, bu makalemizde artık programcılık hayatımızın her yerinde, küüklü büyüklü her program için ihtiyaçtan çok bir zorunluluk haline gelen isimlendirme tekniklerine, tarihelerine değinecek,kendi isimlendirme stilimizi nasıl oluşturabiliriz ona bakacağız. 1. bölümün tamamını ,yani bu yazıyı tümüyle bu alacakken , 2. bölümünde özellikle Microsoft'un .Net için de tavsiye ettiėi konvansiyon olan Pascal & Capitalized Form (Pascal ve Büyük harfler notasyonu) ve uygulaması üzerinde duracağız.

Neden İsimlendirme Konvansiyonlarını Bilmeliyiz?

Tabii ki bu konvansiyonları kullanmak zorunda değiliz,kendi konvansiyonumuzu oluşturup kodlamaya da geçebileceėimiz gibi,konvansiyonsuz da kodlama yapabiliriz. Fakat ileri düzey programlamada isimlendirmenin birçok avantajı vardır.

İsimlendirme kavramı,programlama dünyasında kompleks kodların yazılmaya başlanmasıyla,özellikle de OOPL (Nesne Yönelimli Programlama Dilleri) nin gelişmesiyle büyük önem kazandı. Çünkü ortak olmayan ve anlamsız isimler,modüllere bölünmüş ve çözüm uzaylarına ayrılmış,spesifikasyonları hazırlanmış,yani en önemli bölümü halledilmiş bir programın sadece kodlama aşamasında çeşitli ciddi hatalara yol açılmasına sebep oluyordu. Bir kişiden fazlasının çalışmasını gerektiren projelerde insanlar birbirlerinin yazdıkları kodu anlamıyor, hatta bir kişinin kendi yazdığı programı bile daha sonra baktığında anlaması güçleşiyordu.

İsimlendirme konvansiyonlarını kullanmanın diėer bazı avantajları ise şunlardır :

- o Programa vereceėimiz isimler anlamlı olur.
- o Hepsi bir kurala baėlı olduėu için düzenli görünür.
- o İsim seçme işlemi artık mekanik olduėundan üzerinde düşünmeye gerek kalmaz, hızlı çalışsınız.
- o Takım çalışmalarında aynı dili konuşmanızı sağlar.
- o Kodlarınız anlaşılır olacaėından daha az yorum yazabilirsiniz.
- o Kodunuzu böceklerden(bugs) arındırırken faydası olur.
- o Kod standardize olduėu için daha sonra programınızın kodunu başka bir program yardımıyla iyileştirebilirsiniz.
- o Ortam hazırlayıcıları tarafından belirlenen notasyonu kullanmak,ortam tarafından otomatik olarak koda yerleştirilen kod paraları ile de uyumlu olacaėı için (ör: Form Designer'ın koda eklentileri) tam uyum sağlar.

İsimlendirme konvansiyon Çeşitleri

Bu sorunlara bir çözüm bulmak için notasyon adı verilen standartlar geliştirildi. Ortamların farklılıėından dolayı birçok standard ortaya çıktı. Bunlardan bazıları şunlardır :

Hungarian notation (Macar notasyonu):

Macar notasyonu diye bilinen bu notasyon diğer notasyonların atası olarak kabul edilmesi itibarıyla,günümüzde geçerliliği azalmıştır.

DOS'un ilk çıktığı zamanlarda Microsoft'un şef direktörü Charles SIMONYI tarafından geliştirilen bu tanımlayıcı isimlendirme notasyonunun temelinde,ismin önüne tipini yazarak aktif isimlendirmeyi sağlamaktır. Örnek verecek olursak, bir boolean flag için "bFlag" isimlendirmesi uygun bir isimlendirme şeklidir. String olarak strFirstName,integer olarak iNumberOfDays uygun isimlendirmelerdir.

Bu isimlendirmenin getirdiği faydalar artık modern programlama ortamlarının geliştirilmesiyle ortadan kalkmıştır. Çünkü,mesela .Net gibi bir ortamda bir değişkenin tipi zaten kodun her yerinde bellidir,bundan dolayı ismi uzatmaya gerek yoktur. Yani,bu notasyonun günümüzde kullanımı artık azalmıştır.

Ayrıca ortamların desteklediği tür sayısı günden güne arttığından bu tür bir isimlendirmeye gitmenin bayağı bir güç olacağı açıktır. Bu türün Extended Hungarian Notation,Modified Hungarian Notation ,Simple Hungarian Notation Hungarian Notation türleri bulunmaktadır.

MFC naming (Member-First Case) (İlk harfi tanımlayıcı notasyon):

Bu notasyonun temelinde tanımlayıcının tipinden çok türü önemlidir,yani int mi,short mu olmasından çok üye,sınıf,fonksiyon olmasına göre isimlendirilir. Event isimleri ise (On) ile başlar. Örnek olarak m_socket, i_counter,OnClose bu notasyona göre iyi isimlendirilmiş tanımlayıcılardandır.

Bu isimlendirme tekniğinin ise eskidiği Macar notasyonunda belirttiğimiz nedenlerden ötürü açıktır.

GNU Notation (GNU Derleyici Notasyonu)

Üstte belirttiğimiz diğer notasyonlardan farklı olarak bu notasyonda kelimeler arasında altçizgi (_) karakteri bulunma şartı getirilmiştir. Örneğin global_number_increase güzel bir isimlendirme iken icantreadthis iyi değildir. Ayrıca bazı GNU derleyicilerinde 8 ve/veya 14 harften fazlasına izin verilmediğinden zorunlu olarak bu derleyicilerin standartlarına harf sınırlaması da getirilmiştir.

Ayrıca yine bazı derleyicilerde (__) ile başlayan değerler ayrılmıştır. Bundan dolayı altçizgi ile başlayan isimlendirmeler iyi isimlendirme örneği değildirler.

Diğer bazı notasyonlar ise Sun – Java notation, SmallTalk – Roll Based Naming, Taligent Form dur.

Kendi İsimlendirme Konvansiyonumuzu Oluşturma

Yazının bu kısmına kadar , varolan isimlendirme çeşitlerini iyice anladığımızı umuyorum. Fakat hala benim kendi isimlendirme standardım olmalı diyorsak, dikkat etmemiz gereken bazı noktalar var.

Bir isimlendirme konvansiyonu oluştururken,dikkat etmemiz gerekenlerden ilki, isimlerin anlaşılabilir kadar uzun, fakat yazılabilecek kadar kısa olmasıdır. Bunları oluştururken konvansiyonları kullanmanın temel faydalarına zarar vermemeyi gözetmeliyiz.

İsimlendirme konvansiyonumuzu seçerken ortam,dil, ve kültür özelliklerine, isimlendirme konvansiyonu mantığının temelinde yatan estetik kaygıya ve algoritma oluşturma'nın temel şartlarına (verimlilik,isteneni verme vs vs ...) dikkat etmemiz gerekir.

Tüm İsimlendirme Konvansiyonlarında Bulunması Gereken Temel Özellikler

Bütün standartlarda ortak olması gereken noktaları ise şöyle sıralayabiliriz :

- Tanımlayıcının(değişkenin,sınıfın,metodun vb...) amacı doğrultusunda isimler verilmesi gerekir. Mesela okuldaki öğrenci sayısını tutan bir değişkene "tamsayı" şeklinde isim vermek yerine "ogrencisayisi" şeklinde isim vermek daha mantıklı olacaktır.
- Tanımlayıcının ismi büyük ve küçük harfleriyle okunabilir ve anlaşılır uzunlukta olmalıdır.
- Mümkün olduğunca kısaltmaları azaltmalıdır. Çünkü kısaltmalar çoğu zaman tehlikeli olabilmektedir. Örneğin "Ctr" "Control" olarak anlaşılabilceği gibi "counter" olarak da anlaşılabilir.
- Tanımlayıcıların isimleri,diğer tanımlayıcılar arasında ayırt edici özellik olarak kullanılmamalıdır. Örneğin "Counter" ve "counter" adında iki değişkenimiz olmamalıdır

Daha sonra iş kendi isimlendirme standardımızın şartlarını oluşturmaya bakıyor. Bunun için bu yazı genel bir fikir verebilir. İsimlendirme tekniklerinizi,hiçbir tanımlayıcı tipi açıkta kalmayacak şekilde tasarladıktan sonra projenin daha sonra da aynı mantıkla geliştirilmesi ve isimlendirme konvansiyonunuzun kalıcılığını koruyabilmesi için iyi bir şekilde dökümante etmelisiniz.

Dökümantasyonunuz isimlendirme konvansiyonunuzla ilgili herşeyi içermelidir(Tip isimleri,ön ekler,arka ekler,kısaltmalar,eklentiler,özel karakterler,vb...) .

Ve son söz olarak , unutmayalım ki , bir çok kod bir kez yazılır ama binlerce kez okunur. Bunu göz önüne alarak kodlamamızı daha profesyonel standartlara taşıyalım.

Yazının 2. bölümü Microsoft'un .NET ortamı için önerdiği formlar olan PASCAL & CAPITALIZED FORM , CAMEL FORM adlı notasyonları derinlemesine inceleyecek , ve herhangi bir isimlendirme sistemimize aykırı davranıldığını çok büyük kod parçalarında nasıl anlayacağımızı anlatacağım.

Görüşmek üzere,

C# 'ta Kapsülleme, Erişim Belirteçleri ve Polymorphism

Geçen yazımda inheritance' tan bahsederken encapsulation diye bir kavramdan bahsetmiştik, şimdi bu kavramı açıklayacağım. Daha önceki yazımda belirttiğim gibi 3 OOP prensibinden biri. Türkçe karşılığına gelirsek kapsülleme demek. Ancak bilgisayar terimi olarak biraz açarsak kapsülleme, yönettiği kod ve veriyi birbirine bağlayan ve bu ikisini dış kaynaklı karıştırma ve yanlış kullanımdan koruyan bir mekanizmadır. Bu sayede veriyi dış ortamdan koruyan bir ambalaj vazifesi gördüğünü de söyleyebiliriz. Şimdi kapsüllemeyi biliyoruz da C#'ta yada .Net Framework'ünde ne gibi farklılıklar var diyeceksiniz? Öncelikle .Net Framework'ünde gelen yeniliklerden bahsedelim.

I. Erişim belirteçlerinin(Access Modifiers) varlığı kapsüllemeyi çok daha rahat yapabilmemize olanak sağlar. Bu sayede bir metod veya bir değişken anahtar sözcükler(keywords) aracılığıyla sadece önceden belirlenen sınırlar dahilinde kullanılabilir. Yada bunlara belirlenen sınırlar içinden ulaşılabilir. Burada bahsedilen keyword'leri birazdan açıklayacağım. (Tabii ki C#'ta kullanılan keywordleri açıklayacağım. Ve kullanımlarını basitçe anlatacağım.)

II. Özellik(Property) Sahalarının kullanımı (Bunun yapımını ilerde C# kodu ile göstereceğim.) Bu sayede .Net Framework kapsüllemeyi destekler.

III. Soyut sınıf(abstract class) ve soyut metodların(abstract methods) kullanımı. Aslında kalıtım(inheritance) konusunu anlatırken taban sınıfımız(base class) soyut sınıf idi. Onun için bu kısmı sadece açıklayacağım. Örnek vermeyeceğim. Örneği görmek isteyenler miras(inheritance) konusunu anlattığım yazıdaki örneği incelerlerse istedikleri bilgiye ulaşabilirler.

Evet bu kadarlık giriş yeter. Şimdi yukarıda anlattığım 3 maddeyi enine boyuna tartışalım.

I. Erişim belirteçlerinin ne işe yaradıklarından yukarıda bahsettiğim için burada direkt erişim belirteçlerinin neler olduklarını yazalım ve erişim sınırlarını çizelim. Erişim sınırları geniş olandan dar olana doğru bir sıralama yaparsak.

- public: Bütün her yerden erişilmek istenen veriler public anahtar sözcüğü ile birlikte kullanılır. Sadece aynı proje içerisinde değil diğer projeler içerisinde de erişilebilir.
- internal: Aynı assembly içindeki tüm sınıflar erişebilir. Yani public anahtar sözcüğünün sadece aynı proje içinden erişilebileni. (VB .Net'te ise Friend anahtar sözcüğüne karşılık gelir.)
- protected: Protected anahtar sözcüğü ile birlikte kullanılan verilere ise sadece bir alt sınıfa kadar erişilebilir.
- private: Bu ise sadece tanımlandığı sınıfta geçerli olan veriler için kullanılır.

Ancak kontrollerde(controller) yaygın olan kullanım şekli kontrollerin dışarıdan erişilmesi istenen metodlarının(aynı anda diyelim ki 3 tane kontrol'ün belli metodlarının çalışması gerekli olabilir.) public anahtar sözcüğü kullanılan bir metod içinde tanımlanmasıdır. Şimdi bu durumun nasıl yapıldığını gösteren mini bir örnek kod yazalım. Kodda belirtilen kontrollerin daha önceden tanımlanmış olduğunu düşünelim.

```
public void ChangeColor(Color color)
{
    this.groupBoxLine.BackColor = color;
    this.groupBoxOutCity.BackColor = color;
    this.groupBoxExternalPriceDetails.BackColor = color;
```

```
this.groupBoxInternalPriceDetails.BackColor = color;
this.groupBoxUser.BackColor = color;
this.groupBox1.BackColor = color;
this.btnAddNewLine.BackColor = color;
this.btnAddNewUser.BackColor = color;
this.btnCentralReport.BackColor = color;
this.btnChangePassword.BackColor = color;
this.btnDeleteExternalLine.BackColor = color;
this.btnDeleteInternalLine.BackColor = color;
this.btnDeleteUser.BackColor = color;
this.btnExit.BackColor = color;
}
```

Yukarıdaki metod bir renk parametresi gönderilerek çağrıldığı zaman yukarıda yazan bütün (daha öncede private anahtar sözcüğü ile tanımlanmış olduklarını kabul etmiştik.) kontrollerin rengini gönderilen renge değiştirmeye yarıyor. Bu sayede yukarıdaki kontrollerin hepsinin BackColor dışındaki metodları dış dünyadan soyutlanmış oluyor.

Aslında yaptığımız metod public anahtar sözcüğü ile tanımlanmayıp internal anahtar sözcüğü ile de tanımlanabilir. Bu bizim metodun içindeki kontrollere ait BackColor metodlarının dış dünyadan ne kadar soyutlanmasını istediğimize bağlıdır.

II. Özellik sahaları sınıflara ait özel(private) değişkenlerin aynı metodlar gibi dış dünyaya açılmalarını sağlıyor. Sadece okuma amaçlı dışa açılım yapılabildiği gibi hem okuma-hem yazma amaçlı bir açılım da yapılabilir. Teorik olarak sadece yazma amaçlı da bir açılım olsa da ne kadar mantıklı olur bilmem!!!! Şimdi örneklerimize geçelim.

```
private int currentExNumber = -1;
private int loginStatus = 0;

public int CurrentExNumber //Sadece okuma amaçlı özellik
{
    get
    {
        return currentExNumber;
    }
}

public int LoginStatus //Hem okuma hem yazma amaçlı özellik
{
    get
    {
        return loginStatus;
    }
    set
    {
        loginStatus = value;
    }
}
```


Şimdi yukarıdaki özellikleri kullanırken nesneadi.LoginStatus ve nesneadi.CurrentExNumber şeklinde kullanabiliriz. Yalnız dikkat etmemiz gereken CurrentExNumber kullanılacağı zaman sadece eşit işaretinin(=) sol tarafında kullanılabilecek olması. Çünkü başta da belirttiğimiz gibi sadece okuma yapabildiğimiz için get metodu var. Zaten bir değer atamaya kalkarsak hata verecektir.(Derleme esnasında özelliğin sadece okuma amaçlı olduğuna dair debug penceresinden mesaj verir.) Bu sayede de değiştirilmesini istemediğimiz ama kullanmak zorunda olduğumuz verilerin dış ortamdan hem soyutlanmasına hem de bunların dış ortama belirli izinler dahilinde açılımına izin vermiş olduk.

III. Aslında soyut sınıf ve soyut metod'dan daha önce az da olsa miras konusunu anlatırken bahsetmiştim. Ancak şimdi biraz polymorphism'den bahsederek bu kavramları biraz daha açacağım. Polymorphism kapsülleme ve miras'dan ayrı düşünülemez. Polymorphism Yunancada "çok formluluk" anlamına gelmektedir. Polymorphism ile soyut sınıf arasındaki ilişkiden bahsetmeden önce soyut sınıf ve soyut metodlarla ilgili bir iki ayrıntı daha verelim. Soyut sınıf sadece taban sınıflarında kullanılır ve yeni nesne yaratılmasında kesinlikle kullanılamaz. (Yani new anahtar sözcüğü kullanılamaz.)

Soyut metodlara gelince bunların ise soyut sınıflarda kullanılacağından bahsetmiştik. Bunun bize sağladığı avantaj bu metodların türetilen sınıflarda nasıl gerçekleştirildiğini bilmek zorunda olmamamızdır. Aslında bunu söyleyerek polymorphism'in yararından bahsetmiş olduk. Yani polymorphism veri soyutlaması yaparak sadece ilgilenmemiz gereken olaylar ve verilerle ilgilenmemize olanak sağlıyor. Bu sayede taban sınıfından türetilen ve aynı metodu farklı gerçekleştirimlerle(implementation) kullanan birden fazla sınıfa sahip olabiliyoruz. En basit örnek üçgen bir çokgen, kare de bir çokgen ve her ikisinin de bir alanı mevcut. Hemen basitçe bir taslak çıkarırsak çokgen sınıfı soyut taban sınıfı ve alan adında soyut bir metoda sahip. Üçgen ve kare sınıfları ise türetilen sınıflar ve alan metodunu istedikleri biçimde gerçekleştiriyorlar. (Bu işlemlerin nasıl yapıldığı miras konusunu anlattığım yazıda mevcuttur.)

Bir de soyut özellikler(abstract property) var. Bunların kullanımı ise soyut metodlar ile özelliklerin birlikte kullanımı ile ortaya çıkmakta. Buna bir örnek kod verirsem anlaşılması daha kolay olacaktır. Ancak bunların kullanımına çok sık rastlamadığımı belirtmem gerekir.

Sanırım aşağıdaki örnek kod parçası soyut özelliklerin kullanımını daha da netleştirmiştir.

```
abstract class Taban // Soyut sınıf
{
    protected int CurrentExNumber = -1;

    public abstract int GetCurrentExNumber// Soyut özellik
    {
        get;
    }
}

class Turet: Taban //Turet adlı bir sınıf türetiliyor
{
    public override int GetCurrentExNumber// overriding property
    {
        get
        {
```

```
        return CurrentExNumber+1;
    }
}
```

Polymorphism'den bahsettik. Şimdi ise yalancı polymorphism'den bahsedelim. Aslında bir örnekle biraz daha açarsam daha net olur. Diyelim ki bir karşılaştırma metodunuz var ve hem integer hem de string veri tiplerini karşılaştırmak istiyorsunuz. Yalancı polymorphism sayesinde aynı isimde iki metod yazarak bu isteğinizi gerçekleştirebilirsiniz. Bunun için mini bir örnek kod yazalım isterseniz.

Aşağıda yazacağım metodların aynı sınıf içinde yazıldığını düşünelim. Şimdi bu metodları kullanırken metodların içinde yer aldığı sınıftan üretilen nesnenin `karsilastir()` yazdığımız anda kod tamamlayıcısı bize iki seçenek sunar biri bu metodun iki tane integer veri tipi ile çalıştığı, ikincisi ise bu metodun iki tane string veri tipi ile çalıştığıdır. Bu sayede bir arabirim ile birden fazla metod gerçekleştirilmiş olur.

Aslında bir metodun birden fazla gerçekleştirime sahip olması olayına overloading denir.

Dikkat edilmesi gereken nokta overloading ile overriding'in birbirine karıştırılmamasıdır. Unutmayın overloading'te bütün işlemler aynı sınıf içerisinde oluyor. Overriding'te ise tek bir sınıf yerine taban sınıfı ile bu sınıftan türetilen sınıflar için içine giriyor.

```
public void karsilastir(int sayi1, int sayi2)
{
    //Metodun iç implementasyonunu sizlere bıraktım.
}

public void karsilastir(string data1, string data2)
{
    Metodun iç implementasyonunu sizlere bıraktım.
}
```

Evet bu yazıda anlatacaklarım sona erdi. Kafanıza takılan kısımlar için mail adresimi tekrarlıyorum

Yeni Nesil İş Uygulamalarının Mimarı C# ve Diğer Diller

Şirket yöneticileri geliştirilecek proje için bir programlama dilini seçmek zorunda kaldığında genellikle şu soruyu sorar : Hangi programlama dili ile projeyi en etkin ve en hızlı şekilde müşteriye sunabileceğim hale getirebilirim? Bu sorunun çözümüne ulaşmak o kadar da kolay olmuyor maalesef. Çözüme zor ulaşmada programlama dillerinin fazla olmasının etkisi olmakla beraber her bir programlama dilinin sunduğu standart kütüphanenin farklı olmasının da etkisi oldukça fazladır. Özellikle günümüz iş uygulamaları birden fazla platformu destelemek zorunda kalmıştır. Buda seçilecek uygulama geliştirme ortamının önemini açıkça göstermektedir. Uygulamaların internet ortamına taşınması ile birlikte bir programlama dilinden beklenen özelliklerde doğal olarak değişmiştir. 1970' li yıllarda bir mikroişlemciyi programlamak ne denli önemli oluyorsa 2000'li yıllarda interneti programlamak o kadar önemli olmuştur.

İnternet'in iş dünyasına girişi ile birlikte geliştirilen uygulamalardan beklenenler de değişmiştir. Bu durum doğal olarak uygulama geliştiricileri doğrudan etkilemiştir. İnternet ortamında çalışan ve dağıtık yapıda çalışabilen çok yönlü bir uygulama geliştirmek eski yöntemlerle imkansız değildir ancak inanılmaz derecede zaman ve insan gücü gerektirmektedir. Bu zorlukları aşmak için gelişen teknolojiye ve isteklere paralel olarak programlama dilleri de doğal gelişim içine girmiştir. Bu yazıda son yıllarda iş ve kişisel uygulama geliştiricilerin adını sıkça duyduğu C# programlama dili ve diğer dillerle olan ilişkisi anlatılacaktır. C# programlama dilinin sunduğu imkanları anlatmaya başlamadan önce programlama dillerinin tarihsel gelişimine göz atmak gerekir. Zira C# dili yıllardır yoğun bir şekilde kullanılan C,C++ ve JAVA dillerinin temelleri üzerine kurulmuştur. Şunu da hemen belirtelim ki, son geliştirilen ilk geliştirilenden çoğu zaman daha iyi olacaktır. Bu yüzden eski ile yeniye karşılaştırırken ticari amaçları bir kenara bırakıp objektif bir gözle değerlendirmek gerekir.

C#'ı konuşmadan önce C, C++ ve C# ile yakından ilişkili olan JAVA'dan bahsetmek gerekir.

C dili ve Yapısal Programlama

Düşündüklerimizi makinelerle yaptırma isteğimizin bir sonucu olarak programlama dilleri doğmuştur. Makineleri anlamak insanoğlu için o kadar da kolay olmamıştır. Zira makinelerin(bilgisayarların) anladığı dilden konuşmak insanlar için gerçekten zor bir iştir. Gün geçtikçe makineleri anlamak ve onları programlamak için yeni arayışlar içine girildi. Somutlaştırılmış makine komutları sayesinde bilgisayarları daha etkili bir şekilde yönetmek mümkün hale gelmiştir. Zaman ilerledikçe bilgisayarlar sadece belirli bilimsel hesaplamaları yapmak için kullanılan araç olmaktan çıkıp insanların yaşamlarında rutin işleri yapabilecek araç haline geldi. Bilgisayarların insanların ihtiyaçlarına hızlı bir şekilde cevap verebilmesi için onları hızlı bir şekilde programlamak gerekiyordu. Klasik yöntemlerle(makine komutlarıyla) hızlı çözümler üretilemez hale gelince daha yüksek seviyeli programlama dillerine ihtiyaç duyuldu. 1980'li yıllarda en çok kullanılan programlama dili olan "C" bu anlamda atılmış büyük bir adımdır. Yapısal programlama

modeli her ne kadar C dilinden önce de yapılıyor idiye de asıl büyük gelişmeler C dili ile birlikte olmuştur. C gibi makine diline göre yüksek seviyeli programlama dilleri ile büyük projeler yapılabilirdi. Artık uygulamalar sadece bilimsel çalışma aracı olmaktan çıkıp iş dünyasında kullanılabilen uygulamalar haline geldi. Bütün bu iyi gelişmelerin yanında zaman su gibi akıp gidiyordu, buna paralel olarak projeler büyüyor ve teknoloji artan ivmeyle geliyordu. Yavaş yavaş anlaşıldı ki C dili çok büyük projelerde yetersiz kalıyordu. Yeni bir programlama modeline ihtiyaç duyuldu ve C++ dilinin temelleri atıldı.

C++ ve Nesne Yönelimli Programlama

Yapısal programlama modeliyle çok büyük projeleri kontrol altına almak neredeyse imkansızdır. Bu sorunun üstesinden gelmek için yeni bir model gerekiyordu. Nihayet Bjarne Stroustrup tarafından C dili baz alınarak yeni bir programlama dili geliştirildi. Bu dilin adı : C++'tır. C++, C'nin üzerine inşaa edildiği için ilk başlarda "C with Classes"(Sınıflı C) olarak adlandırıldı. Peki bu dil C'den farklı olarak programcılara ne sunuyordu? C++ dilinin sunduğu en büyük yenilik nesne yönelimli programlamayı destekliyor olmasıdır. Nesne yönelimli programlama tekniği günümüzde de yaygın bir şekilde kullanılan bir tekniktir. Bu teknik gerçek hayatı modellemede büyük bir başarı sağlamaktadır. Söz gelimi bir projeyi parçalara ayırıp bu parçalar arasında programlama yolu ile bağlantılar kurmak çok basit hale gelmiştir. Nesne yönelimli programlama tekniği proje geliştirme aşamasında burada sayamayacağımız birçok kolaylık sağlamaktadır.

C++ dilinin diğer bir özelliğide C programcılarına hitap etmesiydi. C dilindeki temel kurallar aynen C++ dilinde de mevcuttur. Bu yüzden C++ dilini ve nesne yönelimli programlama tekniğine geçiş yapmak için C dilini iyi bilmek gerekir. Daha doğrusu C++ dilini sadece nesne yönelimli programlamayı destekliyor şeklinde düşünmemek gerekir. Günümüzde birçok alt seviye işlemlerde(haberleşme, işletim sistemi, aygıt sürücüler) C++ dilinin yoğun bir şekilde kullanılması bunun bir kanıtıdır.

İnternetin Gelişimi ve JAVA Dili

İnterneti'nin gelişimi bilgisayar dünyasındaki en önemli ilerlemelerden biridir. Programlama dünyasında JAVA dilinin ortaya çıkması en az internetin ilerlemesi kadar önemlidir. Çünkü C ve C++ dilleri ile yalnızca belirli sistemlere yönelik uygulamalar geliştirilebiliyordu. Oysa internet sayesinde birçok farklı sistem birbirine bağlanır hale gelmiştir. Artık sistemlerden bağımsız uygulama geliştirmek gerekiyordu. Daha doğrusu interneti hedef alacak uygulama geliştirmek gerekiyordu. Programcılar gelişen internet ortamına yabancı kalamazdı. Bu amaç doğrultusunda Sun Microsystems isimli firma önceleri OAK olarak anılan JAVA isimli programlama dilini ortaya çıkardı. JAVA, dil olarak C++ dilinin devamı gibi düşünülebilir. Ama amaç tamamen farklıdır. Zira Sun firması ortaya JAVA dili ile birlikte yeni bir uygulama geliştirme modelide sunmaktaydı. Bu programlama modelinde en büyük hedef sistemler arası taşınabilir kod yazmaktır. Yani bir uygulamayı hem Microsoft platformunda hemde Unix ve Linux platformlarında çalıştırabilmek hedeflenmiştir. Böylece geliştirilen uygulamalar işletim sistemi ve işlemciden bağımsız hale gelecektir.

Peki sistemler arası bu yüksek taşınabilirlik nasıl olmaktadır? Cevabı basit : Ara Dil. Evet, JAVA dilinde yazılmış kodlar derlendiğinde kodlar makine komutların çevrilmeden "ara kod" denilen "bytecode" a çevrilmektedir. Bytecode'a çevrilen program çalıştırıldığında Java Sanal Makinesi devreye girer ve uygulamanın çalıştırıldığı sisteme özgün makine kodunu üretir. Bu durumda Sun firmasının bir çok sistemde çalışabilecek Java Sanal Makinesi üretmesi gerekiyordu. Nitekim zamanla günümüzde yaygın kullanılan bütün sistemlerde sorunsuz çalışabilecek Java Sanal Makineleri geliştirildi. Hatta şu an için bazı cep telefonları ve çeşitli sim kartlarında bile JAVA programlarını çalıştıracak Java Sanal Makineleri mevcuttur.

JAVA ile C++ dili her ne kadar birbirine çok benzer olsada aynı kategoride değildir. Elmayla armutu karıştırmamak gerekir. Eğer "JAVA mı C++ mı" diye bir soru sorulursa cevap "her ikisi de" olacaktır. Çünkü ikisininde kullanım amacı farklıdır. Bir firma bir proje için hiçbir zaman bu iki dilden birisini seçmek durumunda kalmayacaktır. JAVA ile aynı kefiye koyabileceğimiz dil birazdan anlatacağım C# dilidir.

C# Dili ve .NET Platformu

JAVA'nın platform bağımsız kod üretmedeki başarısı su götürmez bir gerçektir. Bir çok kurumsal dev projede JAVA dilinin ve J2EE platformunun olanaklarından faydalanılması bunun en önemli göstergesidir. Günümüzde büyük projelerde birden fazla programlama dili kullanılabilmektedir. Ancak JAVA'nın diller arası uyumlu çalışmaya destek verememesi JAVA'nın bir eksikliği olarak görülmüştür. Diller arası uyumlu çalışma alanında en büyük başarıyı Microsoft firması sağlamıştır. Son dönemlerde sıklıkla kullanılan COM teknolojisi bu uyumluluğa bir örnektir. COM sayesinde farklı dillerde yazılan yazılım parçacıkları diğer bir uygulamada kullanılabilmektedir.

JAVA'nın programlamadaki büyük bir boşluğu doldurması onun en büyük rakibi olan Microsoft firmasının gözünden kaçmadı. En sonunda Microsoft'un bir ürünü olan Visual Studio yazılım geliştirme aracına JAVA yı da ekleme kararı aldı. Visual J++ adı altında Windows platformuna entegre edilen JAVA dili bu platformda pek başarılı olamadı. Bu oluşumun başarılı olmadığını gören Microsoft yeni arayışlar içine girdi. Microsoft başkasının malını kendi ürününe entegre etmek yerine kendi ürününe geliştirmeye karar verdi ve .NET yazılım geliştirme platformunu ortaya çıkardı. .NET temel felsefe olarak J2EE platformuna benzemektedir ancak .NET'in derinliklerine daldıkça çok yeni kavramlarla karşılaşırız. Bu yeniliklerden en önemlisi "diller arası uyumluluk" tur. J2EE platformunda sadece JAVA dili kullanılıyorken .NET platformunda birçok dil kullanılabilmektedir. Bu dillerin sayısı oldukça fazladır. Üstelik Microsoft tarafından .NET platformu için sıfırdan yeni bir dil tasarlanmıştır. Yapı olarak C++ ve JAVA dillerine benzerliği ile bilinen bu dil Anders Hejlsberg tarafından geliştirilen C# (C Sharp)'tan başka bir şey değildir..

JAVA, C++ diline nasıl benziyorsa C# dilide C++ ve JAVA'ya benzemektedir. Programlama modeli yine her üç ortamda da nesne yönelimlidir. Değişen şey bu modelin uygulanış şeklidir. C++'ta kaynak kod derleyici tarafından makine koduna, JAVA'da bytecode'a C#'ta ise IL(Intermediate Language-Ara Dil)'a çevrilmektedir. Burda

vurgulanması gereken en önemli nokta JAVA'da bytecode JAVA sanal makinesi tarafından yorumlanarak çalıştırılırken, .NET'te IL kodları derlenerek çalıştırılmaktadır. Hemen şunu da belirtelim ki, derleme işlemi yorumlama işleminden performans açısından daha öndedir.

C# dil olarak C++ ve JAVA'ya çok benzemektedir. Bu yüzden C# dilini konuşurken .NET platformunu göz önünde bulundurmalıyız. Dilleri sadece birer araç olarak görmemizde fayda var. İsterseniz lafı daha fazla uzatmadan JAVA/J2EE ve C#/.NET'i karşılaştırıp benzerliklerini ve farklılıklarını ortaya koyalım ardından C#'ı diğer .NET dillerinden ayıran özellikleri inceleyip "neden C#" sorusuna cevap arayalım.

C# ile .NET mi JAVA ile J2EE mi?

Saf C# ve JAVA dilleri düşünüldüğünde birkaç nokta dışında bu iki dil birbirine benzemektedir. Bu yüzden karşılaştırma yaparken bu dillerin kullanıldıkları platformlarda göz önünde bulundurmak gerekir. İsterseniz madde madde her bir özelliği iki platform için değerlendirelim.

1-) Mimari : .NET ve J2EE çalışma biçimi olarak birbirine çok benzer. Her iki platformda da uygulama kaynak kodu ara bir koda dönüştürülür. Aradaki en büyük fark bu ara kodun işletilmesi sırasında görülür. .NET'te ara kod çalışma zamanında derlendikten sonra çalıştırılırken JAVA'da yorumlanarak çalıştırılır.

2-) Çalışma Zamanı(Runtime) Mimarisi : J2EE platformundaki Java Sanal Makinesi ile .NET platformundaki CLR(Common Language Runtime) birimi eşdeğerdedir. JVM, bytecode'un işletilmesinden sorumlu iken CLR, IL kodlarının işletilmesinden sorumludur.

3-) Sistemler Arası Taşınabilirlik : Teorik olarak C# ve JAVA ile yazılmış uygulamalar sistemden bağımsızdırlar. Günümüzde C# ile .NET ortamında geliştirilen uygulamaların bir çok mobil cihazda ve Windows sistemlerinde kullanıldığını düşünürsek bu teorinin yavaş yavaş gerçeğe dönüştüğü görülebilir. Yakın bir gelecekte .NET altyapısının Linux versiyonunun da çıkacağı bilinmektedir. JAVA ise bu konuda kendisini çoktan kanıtlamış durumdadır.

4-) Diller Arası Uyumluluk : J2EE platformunda sadece JAVA dili kullanılırken .NET ortamında C#,C++,VB.NET ve hatta JAVA dili bile kullanılabilir. Üstelik farklı dillerde yazılmış parçacıklar diğer bir dilde sorunsuzca kullanılabilir. Bu sayede bütün programcılar .NET platformunda rahat programlama yapabilmesi sağlanmıştır. .NET uyumlu herhangi bir dilde geliştirilen bütün uygulamalar aynı ara koda dönüştürüldüğü için .NET dilleri arasında büyük performans farklılıkları meydana gelmez.

5-) Web Servisi Kullanımı : Web Servisleri dağıtık yapıda geliştirilen uygulamaların temel parçası olmuştur. Özellikle iletişimin XML tabanlı olması web servislerinin önemini göstermektedir. Her iki dil ile web servislerine erişmek mümkün olsada C# ile bir web servisini kullanmak oldukça kolaydır. C# ve .NET'in web servislerine kolay erişmesi bir avantaj olarak görülebilir.

6-) Bellek Yönetimi : C#'ta aynen JAVA'da olduğu gibi kullanılan nesneleri toplama programcının görevi değildir. Kullanılmayan gereksiz nesneler gereksiz nesne toplayıcısı tarafından zamanı geldiğinde bellekten silinirler. Buna rağmen C# programcıları isterse belleği kendileri de yönetebilir. Yani C# dilinde bellek adreslerini tutan göstericiler(pointer) hala kullanılabilir. JAVA dilinde bu imkan yoktur. C#'ı JAVA dan ayıran en büyük fark budur. Zira gösterici kullanımı sayesinde geriye dönük uyumlulukta sağlanabilmektedir. Örneğin parametre olarak bir gösterici alan sistem fonksiyonunu C#'ta kullanmak mümkündür.

7-) Veri Tipleri : C# dilinin temel felsefesi herşeyin bir nesne olmasıdır. Temel veri türleride dahil olmak üzere herşey birer nesne olarak tanımlanır. C# ve JAVA sağladığı temel veri türleri bakımından birbirlerine çok yakındır.

8-) Tekrar Kullanılabilirlik : Nesne yönelimli programlama modelinin en önemli özelliği geliştirilen sınıfların paketlenerek sonradan tekrar farklı uygulamalarda kullanılabilmesidir. C#' ta sınıflar isim alanları(namespace) içerisinde paketlenerek diğer uygulamalar içinde kullanılabilir. Java'da ise sınıflar "package" dediğimiz bir kavramla paketlenir. Sonuç olarak her iki dilde eşit oranda bu özelliği desteklemektedir. Ancak C#'ta sınıfların organizasyonu daha estetik bir şekilde düzenlenmektedir.

9-) Kontrol Mekanizmaları : Kodların içinde en çok görülen bloklar olan for,while ve if gibi yapılar her iki dilde de vardır. C#'ta JAVA dilinde olmayan ayrıca foreach döngüsü bulunmaktadır. foreach döngüsü ile koleksiyon tabanlı nesnelerin elemanları arasında tek yönde rahatça dolaşılabilir.

10-) Türetme ve Çok Biçimlilik : Nesne yönelimli programlama modelinin C++ dilinden beri kullanılan mekanizmaları olan türetme ve çok biçimlilik her iki dilde de mevcuttur. C++'tan farklı olarak C# ve Java'da sadece tekli türetme mevcuttur.

11-) İstisnai Durumları Yönetme : Uygulamaların en büyük düşmanı olan istisnai durumların(exceptions) her iki dilde de ele alınış biçimi hemen hemen aynıdır.

12-) Sınıf Kütüphanesi : Veritabanı ve dosya işlemleri gibi burada sayamayacağımız bir çok temel işi yapan sınıflar .NET ve J2EE platformunda mevcuttur. Gerek bu sınıfların organizasyonu gerekse de sınıfların kullanılış biçimi bakımından .NET platformunun daha avantajlı olduğunu söyleyebiliriz.

Bütün bu maddeler bir bütün olarak ele alındığında C#'ın JAVA'dan bir kademe önde olduğu görülmektedir. Bu durum elbette proje yöneticilerinin seçimlerini etkilemektedir. Microsoft faktöründe göz önünde bulundurursak C# ve .NET'in gelecekte çok iş yapacağını söylemek için müneccim olmaya gerek yok. Bu arada JAVA'nın halen yaygın bir şekilde kullanıldığını da gözardı etmemeliyiz. Bu durum C# ve JAVA'nın seçiminde sadece teknik özelliklerin değil aynı zamanda Windows ve Linux'te olduğu gibi sosyal etkenlerinde rolü bulunduğunu gösteriyor.

Buraya kadar söylediklerimden belki şöyle bir soru işareti doğmuş olabilir : "C# mı JAVA mı" sorusunu "C# mı C++ mı" şeklinde sorsak neler değişir? Cevap : Çok şey değişir. Evet C#'ın JAVA ile olan ilişkisi C++ ile olan ilişkisinden tamamen farklıdır. C# ile JAVA'yı ancak saf dil olarak karşılaştırabiliriz. Yani dilin sentaksından bahsediyorum. Bu iki dilin kullanıldığı ortam farklıdır. Birinde bir sisteme özgün makine kodu üretilirken diğerinde sistemden bağımsız ara bir kod oluşturulmaktadır. Bu durumda C++ ve C#'ı bir bütün olarak karşılaştırmayı kişisel olarak doğru bulmuyorum. Çünkü ikisi farklı

kategorilerde yarışıyor. Eğer bir gün .NET'in ürettiği ara koddaki komutlar ile çalışan mikroişlemci geliştirilirse o zaman belki C# ile C++'ı karşılaştırabiliriz. Peki C# mı C++? Cevap : Her ikiside. Eğer şirketiniz Intel işlemciler için bir işletim sistemi geliştiriyorsa elbette C++ ve C dilleri seçilmelidir. Şirketiniz dağıtık yapıda çok geniş bir çalışma ağı olan bir uygulama geliştiriyorsa o zaman C# ve .NET'i seçmeniz daha doğru olacaktır. Bu seçim bir projede hangi dilin kullanılacağını değerlendirmek içindi. İşe bir de programcılar açısından bakalım. Bir programcının hem C++ hem C# hemde JAVA bilmesine gerek var mı? Bence gerek var yada yok. Kesin bir cevabı verilemez bu sorunun. Daha doğrusu bir programcı ihtiyaç dahilinde herhangi bir programlama dilini kullanabilmelidir. Ancak şunu da unutmayalım ki iyi bir programcı çok sayıda programlama dili bilen demek değildir. İyi bir programcı .NET platformunda olduğu gibi programlama dilinden bağımsız kod üretebilmelidir.

Diğer .NET Dilleri ve C#

Daha öncede dediğim gibi .NET platformunda bir çok programlama dilini kullanabiliriz. Bu dillerin en önemlileri C#, VB.NET, C++.NET ve J# dilleridir. Bu dillerden bir tanesinin özel bir konumu vardır. Tahmin edeceğimiz gibi bu dil C#'tır. C# .NET platformu için sıfırdan geliştirilmiş yeni bir dildir. Diğer diller ise eski versiyonları değiştirilerek .NET'e uyumlu hale getirilmiştir. Özellikle Visual Basic dilinin devamı gibi görünen VB.NET dilinde bir çok radikal değişiklik yapılmıştır. Örneğin VB dili nesne yönelimli programlama tekniğini destekler hale getirilmiştir. Bu eklentilerin çok başarılı olduğu söylenemez. Çünkü bu şekildeki zoraki eklentiler dilin en başta tasarlanma amacına uygunluğunu ortadan kalkmaktadır. Bu amaçla Microsoft, hem nesne yönelimli programlama tekniğine tam destek veren, C++ dilinin güçlü özelliklerinden yoksun olmayan ve aynı şekilde Visual Basic dilinin kolaylığından esinlenerek C# dilini çıkardı.

Peki .NET dilleri arasında C#'ı tercih etmemize neden olacak başka neler var? Her şeyden önce C# öğrenilmesi kolay bir dildir. Az sayıda anahtar sözcük içermesine rağmen bir çok olanağı programcının hizmetine sunmuştur. C# nesne yönelimli programlama diline tam destek verdiği içinde seçilebilir. C#'ta değişken kavramı neredeyse kalkmıştır. Bunda bütün temel veri türleri de dahil olmak üzere bütün sınıfların Object diye adlandırılan bir sınıftan türetilmesinin etkisi vardır. C# dili güç ve hızlilik arasındaki dengeye estetik bir şekilde korumaktadır. Temsilci ve olaylarla VB'deki olay mantığına benzer bir model sunarken aynı zamanda göstericileri kullanmaya imkan vererek C++ dilinin güçlü özelliklerinden yoksun bırakmamıştır. .NET sınıf kütüphanesinin büyük bir kısmı C# ile geliştirilmiştir. Yani bu kütüphaneyi en etkin biçimde C# ile kullanabiliriz. Dahası C# dili .NET'in çalışma mimarisi de gözönünde bulundurularak sıfırdan tasarlandığı için .NET'in bütün olanaklarından en etkin biçimde C# ile faydalanabiliriz.

C# için söylenebilecek son söz : C#, modern programlama tekniklerine tam destek veren, internet çağının gerektirdiği tüm yazılım bileşenlerini geliştirmeye izin veren, hızlı ve etkin bir şekilde kodlama yapılabilen, C++ ve JAVA'nın güzel yönlerini alıp geriye dönük uyumluluğu JAVA'da olduğu gibi gözardı etmeyen bir programlama dilidir.

Sonuç

İnternet'in ve haberleşme teknolojisinin çok ileri bir seviyede olduğu bir dönemde internet üzerinde kullanılabilecek yazılım bileşenlerini programlamak son derece önem kazanmıştır. Her ne kadar C# ve JAVA öncesi dillerle herşey yapılabiliyor olsada projelerin boyutlarının büyümesi bu dillerin artık yetersiz olduğunun bir göstergesidir. Özellikle yeni nesil iş uygulamalarında C# ve JAVA, C++'tan bir adım önde görünüyor. Tabi bu durum C++ dilinin kötü olduğunu göstermez. Nitekim C# ve JAVA dillerinin her ikisinde C++ dilini örnek almıştır. Değişen tek şey günün ihtiyaçlarıdır. Aynı zamanda C# dili JAVA, C++ .NET, VB.NET ve J# gibi diller önünde de bir adım önde görünüyor.

MD5 ile Veri Şifreleme

Bu makalemizde herhangi bir string ifadenin nasıl MD5 ile şifreleneceğini öğreneceğiz. Bu sırada web.config, Panel Nesnesi, Stored Procedure gibi konulara da değineceğiz.

Aşağıda verdiğim örnek, çoğu zaman kullandığımız Kayıt Formu ile Login Formundan oluşuyor. Kayıt olurken, email adresi ve parola bilgileri soruluyor. Bunun sonrasında parola bilgisi MD5 algoritması ile şifrelenip veritabanına veriler yazılıyor.

Login Formumuzda ise, aynı veriler istenerek, yine parolamız MD5 algoritması ile veritabanına gönderiliyor. Yani SQL'deki "Select" cümlesi aracılığı ile kontrolümüzü yapıyoruz.

Örneğimize geçmeden önce örneğimiz içerisinde kullandığımız Panel nesnemizin bazı özelliklerini inceleyelim.

Height = Panelimizin yüksekliği (pixel cinsinden)
Width = Panelimizin genişliği (pixel cinsinden)
BackColor = Panelimizin arkafon rengi
BackImageUrl = Panelimizin arkasında resim göstermek istiyorsak
BorderColor = Panelimizin sınır çizgisinin rengi
BorderWidth = Panelimizin sınır çizgisinin genişliği (pixel cinsinden)
Font = Panelimizin içerisinde gösterilecek metinlerin Font adı
Visible = Panelimizin görüntülenme ayarı (true/false değerleri alır)

Şimdi de veritabanına bağlanmak amaçlı kullandığımız bağlantı satırımızı nasıl kullandığımıza bakalım.

Klasik ASP içerisinde veritabanına bağlanmak istediğimizde bunu çoğu zaman asp dosyamızın içerisine yazıyorduk. Veya başka bir sayfaya yazıp, onu kullanacağımız sayfaya dahil ediyorduk. Hatırlarsanız bu yöntem "Include File" yöntemi deniyordu. Bu durum güvenlik açısından birçok açık ortaya çıkartmak ile beraber, yetersiz de kalıyordu.

.Net'te ise bu sıkıntılar atlatıldı. Şimdi projemiz ile ilgili birçok veriyi saklayabileceğimiz, güvenli bir dosyaya kavuştuk. İşte bu dosyanın adı web.config

Web.config dosyasının ayrıntılarını burada işleyemeyeceğim. Sadece veritabanı bağlantı satırımızı nasıl web.config sayfamıza yazmamız gerektiğini ve aspx dosyamızdan nasıl çağırıldığını göstereceğim.

Örneğin **web.config** dosyamızın içeriği:

```
<configuration>
  <appSettings>
    <add key="strConn"
value="server=localhost;uid=dtuser;pwd=dtpass;database=dotnet" />
  </appSettings>
</configuration>
```

Gelelim aspx dosyamızdan nasıl çağırabileceğimize:

```
string dbConnStr = ConfigurationSettings.AppSettings["strConn"];
```

Sanırım artık konumuza dönebiliriz. İlgili tüm açıklamaları kod satırları arasında anlatmağa çalıştım. Ayrıca CodeBehind yönetimi kullanarak kodladım. Bu yöntemden de kısaca bahsetmek gerekirse, CodeBehind yöntemi ile kodumuz ile görselliğimizi tamamen ayırıyoruz. Böylelikle tasarım değişikliği gibi durumlarda hiçbir sıkıntı çekmiyoruz. Örneği incelediğinizde durumu da farkedeceksiniz.

Fakat öncelikle, veritabanımızın yapısını, stored procedure ve web.config dosyamızın ilgili kodlarını verelim.

Tablomuz:

registerUser

```
user_id int (IDENTITY)
user_email varchar 255
user_password binary 16
```

Stored Procedure:

sp_ins_regUser

```
CREATE PROCEDURE sp_ins_regUser
    @user_email varchar(255),
    @user_password binary(16)
AS
INSERT INTO
    registerUser
    (user_email, user_password)
VALUES
    (@user_email, @user_password)
GO
```

sp_sel_loginCheck

```
CREATE PROCEDURE sp_sel_loginCheck
    @uemail varchar(255),
    @upwd binary(16)
AS
SELECT
    user_id
FROM
```

```
        registerUser
WHERE
        user_email = @uemail AND
        user_password = @upwd
GO
```

Web.config

```
<configuration>
  <appSettings>
    <add key="strConn"
value="server=localhost;uid=dtuser;pwd=dtpass;database=dotnet" />
  </appSettings>
</configuration>
```

İlk önce görsel arayüzümün bulunduğu, kişinin kayıt olduğu sayfa olan:

register.aspx

```
<%@ Page Language="c#" Inherits="registerForm.regForm" Src="register.aspx.cs"%>
<html>
<body>
<asp:Panel id="register" runat="server">
  <form id="regForm" method="post" runat="server">
    <table cellpadding="3" cellspacing="0" border="0">
      <tr>
        <td colspan="2">Kayıt Formu</td>
      </tr>
      <tr>
        <td><b>E-posta Adresiniz</b></td>
        <td><asp:TextBox id="emailAdr" runat="server"
MaxLength="255"></asp:TextBox></td>
      </tr>
      <tr>
        <td><b>Parolanız</b></td>
        <td><asp:TextBox runat="server" MaxLength="10" id="parola"
TextMode="Password"></asp:TextBox></td>
      </tr>
      <tr>
        <td align="right" colspan="2"><asp:Button id="btnOk"
OnClick="doRegister" runat="server" text="Formu Gönder"/></td>
      </tr>
    </table>
  </form>
</asp:Panel>
<asp:Panel id="registerStatus" runat="server">
  <asp:Label id="lblInfo" runat="server"></asp:Label>
</asp:Panel>
</body>
</html>
```

Bu sayfanın kodlarını işleyen:

register.aspx.cs

```
using System;
using System.IO;
using System.Web.UI; //web textbox larına ulaşabilmemiz için gereken class
using System.Security.Cryptography; //md5 için gerekli class
using System.Text; //UTF fonksiyonu için gerekli class
using System.Data; //veritabanı işlemleri için gerekli class
using System.Data.SqlClient; //veritabanı işlemleri için gerekli class
using System.Configuration; //web.config dosyamızdan veri okuyabilmek amaçlı class

namespace registerForm
{
    public class regForm : System.Web.UI.Page
    {
        //kodlamada kullanacağımız nesnelerimizi tanımlıyoruz.
        protected System.Web.UI.WebControls.TextBox emailAdr;
        protected System.Web.UI.WebControls.TextBox parola;

        protected System.Web.UI.WebControls.Panel register;
        protected System.Web.UI.WebControls.Panel registerStatus;

        protected System.Web.UI.WebControls.Label lblInfo;

        public void Page_Load(Object Src, EventArgs E)
        {
            //sayfa yüklendiğinde panellerimizin görüntülenme ayarlarını yapıyoruz.

            //form ekranı ilk olarak görüntülenecek.
            register.Visible = true;
            registerStatus.Visible = false;
            lblInfo.Text = "";
        }

        //Formu Gönder butonuna tıklandığında çalışan fonksiyonumuz
        protected void doRegister(object sender, EventArgs e)
        {
            //girilen verileri alıyoruz.
            string txtEmailAdr = emailAdr.Text;
            string txtParola = parola.Text;

            //işlem sonucu göstermek amaçlı ikinci panelimizi görünür kılıyoruz.

            register.Visible = false;
            registerStatus.Visible = true;

            try
            {
                //parolanız şifrlenmesi için fonksiyona gönderiyoruz.
                //şifrlenmiş verimiz byte haline geleceği için
```

değişkenimizi

```
//byte olarak tanımlıyoruz.  
byte[] encryptedPassword =
```

```
md5Password(txtParola);
```

```
//veritabanına Email adresini ve şifrelenmiş Parolayı
```

kayıt ediyoruz.

```
SqlConnection conn = new  
SqlConnection(ConfigurationSettings.AppSettings["strConn"]);  
conn.Open();  
SqlCommand sc = new SqlCommand ();  
sc.Connection = conn;  
sc.CommandType = CommandType.StoredProcedure;  
sc.CommandText = "sp_ins_regUser";
```

```
sc.Parameters.Add("@user_email",  
SqlDbType.VarChar, 255, "user_email");  
sc.Parameters["@user_email"].Value = txtEmailAdr;  
sc.Parameters.Add("@user_password",  
SqlDbType.Binary, 16, "user_password");  
sc.Parameters["@user_password"].Value =  
encryptedPassword;
```

```
sc.ExecuteNonQuery();  
conn.Close();
```

```
//try-catch bloğuna soktuğumuz işlemimizde bir
```

sorun çıkmadı ise

```
//ziyaretçimizi bilgilendiriyoruz.
```

```
lblInfo.Text = "Kayıt işleminiz başarı ile
```

```
gerçekleştirilmiştir";
```

```
}  
catch  
{
```

```
//veritabanında bir hata oluştuysa ziyaretçimizi
```

bilgilendiriyoruz.

```
lblInfo.Text = "Kayıt işleminiz sırasında bir hata
```

```
oluştı. Lütfen tekrar deneyiniz.";
```

```
}  
}
```

```
byte[] md5Password(string pass)  
{
```

```
//md5 şifrelenmesi için verimizin byte haline gelmesi gerekli.  
//veri 8-bit şeklinde dönüştürülmesi için ilk önce UTF
```

fonksiyonuna gönderiliyor.

```
UTF8Encoding encoder = new UTF8Encoding();  
//md5 şifrelemesi için nesnemizi oluşturuyoruz.  
MD5 md5 = new MD5CryptoServiceProvider();  
//verimizi md5 ile çalıştırıyoruz.  
//dikkat ederseniz UTF fonksiyonundan dönen değeri GetBytes
```

ile alabildik.

```
byte[] donenDeger =  
md5.ComputeHash(encoder.GetBytes(pass));  
//şifrelenmiş değerimizi geri gönderiyoruz.  
return donenDeger;
```

```
}  
  
}  
  
}
```

Kullanıcı adı, parola verilerinin girildiği görsel sayfa olan:

login.aspx

```
<%@ Page Language="c#" Inherits="loginForm.Login" Src="login.aspx.cs"%>  
<html>  
<body>  
<asp:Panel id="loginStatus" runat="server">  
    <asp:Label id="lblInfo" runat="server"></asp:Label>  
</asp:Panel>  
  
<asp:Panel id="pnlLogin" runat="server">  
    <form id="loginForm" method="post" runat="server">  
        <table cellpadding="3" cellspacing="0" border="0">  
            <tr>  
                <td colspan="2">Siteye Giriş</td>  
            </tr>  
            <tr>  
                <td><b>E-posta Adresiniz</b></td>  
                <td><asp:TextBox id="emailAdr" runat="server"  
MaxLength="255"></asp:TextBox></td>  
            </tr>  
            <tr>  
                <td><b>Parolanız</b></td>  
                <td><asp:TextBox id="parola" runat="server"  
MaxLength="10" TextMode="Password"></asp:TextBox></td>  
            </tr>  
            <tr>  
                <td align="right" colspan="2"><asp:Button id="btnLogin"  
OnClick="doLogin" runat="server" text="Formu Gönder"/></td>  
            </tr>  
        </table>  
    </form>  
</asp:Panel>  
</body>  
</html>
```

login.aspx dosyamızı işleyen sayfamız:

login.aspx.cs

```
using System;
```

```
using System.IO;
using System.Web.UI; //web textbox larına ulaşabilmemiz için gereken class
using System.Security.Cryptography; //md5 için gerekli class
using System.Text; //UTF fonksiyonu için gerekli class
using System.Data; //veritabanı işlemleri için gerekli class
using System.Data.SqlClient; //veritabanı işlemleri için gerekli class
using System.Configuration; //web.config dosyamızdan veri okuyabilmek amaçlı class
```

```
namespace loginForm
```

```
{
```

```
    public class Login : System.Web.UI.Page
```

```
    {
```

```
        //kodlamada kullanacağımız nesnelerimizi tanımlıyoruz.
```

```
        protected System.Web.UI.WebControls.TextBox emailAdr;
```

```
        protected System.Web.UI.WebControls.TextBox parola;
```

```
        protected System.Web.UI.WebControls.Panel pnlLogin;
```

```
        protected System.Web.UI.WebControls.Panel loginStatus;
```

```
        protected System.Web.UI.WebControls.Label lblInfo;
```

```
        public void Page_Load(object sender, System.EventArgs e)
```

```
        {
```

yapıyoruz.

```
            //sayfa yüklendiğinde panellerimizin görüntülenme ayarlarını
```

```
            //form ekranı ilk olarak görüntülenecek.
```

```
            pnlLogin.Visible = true;
```

```
            loginStatus.Visible = false;
```

```
            lblInfo.Text = "";
```

```
        }
```

```
        //Formu Gönder butonuna tıklandığında çalışan fonksiyonumuz
```

```
        protected void doLogin(object sender, System.EventArgs e)
```

```
        {
```

```
            //girilen verileri alıyoruz.
```

```
            string uid = emailAdr.Text;
```

```
            string pwd = parola.Text;
```

kılıyoruz.

```
            //işlem sonucu göstermek amaçlı ikinci panelimizi görünür
```

```
            pnlLogin.Visible = false;
```

```
            loginStatus.Visible = true;
```

```
            //parolanız şifrlenmesi için fonksiyona gönderiyoruz.
```

```
            //şifrlenmiş verimiz byte haline geleceği için değişkenimizi
```

```
            //byte olarak tanımlıyoruz.
```

```
            byte[] encryptedPwd = md5Password(pwd);
```

```
            //veritabanına ilgili verileri göndererek kontrolümüzü yaparız.
```

```
            SqlConnection conn = new
```

SqlConnection(ConfigurationSettings.AppSettings["strConn"]);

```
            conn.Open();
```

```
            SqlCommand sc = new SqlCommand ();
```

```
            sc.Connection = conn;
```

```
            sc.CommandType = CommandType.StoredProcedure;
```

```
            sc.CommandText = "sp_sel_loginCheck";
```



```

"uemail");
        sc.Parameters.Add("@uemail", SqlDbType.VarChar, 255,
        sc.Parameters["@uemail"].Value = uid;
        sc.Parameters.Add("@upwd", SqlDbType.Binary, 16, "upwd");
        sc.Parameters["@upwd"].Value = encryptedPwd;

        try
        {
            //Elimizdeki verilerle çalıştırdığımız SP'mizden geri bir
            //için SqlCommand nesnesinin ExecuteScalar()
            //user_id şunda bizim işimize yaramıyor, fakat nasıl
            //bu satırı da kodumuza ekledim.
            string user_id = sc.ExecuteScalar().ToString();
            //Email ve parola doğru ise bilgilendiriyoruz.
            lblInfo.Text = "Hoşgeldiniz ";
        }
        catch
        {
            //Email ve parola yanlışsa tekrar girmesini istiyoruz.
            lblInfo.Text = "Yanlış E-posta Adresi/Parola. Lütfen
            bilgilerinizi kontrol edip tekrar deneyiniz.";
        }
        conn.Close();
    }

    byte[] md5Password(string pass)
    {
        //md5 şifrelenmesi için verimizin byte haline gelmesi gerekli.
        //veri 8-bit şeklinde dönüştürülmesi için ilk önce UTF
        UTF8Encoding encoder = new UTF8Encoding();
        //md5 şifrelemesi için nesnemizi oluşturuyoruz.
        MD5 md5 = new MD5CryptoServiceProvider();
        //verimizi md5 ile çalıştırıyoruz.
        //dikkat ederseniz UTF fonksiyonundan dönen değeri GetBytes
        byte[] donenDeger =
        md5.ComputeHash(encoder.GetBytes(pass));
        //şifrelenmiş değerimizi geri gönderiyoruz.
        return donenDeger;
    }
}

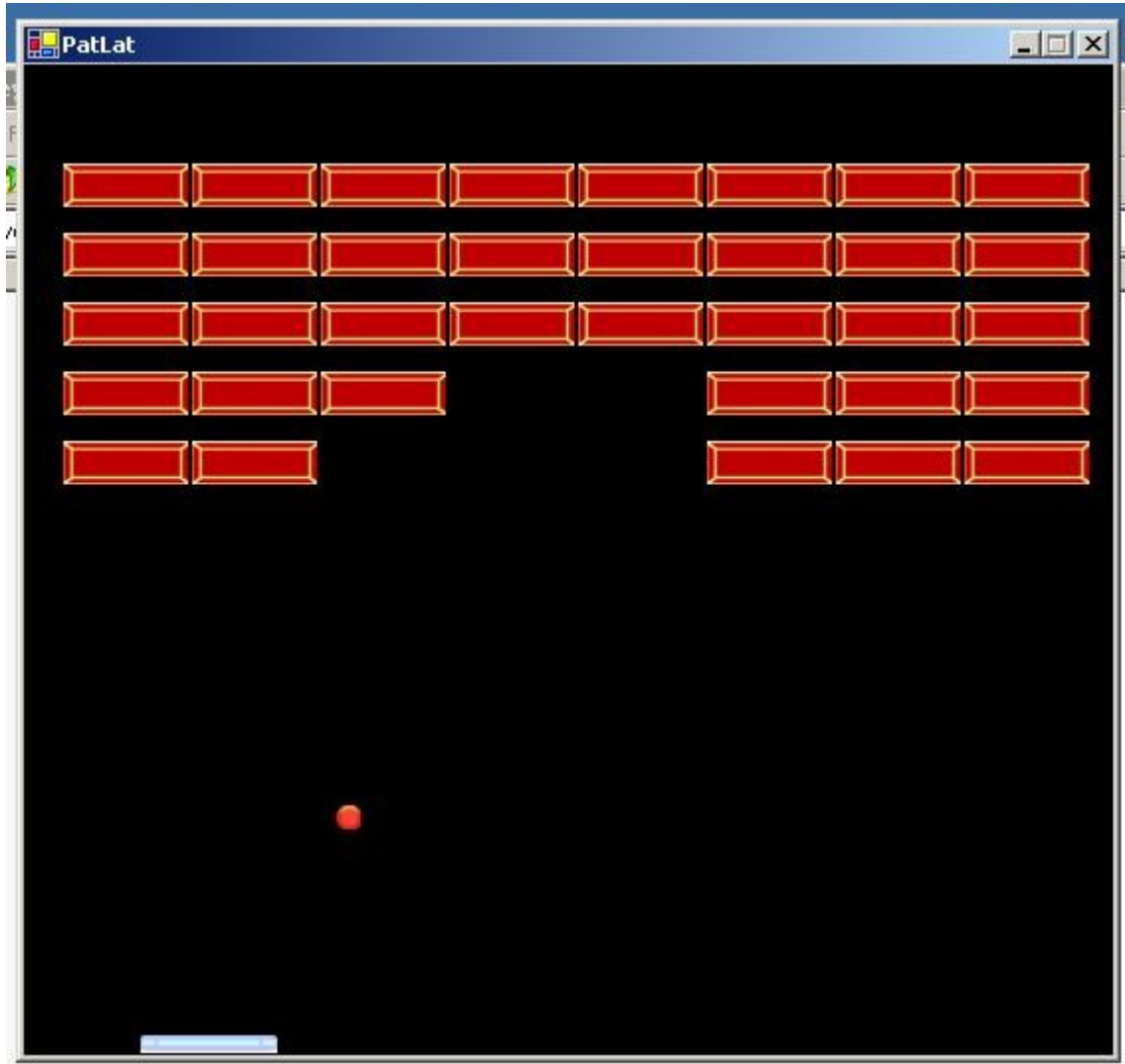
```

C# ve GDI+ Kullanılarak Yapılan DXBALL Oyunu

Bu yazıda anlatacađım oyun, bazılarının DXBall, bazılarının Alleyway... diye bildiđi bir çok ismi olan bir oyun.

Oyunun tm dosyaları burada verilmeyecektir ancak .NET ile yeni bir proje aılıp .cs dosyaları kopyalanırsa, oyun alıřacaktır. Ayrıca oyunda kullanılan top,blok şekilleri iin gerekli olan resimleri de istediđiniz gibi seebilirsiniz. Top resminin dosya adı tp.bmp, blokların dosya adı brick.bmp, alttaki cubuk resminin dosya adı ise blk.bmp olmak zorundadır.'dir. Oyunu normal haliyle alıřtırmak istiyorsanız download edebilirsiniz. (Bilgisayarda .NET ve Framework 1.1 kurulu olmak zorunda!)

Programı derleyip alıřtırdığınızda oyunun ekran grntsnn ařađıdaki gibi olduđunu greceksiniz.



Oyunun kaynak kodunu indirmek iin tıklayın.

nemli: Sinifların neler olduđunu uzun uzun paragraflar halinde anlatmaktansa kodun yanında komut satırlarıyla anlatılacaktır.

Simdi adim adim oyunu açıklayalım;

- İlk olarak Top.cs dosyamız var. top sınıfımızda, en önemli metot olan move metodunda topun koordinatlarına bakılır ve koordinatlara göre ne tarafa hareket ettirileceği belirlenir.

Top.cs:

```
using System;

using System.Drawing;

using System.Drawing.Imaging;

namespace alleyway

{

public class Top

{

public Point pos; //topun koordinatlarini tutmaya yarayacak.

private Bitmap tp=null; //topun resmi için gerekli.

public int Xhareket=4; //topun x'de hareket hizini burada belirleyecegiz.

public int Yhareket=4; //topun y'de hareket hizini burada belirleyecegiz.

public Top(int x,int y)

{

pos.X = x; //ana formdan gelen topun nerede baslatilacagi bilgisi

pos.Y = y; // burada x ve y koordinatlari olarak girilir.

if (tp==null) //eger resim yüklenmediyse yükleme islemi yapilacak.

{

tp = new Bitmap("tp.bmp");

}

}

public Rectangle GetFrame()

{

Rectangle myRect = new Rectangle(pos.X, pos.Y, tp.Width, tp.Height);
```

```

return myRect; //top nesnemizin konumunu dikdörtgen olarak belirlememize yarar.

} // bu sayede top ile blokların çarpışıp çarpışmadığını daha kolay
//anlayacağız.

public void Draw(Graphics g)
{
    g.DrawImage (tp,pos.X,pos.Y,tp.Width ,tp.Height ); //top nesnesi ekrana
        //çizdirilir. X,y koordinatlarından sonra ebatları parametre olarak girilir.
}

public void Remove(Top t)
{
    Yhareket=-Yhareket;
    pos.X += Xhareket;
    pos.Y += Yhareket;
}

public void Move(int hak,Grup grp,Top t,Rectangle r, Rectangle rblok)
{
    //topun pencere içinde duvarlara ve cubuğa çarptıkça simetrik olarak sekmesini
    //burada sağlayacağız.

    if(t.GetFrame().Intersects(rblok)) //eger top bloka çarptıysa,
    {
        Yhareket=-Yhareket; // topun y yönünü ters çevir
    } // böylelikle sekmeyi sağla.

    //top ekrandaysa,
    if(pos.X > 539 || pos.X < 0)
    {
        //topun x hareketini ters çevir
        Xhareket = -Xhareket;
    }
}

```

```

}

// top yukari carparsa
if(pos.Y < 0)
{
    //y hareketini degistir
    Yhareket = -Yhareket;
} // yanarsa,
else if(pos.Y > 500)
{
    //top durur
    Xhareket = 0;
    Yhareket = 0;
    //hakki 1 azalt
    hak -= 1;
}

// top koordinatlarini degistir
pos.X += Xhareket;
pos.Y += Yhareket;
}
}
}

```

- Blok.cs sinifi ise alt taraftaki yönlendireceğimiz cubuktur.

Blok.cs:

```

using System;

using System.Drawing;

using System.Drawing.Imaging;

namespace alleyway

```

```
{

public class Blok

{

public Point pos; //x ve y koordinatlarini belirlemek için kullanılacak.

static Bitmap Cubuk = null; //Çubuk resmi için kullanılacak.

int inc = 6; //çubugun sürüklemeye hızını belirleyecek

int LastposX = 0; //su anda son pozisyon olmadigindan ikisi de sıfırlanır.

int LastposY = 0;

public Blok(int x, int y)

{

pos.X = x; //x konumu verilir.

pos.Y = y; //y konumu verilir.

if (Cubuk == null) //çubuk ekranda yoksa,

{

Cubuk = new Bitmap("blk.bmp"); //çubuk ekrana çizdirilir.

}

}

public Rectangle GetFrame()

{

Rectangle myRect = new Rectangle(pos.X, pos.Y, Cubuk.Width, Cubuk.Height); //top

sinifindaki islevi görür.

return myRect;

}

public void Draw(Graphics g)

{

Rectangle destR = new Rectangle(pos.X, pos.Y, Cubuk.Width, Cubuk.Height); //çubuk

hareketlerinde kullanılacak.

Rectangle srcR = new Rectangle(0,0, Cubuk.Width, Cubuk.Height);
```

```
g.DrawImage(Cubuk, destR, srcR, GraphicsUnit.Pixel);
```

```
LastposX = pos.X;
```

```
LastposY = pos.Y;
```

```
}
```

```
public void MoveLeft(Rectangle r)
```

```
{
```

```
if (pos.X <= 0) //eger pencerenin en solundaysa hareket etmeyecek, degilse sola dogru hareketine devam edecektir.
```

```
return;
```

```
pos.X -= inc;
```

```
}
```

```
public void MoveRight(Rectangle r)
```

```
{ //eger pencerenin en sagindaysa hareket etmeyecek, degilse saga dogru hareketine devam edecektir.
```

```
if (pos.X >= r.Width - Cubuk.Width)
```

```
return;
```

```
pos.X += inc;
```

```
}
```

```
}
```

```
}
```

- Brick.cs sinifi ise patlatilacak bloklari temsil eder. Burada sadece basit özellikleri tuttur. Bloklari kontrol eden Grup sinifinda tüm bloklari ayni anda kontrol edildiğini göreceksiniz. Grup sinifinda tüm bloklari tutan bir matris bulunmaktadır.

Brick.cs:

```
using System;
```

```

using System.Drawing;

using System.Drawing.Imaging;

namespace alleyway

{

public class Brick

{

public int yukseklik,genislik,basx,basy; //blokların özellikleri burada tutulur.

public bool vuruldu=false; //top tarafından vurulup vurulmadığı buradan anlaşılabilecektir.

public Brick(int yuk,int gen,int startx,int starty)

{

this.yukseklik=yuk;

this.genislik=gen;

this.basx=startx;

this.basy=starty;

this.vuruldu=false; //yeni yaratıldığı için vurulmadı olarak işaretlenir.

}

}

}

```

- Grup.cs sınıfında ise blok kontrolü yapılır.

Grup.cs:

```

using System;

using System.Drawing;

using System.Drawing.Imaging;

namespace alleyway

{

public class Grup

{

```



```

Bitmap br=null;

public Brick[,] yapi; //brick sinifinda nesneleri tutan matrisimiz burada yaratiliyor.

int sut=5; //blokların sütun ve satır sayıları bu kısımda girilir.

int sat=8;

public Grup()

{

reset(); //vurulmuş durumda olan bloklar varsa bunları da vurulmadı olarak işaretleyen
ve ilk haline döndüren metod çağrılır.

}

public void Draw(Graphics g)

{

for (int i=0;i<SAT;i++)< font>

{ // bu iç içe for döngüleri bölümünde bloklar tek tek dolasılarak

for (int j=0;j< çizdirilmez) yapılır(ekrana görünmez varsa olan vurulmuş>

{ // vurulmayanlar ise aynı şekilde ekrana çizdirilir.

if (yapi[i,j].vuruldu==false)

{

Brick temp= (Brick) yapi[i,j];

g.DrawImage(br,temp.basx,temp.basy,br.Width,br.Height);

}

}

}

}

public void reset(){ //tüm bloklar tekrar oluşturulur.

yapi = new Brick[sat,sut];

if (br==null)

{

```

```

br = new Bitmap("Brick.bmp");

}

for (int i=0;i< olarak vurulmadi bloklar dolasilarak matris>

{

for (int j=0;j<SUT;J++)< font>

{

yapi[i,j]=new Brick(br.Height,br.Width,(i) * 65 +20, (j * 35) + 50);

yapi[i,j].vuruldu=false;

}

}

}

}

}

```

- Form1.cs,oyunun ana formudur yani ekranimizdir. Bu formda esas önemli olan yerler açıklanacaktır.

Form1.cs:

```

using System;

using System.Drawing;

using System.Collections;

using System.ComponentModel;

using System.Windows.Forms;

using System.Data;

using System.Threading;

using System.Runtime.InteropServices;

namespace alleyway

{

public class Form1 : System.Windows.Forms.Form

```

```

{

public int Hak=3; //kaç hakkımız olduğunu belirleriz.

int puan=0; //puan başlangıçta sıfırdır.

private bool flag = true;

private Blok blok = new Blok(275, 490); //alttaki çubugun yeri belirlenir.

private Top top = new Top(250, 300); // topun konumu belirlenir.

private Grup grup=new Grup();

private System.Windows.Forms.Timer timer1;

private System.ComponentModel.IContainer components;

public Form1()

{

InitializeComponent();

timer1.Start(); //top hareketleri ve ekran değişimleri bu timer sayesinde
belirlenir. Her 50 salisede ekran tekrar

//çizdirilir.

SetStyle(ControlStyles.UserPaint, true);

SetStyle(ControlStyles.AllPaintingInWmPaint, true);

SetStyle(ControlStyles.DoubleBuffer, true);

}

protected override void Dispose( bool disposing ) //bu kodlari .net kendisi yaratir.

{

if( disposing )

{

if (components != null)

{

components.Dispose();

}

}

}

```

```

}

base.Dispose( disposing );

}

#region Windows Form Designer generated code

///
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
///

private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.timer1 = new System.Windows.Forms.Timer(this.components);

    // timer1 özelliklerini burada tanımlarız.

    this.timer1.Enabled = true;
    this.timer1.Interval = 50;
    this.timer1.Tick += new System.EventHandler(this.timer1_Tick);

    // Form1 için yapılan değişiklikler buradadır.

    this.AutoScaleBaseSize = new System.Drawing.Size(6, 16);
    this.ClientSize = new System.Drawing.Size(550, 500);
    this.Font = new System.Drawing.Font("Comic Sans MS", 8.25F,
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((System.Byte)
    (0)));
    this.KeyPreview = true;
    this.Location = new System.Drawing.Point(150, 200);
    this.MaximizeBox = false;
    this.Name = "Form1";
    this.StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen;
    this.Text = "PatLat";

```

```

this.KeyDown += new System.Windows.Forms.KeyEventHandler(this.Form1_KeyDown);

this.Load += new System.EventHandler(this.Form1_Load);

this.Paint += new System.Windows.Forms.PaintEventHandler(this.Form1_Paint);
}

#endregion

[STAThread]

static void Main()

{

Application.Run(new Form1());

}

private void Form1_Paint(object sender, System.Windows.Forms.PaintEventArgs e)

{

Graphics g = e.Graphics;

g.FillRectangle(Brushes.Black, 0, 0, this.ClientRectangle.Width, ClientRectangle.Height);

blok.Draw(g); //oyunumuzdaki tüm nesneler ekrana çizdirilir.

top.Draw (g);

grup.Draw(g);

}

private void Form1_KeyDown(object sender, System.Windows.Forms.KeyEventArgs e)

{ //bu kismda tusa basilinca çubugun hareket ettirilmesi gerçekleştirilmiştir.
Case yapisi içinde sag saol hareketleri

//yapilir.

string result = e.KeyData.ToString();

Invalidate(blok.GetFrame());

switch (result)

{

case "Left": //sol tusuna basildiysa sola git

```

```

blok.MoveLeft(ClientRectangle);

Invalidate(blok.GetFrame()); //tekrar blogu çizdir.

break;

case "Right": //sag tusuna basildiysa saga git

blok.MoveRight(ClientRectangle);

Invalidate(blok.GetFrame()); //tekrar blogu çizdir.

break;

default:

break;

}

}

private void timer1_Tick(object sender, System.EventArgs e)

{ //timer 'in her tikiinde top ve bloklar kesisti mi? kontrolü yapılır ve kesisme
varsa o blok vurulduğu için ekrandan silinir.

flag=true;

for (int i=0;i<8;i++)

{

for (int j=0;j<5 ;j++)

{

Rectangle r_brick = new
Rectangle(grup.yapi[i,j].basx,grup.yapi[i,j].basy,grup.yapi[i,j].genislik,grup.yapi[i,j].yuks
eklik);

if(top.GetFrame().IntersectsWith(r_brick))

{

if (grup.yapi[i,j].vuruldu==false)

{

grup.yapi[i,j].vuruldu=true;

top.Remove(top); //ekrandan topun silinmesi.

```

```
puan=puan+10; //blok vuruldu, puani arttir.

flag=false;//yanmadigimizi belirtmek için flag false yapilir
}

}

}

}

if (flag) //flag true ise yandi demektir.

{

top.Move (Hak,this.grup,this.top,ClientRectangle, blok.GetFrame ());

if (top.pos.Y>500)

{

timer1.Stop(); //hareketi durdur.

Hak--; //hakki 1 azalt

if (Hak==0) //eger hak sifir ise,

{

Form3 form3=new Form3(puan); //oyun bitti ekranini göster.

form3.ShowDialog(this);

if (form3.DialogResult==DialogResult.OK) // O ekranda tekrar oynaya basilirsa,

{

Hak=3; //hakki 3'e çıkar,

puan=0; //puani sifirla ve tekrar baslat.

top.pos.X=265;

top.pos.Y=300;

blok.pos.X=275;

blok.pos.Y=490;

grup.reset();

top.Xhareket=-top.Xhareket;
```

```

Invalidate();

timer1.Start();

}

}

else //hak sifir olmadiyse,

{

Form2 form2=new Form2(Hak); //kaç hakki oldugunu, puanini göster,oyuna devam
et

form2.ShowDialog(this);

top.pos.X=250;

top.pos.Y=250;

blok.pos.X=275;

blok.pos.Y=490;

top.Xhareket=-top.Xhareket;

Invalidate();

timer1.Start();

}

}

}

flag=true;

Invalidate();

}

}

}

```

- Simdi kodunu göreceginiz iki form ise oyun bitince ve yaninca gösterilen formlar oldugundan açıklanacak bir bölüm içermemektedirler.
 - **Oyun Bitince gösterilen form:**

Form3.cs:


```
using System;

using System.Drawing;

using System.Collections;

using System.ComponentModel;

using System.Windows.Forms;

namespace alleyway

{

    ///

    /// Summary description for Form3.

    ///

    public class Form3 : System.Windows.Forms.Form

    {

        int ppuan;

        private System.Windows.Forms.Label label1;

        private System.Windows.Forms.Label label2;

        private System.Windows.Forms.Button button2;

        private System.Windows.Forms.Label label3;

        private System.Windows.Forms.Button OK;

        ///

        /// Required designer variable.

        ///

        private System.ComponentModel.Container components = null;


        public Form3(int ppuan)

        {

            //

            // Required for Windows Form Designer support
```

```
//

ppuan=puan;

InitializeComponent();


//

// TODO: Add any constructor code after InitializeComponent call

//

}


///

/// Clean up any resources being used.

///

protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        if(components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose( disposing );
}


#region Windows Form Designer generated code

///

/// Required method for Designer support - do not modify
```

```

/// the contents of this method with the code editor.

///

private void InitializeComponent()

{

this.label1 = new System.Windows.Forms.Label();

this.label2 = new System.Windows.Forms.Label();

this.OK = new System.Windows.Forms.Button();

this.button2 = new System.Windows.Forms.Button();

this.label3 = new System.Windows.Forms.Label();

this.SuspendLayout();

//

// label1

//

this.label1.Font = new System.Drawing.Font("Bookman Old Style", 15.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((System.Byte)
(0)));

this.label1.Location = new System.Drawing.Point(48, 8);

this.label1.Name = "label1";

this.label1.Size = new System.Drawing.Size(144, 32);

this.label1.TabIndex = 0;

this.label1.Text = "Oyun Bitti!..";

//

// label2

//

this.label2.Font = new System.Drawing.Font("Bookman Old Style", 15.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((System.Byte)
(0)));

this.label2.Location = new System.Drawing.Point(8, 48);

this.label2.Name = "label2";

```

```
this.label2.Size = new System.Drawing.Size(112, 24);

this.label2.TabIndex = 1;

this.label2.Text = "Puaniniz:";

//

// OK

//

this.OK.DialogResult = System.Windows.Forms.DialogResult.OK;

this.OK.Location = new System.Drawing.Point(24, 88);

this.OK.Name = "OK";

this.OK.TabIndex = 2;

this.OK.Text = "Baslat";

this.OK.Click += new System.EventHandler(this.button1_Click);

//

// button2

//

this.button2.Location = new System.Drawing.Point(136, 88);

this.button2.Name = "button2";

this.button2.TabIndex = 3;

this.button2.Text = "Yeter";

this.button2.Click += new System.EventHandler(this.button2_Click);

//

// label3

//

this.label3.Font = new System.Drawing.Font("Bookman Old Style", 15.75F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, ((System.Byte)
(0)));

this.label3.Location = new System.Drawing.Point(120, 48);

this.label3.Name = "label3";
```

```

this.label3.Size = new System.Drawing.Size(56, 24);

this.label3.TabIndex = 4;

this.label3.Click += new System.EventHandler(this.label3_Click);

//

// Form3

//

this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);

this.BackColor = System.Drawing.Color.SkyBlue;

this.ClientSize = new System.Drawing.Size(234, 122);

this.ControlBox = false;

this.Controls.Add(this.label3);

this.Controls.Add(this.button2);

this.Controls.Add(this.OK);

this.Controls.Add(this.label2);

this.Controls.Add(this.label1);

this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedDialog;

this.MaximizeBox = false;

this.MinimizeBox = false;

this.Name = "Form3";

this.StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen;

this.Load += new System.EventHandler(this.Form3_Load);

this.ResumeLayout(false);

}

#endregion

private void Form3_Load(object sender, System.EventArgs e)

{

```

```

this.label3.Text = ppuan.ToString();

}

private void button2_Click(object sender, System.EventArgs e)

{

Application.Exit();

}

}

}

```

- **Yaninca gösterilen form**

Form2.cs:

```

using System;

using System.Drawing;

using System.Collections;

using System.ComponentModel;

using System.Windows.Forms;

namespace alleyway

{

///

/// Summary description for Form2.

///

public class Form2 : System.Windows.Forms.Form

{

int hhak;

private System.Windows.Forms.Label label2;

private System.Windows.Forms.Button button1;

private System.Windows.Forms.Label label1;

```

```
///
/// Required designer variable.
///
private System.ComponentModel.Container components = null;

public Form2(int hak)
{
    //
    // Required for Windows Form Designer support
    //
    hhak=hak;
    InitializeComponent();

    //
    // TODO: Add any constructor code after InitializeComponent call
    //
}

///
/// Clean up any resources being used.
///
protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        if(components != null)
        {

```

```
components.Dispose();  
  
}  
  
}  
  
base.Dispose( disposing );  
  
}
```

```
#region Windows Form Designer generated code
```

```
///
```

```
/// Required method for Designer support - do not modify
```

```
/// the contents of this method with the code editor.
```

```
///
```

```
private void InitializeComponent()
```

```
{
```

```
this.label2 = new System.Windows.Forms.Label();
```

```
this.button1 = new System.Windows.Forms.Button();
```

```
this.label1 = new System.Windows.Forms.Label();
```

```
this.SuspendLayout();
```

```
//
```

```
// label2
```

```
//
```

```
this.label2.Font = new System.Drawing.Font("Microsoft Sans Serif", 16F,  
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point,  
((System.Byte)(0)));
```

```
this.label2.Location = new System.Drawing.Point(176, 32);
```

```
this.label2.Name = "label2";
```

```
this.label2.Size = new System.Drawing.Size(40, 40);
```

```
this.label2.TabIndex = 5;
```

```
//
```



```

// button1

//
this.button1.Font = new System.Drawing.Font("Microsoft Sans Serif", 12F,
System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point,
((System.Byte)(0)));

this.button1.Location = new System.Drawing.Point(232, 16);

this.button1.Name = "button1";

this.button1.Size = new System.Drawing.Size(88, 40);

this.button1.TabIndex = 4;

this.button1.Text = "Devam";

this.button1.Click += new System.EventHandler(this.button1_Click);

//

// label1

//
this.label1.Font = new System.Drawing.Font("Comic Sans MS", 16F,
System.Drawing.FontStyle.Bold, System.Drawing.GraphicsUnit.Point,
((System.Byte)(0)));

this.label1.Location = new System.Drawing.Point(24, 0);

this.label1.Name = "label1";

this.label1.Size = new System.Drawing.Size(136, 80);

this.label1.TabIndex = 3;

this.label1.Text = "Yandiniz!.. Kalan Hak =";

//

// Form2

//
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);

this.BackColor = System.Drawing.Color.SkyBlue;

this.ClientSize = new System.Drawing.Size(330, 79);

this.ControlBox = false;

```

```
this.Controls.Add(this.label2);

this.Controls.Add(this.button1);

this.Controls.Add(this.label1);

this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.FixedDialog;

this.MaximizeBox = false;

this.MinimizeBox = false;

this.Name = "Form2";

this.StartPosition = System.Windows.Forms.FormStartPosition.CenterScreen;

this.Load += new System.EventHandler(this.Form2_Load);

this.ResumeLayout(false);

}

#endregion
```

```
private void Form2_Load(object sender, System.EventArgs e)

{

label2.Text=hhak.ToString();}
```

```
private void button1_Click(object sender, System.EventArgs e)

{

this.Close();

}

}

}
```

C# ile Yazıcı Çıktısı Alma İşlemleri

C# ile Windows iş uygulaması geliştiriyorsanız programınızın mutlaka yazıcı çıktısı alma bölümü olacaktır. Bu makalede C# ile nasıl yazıcı çıktısı alınabileceğinin temelleri üzerinde duracağım.

.NET sınıf kütüphanesi her alanda olduğu gibi yazıcı çıktısı alma ile ilgili bir takım sınıflar sağlamıştır. **PrintDocument** sınıfı yazı çıktısı alma ile ilgili en temel sınıftır. Bu yazıda bu sınıfın özelliklerini, olaylarını ve metotlarını ayrıntılı bir şekilde inceleyip tek sayfalı yada çok sayfalı yazıcı çıktısının nasıl alınabileceğini göstereceğim. Ayrıca yazıcı çıktısı alma ile çok yakından ilgili olan **PrintPreview**, **PageSetupDialog** ve **PrintDialog** gibi sınıfları da inceleyeceğiz.

PrintDocument Sınıfı

Bu sınıf programlarımıza yazıcı çıktısı alma desteğini eklemek için kullanabileceğimiz en temel yapıdır. Bu sınıf türünden bir nesne yaratıldığında çıktı alma ile ilgili hemen her tür bilgiye erişmemiz mümkündür.

```
PrintDocument YaziciCiktisi = new PrintDocument();
```

şeklinde bir tanımlama yaptığımızda varsayılan yazıcı(default printer) ile çalışılmaktadır. Bir dökümanı yazıcıya göndermek için PrintDocument sınıfının **Print()** metodu kullanılır. Print() metodu çağrıldığı anda **PrintPage** olayı meydana gelir. Bu olayı yakalayan kontrol yazıcıya gönderilecek döküman üzerinde işlemler yaparak çıktının şeklini belirlemelidir. Her bir sayfa için ayrıca PrintPage olayı meydana geleceği için her bir olay içinde doğru sayfaları yazıcıya göndermek için bir takım işlemler yapmak gerekecektir. Aksi halde her defasında birinci sayfayı yazıcıya gönderme ihtimalimiz vardır. Kısacası PrintPage olayı olmadan yazıcıya çıktı bilgilerini gönderemeyiz. Bu yüzden ilk olarak PrintPage olayını ve bu olaya ait argümanları içeren **PrintPageEventArgs** sınıfını inceleyelim.

Önce PrintPage olayının argümanlarını içeren PrintPageEventArgs sınıfının üye elemanlarını inceleyelim, ardında bir konsol uygulamasından yazıcıya nasıl bir döküman göndereceğimizi göstereceğim.

PrintPageEventArgs sınıfının üye elemanları :

Graphics : Yazıcıya gönderilecek döküman bilgilerini belirleyen grafik nesnesidir. Yazıcıya gönderilecek bilgilerin tamamı bu nesne içerisinde belirtilecektir. Not : Graphics sınıf GDI+ kütüphanesinin en önemli sınıfıdır.

Cancel : Çıktı alma işleminin iptal edilip edilemeyeceği ile ilgili bilgi veren bool türünden bir elemandır. Eğer değeri true ise çıktı alma işlemi iptal edilecektir.

HasMorePages : Yazıcıya gönderilecek çıktının birden fazla sayfa kapladığı durumlarda PrintPage olayına ilişkin metotta bu özelliğin true olarak değiştirilmesi gerekir. Böylece bundan sonraki PrintPage olaylarında bu değişken kontrol edilerek diğer sayfaların çıktıya gönderilmesi ile ilgili işlemler yapılır.

MarginBounds : Yazıcıya gönderilen çıktı dökümanının en ve boyutlarını temsil eden **Rectangle** türünden bir özelliktir. Rectangle sınıfı da GDI+ kütüphanesinin bir parçasıdır. Bu özellikte yazıcıya gönderilecek çıktının sadece üzerine çizim yapılabilen kısmı belirtilir.

PageBounds : Yazıcıya gönderilen dökümanın tamamının en ve boy değerlerini tutan yine Rectangle sınıfı türünden bir elemandır.

PageSettings: İlgili dökümana ait sayfa ayarlarını tutan ve **PageSettings** sınıfı türünden bir elemandır. PageSettings sınıfının Color, Landscape, Margins, PaperSize, PaperSource, PrinterResolution gibi sayfa ile ilgili bilgi tutan üye özellikleri bulunmaktadır.

Şimdi basit bir örnekle yazıcıya çıktı gönderelim. Örneğimizde varsayılan yazıcınıza, sol üst köşesi (20,20) koordinatlarında eni ve boyu 100 olan bir dörtgen içeren sayfayı göndereceğiz. Gönderilecek sayfadaki dörtgeni çizmek için tahmin edeceğiniz üzere Graphics nesnesini kullanacağız.

```
using System;
using System.Drawing.Printing;
using System.Drawing;

class Printer
{
    static void Main()
    {
        PrintDocument PD = new PrintDocument();
        PD.PrintPage += new PrintPageEventHandler(OnPrintDocument);

        try
        {
            PD.Print();
        }
        catch
        {
            Console.WriteLine("Yazıcı çıktısı alınamıyor...");
        }
        finally
        {
            PD.Dispose();
        }
    }

    private static void OnPrintDocument(object sender, PrintPageEventArgs e)
    {
        e.Graphics.DrawRectangle(Pens.Red,20,20,100,100);
    }
}
```

Yukarıdaki programı derleyip çalıştırdığınızda hiç bir uyarı eğer verilmeden sisteminize bir yazıcı bağlı OnPrintDocument() metodunda hazırlanan içerik yazıcıya gönderilecektir. Eğer sisteminize bağlı bir yazıcı yoksa doğal olarak catch bloğundaki kod çalışacaktır.

Çizilen dörtgen nesnesinin kağıdın neresine basılacağını biz belirliyoruz. MarginBounds özelliğini kullanarak çizilecek içeriğin doğru noktaya çizilmesini sağlayabiliriz. Bu özellik sizin yazıcı ayarlarınız ile ilgili olduğu için programlama yolu ile kod içerisinden değiştirilemez. Yani bu özellik "read only" bir özelliktir. Dikkat edilmesi gereken diğer bir noktada yazıcıya gönderilecek içeriğin PageBounds özelliği ile belirtilen dörtgenin dışına taşmamasıdır. Bu yüzden çizimleri yapılırken bu özellik baz alınmalıdır.

Yukarıda yazdığımız basit programda eksiklik bulunmaktadır. Bu eksiklik çizilecek dörtgenin tek bir sayfaya sığmadığı durumlarda görülür. Söz gelimi eğer dörtgenin yüksekliğini 2000 yaparsak yazıcıdan sadece ilk kağıda sığan bölümü çıkacaktır. Birden fazla sayfası olan çıktıları yazıcıya göndermek için PrintPageEventArgs sınıfının **HasMorePages** özelliği kullanılır. Bu özellik OnPrintDocument() metodu içerisinde **true**

değerine çekilerek çıktı alma işleminin devam ettiği belirtilmelidir. Ayrıca her bir sayfanın içeriğinde metot her çağrıldığında farklı bir biçimde oluşturulacağı için programcının bu ayrımı da kodlaması gerekmektedir. Örneğin yüksekliği 2000 pixel olan bir dikdörtgeni tek sayfada bastıramayacağımız için ilk sayfaya sığmayan diğer bölümleri parçalayarak her bir sayfaya sığacak şekilde ayarlamalıyız. Bu işlem için `PrintPageEventArgs` sınıfının `HasMorePages` değişkenini kullanacağız.

Hemen diğer bölümlere geçmeden önce birden fazla sayfalı yazıcı çıktısı alma işlemine örnek verelim. Bu örnekte bir text dosyasının içeriğini yazıcıya nasıl gönderebileceğimizi inceleyeceğiz. Tabi burda yazının birden fazla sayfada olup olmadığının kontrolünü yapmamız gerekir. Yazıları yazıcı çıktısına göndermek için `Graphics` sınıfının `DrawString` metodunu kullanacağız. Bu metot grafik arayüzüne belirli bir fontta ve font büyüklüğünde yazı yazmamızı sağlar. Önce örneği inceleyelim ardından örnek üzerinde biraz konuşacağız.

```
using System;
using System.IO;
using System.Drawing;
using System.Windows.Forms;
using System.Drawing.Printing;

class Printer
{
    private static StreamReader dosyaAkimi;

    static void Main(string[] args)
    {
        dosyaAkimi = new System.IO.StreamReader("C:\\Print.txt");

        PrintDocument PD = new PrintDocument();
        PD.PrintPage += new PrintPageEventHandler(OnPrintDocument);

        try
        {
            PD.Print();
        }
        catch
        {
            Console.WriteLine("Yazıcı çıktısı alınamıyor...");
        }
        finally
        {
            PD.Dispose();
        }
    }

    public static void OnPrintDocument(object sender, PrintPageEventArgs e)
    {
        Font font = new Font("Verdana", 11) ;
        float yPozisyon = 0 ; int LineCount = 0 ;
        float leftMargin = e.MarginBounds.Left;
        float topMargin = e.MarginBounds.Top;

        string line=null;

        float SayfaBasinaDusenSatir = e.MarginBounds.Height / font.GetHeight() ;
```

```

while (((line=dosyaAkimi.ReadLine()) != null) && LineCount <
SayfaBasinaDusenSatir)
{
    yPozisyon = topMargin + (LineCount * font.GetHeight(e.Graphics));
    e.Graphics.DrawString (line, font, Brushes.Red, leftMargin,yPozisyon);

    LineCount++;
}

if (line == null)
    e.HasMorePages = false ;
else
    e.HasMorePages = true ;
}
}

```

Yukarıdaki program herhangi bir text formatındaki dosyayı yazıcıya göndererek çıktı almanızı sağlayacaktır. Dosyanın içeriğini yazıcıya gönderirken çıktının ne şekilde olacağı tamamen programlama yolu ile bizim tarafımızdan yapılmaktadır. Örneğin çıktının yazı fontunu GDI+ kütüphanesinin bir sınıfı olan Font ile yazı renginide yine GDI+ kütüphanesinin Brushes sınıfının üye elemanları ile rahatlıkla değiştirebiliriz.

Yukarıdaki örnek uygulamada en önemli kısım dosya içeriğinin yazıcıyı gönderilmesi sırasında görülür. Dosya içeriğinin birden fazla sayıda sayfa içermesi durumunda dosya akımından bir sayfaya sığacak kadar satır okunmalıdır. Eğer dosya akımının sonuna gelinmediyse **HasMorePages** özelliği true yapılarak OnPrintDocument metodunun yeniden çağırılması gerekir. Kaynak koddanda gördüğünüz üzere dosya akımından okunan satır null değere eşit olduğunda yani dosyanın sonuna gelindiğinde HasMorePages özelliği false yapılarak Print() metodunun icrası sonlandırılmıştır.

Bir diğer önemli nokta ise yazıcıya gönderilecek her bir sayfada kaç satırın bulunacağını belirlenmesidir. Sayfa başına düşen satır sayısı, sayfanın yazıcıya gönderilecek bölümünün yüksekliğinin yani **e.MarginBounds.Height** 'in çıktıya gönderilecek yazıya ait fontun yüksekliğine bölümü ile elde edilir. Sayfa başına düşen satır sayısı elde edildikten sonra herbir sayfanın içeriği while döngüsü yardımı ile hazırlanır. Okunan satır sayısı **null** değere eşit olmayana kadar ve okunan satır sayısı sayfa başına düşen satır sayısı olana kadar döngüye devam edilir. Döngü içerisinde **PrintPageEventArgs** olay argümanlarını içeren sınıfın Graphics nesnesine **DrawString()** metodu yardımıyla dosya akımından okunan satır yazılır. Bir sonraki satırın çıktı ekranının neresinden başlayacağını tutmak için ise her bir satır okunduğunda yPozisyon'u kullanılan font'un yüksekliği kadar artırılır. Bütün bu işlemleri yaptıktan sonra HasMorePages özelliği ayarlanır ki sonraki sayfalar çıktıya gönderilsin. Eğer dosya sonuna gelinmişse artık basılacak sayfa yok demektir ve **HasMorePages** özelliği false olarak belirlenir.

Not : Dosya akımının neden OnPrintDocument() metodunun içinde tanımlanmadığını merak ediyordunuz. Bunun sebebi OnPrintDocument() metodunun her bir sayfa için yeniden çağırılmasıdır. Eğer dosya akımını bahsi geçen metotta tanımlamış olsaydık her defasında dosya akımı baştan okunacağı için hiç bir zaman dosya akımının sonuna gelemeyecektik ve her defasında sonsuza dek ilk sayfayı çıktıya göndermiş olacaktık. Bu yüzden dosya akımını global düzey diyebileceğimiz bir noktada yani ana sınıfının bir üye elemanı olacak şekilde tanımladık.

Aklınıza takılmış olabilecek diğer bir nokta ise yazıcının renk ayarlarıdır. Eğer yazıcınız renkli çıktı almayı desteklemiyorsa DrawString() metoduna parametre olarak geçtiğimiz **Brushes.Red** parametresinin bir önemi olmayacaktır. Bu yüzden dökümanları yazıcıya göndermeden yazıcının renkli baskıyı destekleyip desteklemediğini kontrol etmek en akıllıca

yöntemdir. Bu şekildeki bir kontrol için PrintDocument sınıfının **PrinterSettings** özelliği kullanılabilir. Bu özellik varsayılan yazıcınız ile ilgili bir takım ayarları yapısında barındıran özelliklere sahiptir. Örneğin varsayılan yazıcınızın renkli baskıyı destekleyip desteklemediğini kontrol etmek için **SupportsColor** özelliğini aşağıdaki gibi kullanabilirsiniz. Not : SupportsColor özelliği bool türündendir.

```
using System;
using System.Drawing.Printing;
using System.Drawing;

class Printer
{
    static void Main()
    {
        PrintDocument PD = new PrintDocument();
        PD.PrintPage += new PrintPageEventHandler(OnPrintDocument);

        if ( PD.PrinterSettings.SupportsColor )
        {
            //renkli baskı ayarları
        }
        else
        {
            //renksiz baskı ayarları
        }
    }

    private static void OnPrintDocument(object sender, PrintPageEventArgs e)
    {
        ....
    }
}
```

PrinterSettings yolu ile elde edebileceğimiz diğer önemli özellikler aşağıda listelenmiştir.

CanDuplex : bool türünden olan bu değişken yazıcının arkalı önlü çıktı almayı destekleyip desteklemediğini belirtir.

Copies: short türünden olan bu değişken yazıcıya gönderilecek dökümanın kaç kopya çıkarılacağını belirtir. Eğer 10 sayfalık bir döküman için bu özelliği 5 olarak girerseniz 50 adet kağıdınızı yazıcıya yerleştirmeyi unutmayın.

CanDuplex : bool türünden olan bu değişken yazıcının arkalı önlü çıktı almayı destekleyip desteklemediğini belirtir.

Duplex : Duplex enum sabiti türünden olan bu değişken arkalı önlü baskı özelliğini belirler. Duplex numaralandırması Default,Sizmplex,Horizontal ve Vertical olmak üzere dört tane üyesi vardır.

IsDefaultPrinter : PrinterName ile belirtilen yazıcının bilgisayarınızdaki varsayılan yazıcı(default printer) olup olmadığını belirtir.

IsValid : PrinterName ile belirtilenin gerçekten sisteminize ait bir yazıcı olup olmadığını belirtir.

PaperSizes : Yazıcı tarafından desteklenen sayfa ebatlarının **PaperSizeCollection** türünden bir koleksiyon nesnesi ile geri döner. Bu koleksiyondaki her bir elemanın **System.Drawing** isim alanında bulunan **PaperSize** türündendir. **PaperSize** sınıfının **Width**(sayfa eni), **Height**(sayfa boyu),**Kind**(sayfa türü) gibi özellikleri bulunmaktadır.

PaperSources : Yazıcı tarafından desteklenen sayfa kağıt alma kaynaklarını **PaperSourceCollection** türünden bir koleksiyon nesnesi ile geri döner. Bu koleksiyondaki her bir elemanın **System.Drawing** isim alanında bulunan **PaperSource** türündendir. **PaperSource** sınıfının **Kind** özelliği **PaperSourceKind** numaralandırması türünden bir nesne olup kağıt kaynağının tipini belirtir. Bu numaralandırmanın bazı sembolleri şunlardır : **Envelope**, **Cassette**, **Custom**, **Manuel**, **TractorFeed**.

PrintToFile : Çıktının herhangi port yerine bir dosyaya yazdırılıp yazdırılmayacağını tutan **bool** türünden bir değişken. Bu değişken daha çok birazdan göreceğimiz **PrintDialog** ekranının görüntülenmesi sırasında değiştirilip kullanılır.

Çıktı Ön-İzleme Penceresi

Profesyonel uygulamaların tamamında yazıcıya çıktı göndermeden önce kullanıcıya ön izleme imkanı sağlanır. .NET ortamında program geliştiriyorsanız Windows'un standart ön izleme penceresini programlama yolu ile görüntülemeniz son derece kolaydır. Bu ekranın görüntülenmesi için **System.Drawing** isim alanında bulunan **PrintPreviewDialog** sınıfı kullanılır. Bu sınıf ile ilişkilendirilmiş **PrintDocument** nesnesinin **PrintPage** olayına ilişkin metod çalıştırılarak ön-izleme penceresindeki içerik elde edilir.

Bir **PrintPreviewDialog** nesnesi oluşturulduktan sonra nesnenin **Document** özelliğine **PrintDocument** türünden bir nesne atanır. Ve ardından **PrintPreviewControl** türünden olan nesne üzerinden **Show()** yada **ShowDialog()** metotları kullanılarak ön izleme ekranı gösterilir.

Ön izleme çıktısının görüntülendiği pencereyi elbette **PrintDocument** sınıfının **Print()** metodunu çağırmadan önce göstermeliyiz. Daha önce yaptığımız ve dosya içeriğini yazıcıya gönderen uygulamanın **Main()** metodunu aşağıdaki gibi değiştirerek ön izleme ekranından çıktıya gönderilecek içeriği görüntüleyelim.

```
static void Main(string[] args)
{
    dosyaAkimi = new System.IO.StreamReader("C:\\Print.txt");

    PrintDocument PD = new PrintDocument();
    PD.PrintPage += new PrintPageEventHandler(OnPrintDocument);

    PrintPreviewDialog pdlg = new PrintPreviewDialog();
    pdlg.Document = PD;
    pdlg.ShowDialog();

    try
    {
        PD.Print();
    }
    catch
    {
        Console.WriteLine("Yazici çiktisi alinamiyor...");
    }
    finally
```

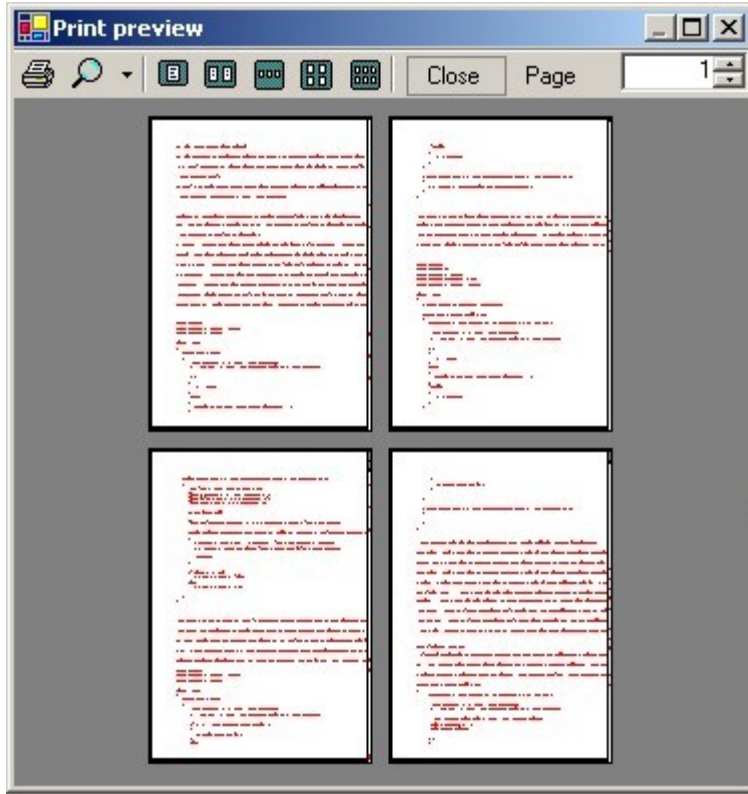


```

    {
        PD.Dispose();
    }
}

```

Programı yeni haliyle derleyip çalıştırdığımızda ilk önce öikti ön izleme ekranı aşağıdaki gibi gösterilecektir. Not : Çıktıya gönderilecek dosyanın yolu örneğimiz için "C:\Print.txt" şeklinde olmalıdır.



Sayfa Düzenleme Ekranı (PageSetupDialog Sınıfı)

Dökümanı çıktıya göndermeden önce gönderme işleminin hangi yazıcı ayarları ile yapılacağını belirlemek için genellikle sayfa düzenleme ekranı gösterilir. Bu ekranda kağıt tipinden, yazıcının kağıt kaynağına kadar bir çok özelliği değiştirmeniz mümkündür. Bu ekranda yapılan bütün değişiklikler PrintDocument sınıfının PrinterSettings özelliğine aktarılır. Sayfa düzenleme ekrana System.Drawing isim alanında bulunan **PrintSetupDialog** sınıfı ile gerçekleştirilir. Bu sınıfın kullanımı **PrintPreviewDialog** sınıfının kullanımı ile neredeyse aynıdır. Bu yüzden ayrıca açıklamaya gerek duymuyorum.

Son olarak yazıcı ön izleme ekranından önce sayfa düzenleme ekranının gösterilmesini sağlamak için uygulamamızın Main() metodunu aşağıdaki gibi değiştirin ve çalıştırın.

```

static void Main(string[] args)
{
    dosyaAkimi = new System.IO.StreamReader("C:\\Print.txt");

    PrintDocument PD = new PrintDocument();
    PD.PrintPage += new PrintPageEventHandler(OnPrintDocument);
}

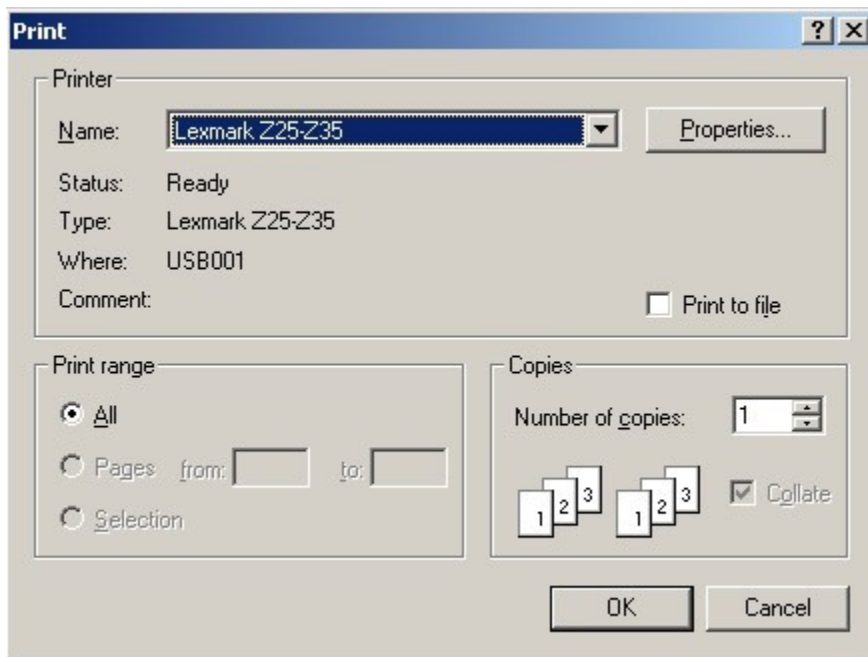
```

```
PrintDialog pdialog = new PrintDialog();  
pdialog.Document = PD;  
pdialog.ShowDialog();
```

```
PrintPreviewDialog pdlg = new PrintPreviewDialog();  
pdlg.Document = PD;  
pdlg.ShowDialog();
```

```
try  
{  
    PD.Print();  
}  
catch  
{  
    Console.WriteLine("Yazici çıktısı alınamıyor...");  
}  
finally  
{  
    PD.Dispose();  
}  
}
```

Programı derleyip çalıştırdığınızda karşınıza ilk çıkacak görüntü aşağıdaki ekran olacaktır.



Bu örnekle birlikte yazıcı çıktısı alma ile ilgili temel işlemlerin anlatıldığı yazımızın sonuna geldik. .NET teki yazıcı çıktısı alma işlemleri bu anlattıklarım ile sınırlı değildir. Ancak bu yazıda anlatılanlar bu konuya çok hızlı bir giriş yapmanızı sağlamıştır. İlerleyen yazılarda görüşmek dileğiyle.

Kaynaklar :

MSDN Yardım Dökümanları

Düzenli İfadeler(Regular Expressions) Nedir?

Regular expression bir metni düzenlemek yada metin içerisinde belli kurallara uyan alt metinler elde etmek için kullandığımız bir dildir. Bir regular expression, string tipindeki karakter topluluğuna uygulanır. Sonuç olarak substringler oluşur yada orjinal metnin bir kısmını içeren değiştirilmiş yeni metinler elde edilir.

Regular Expression'larda Kullanılan Özel Karakterler ve Etkileri

Regular expression desenleri tanımlamada kullanılan özel karakterleri örnekleri ile anlatırsak sanırım regular expressionlar daha tanıdı ve kolay gelebilir.

1.) "." Karakteri

Tek bir karakteri temsil eder(yeni satır karakteri hariç).

"CSharp.edir" şeklindeki bir desen CSharpnedir, CSharpNedir, CSharpSedir, CSharp3edir gibi stringleri döndürebilir.

2.) "[" Karakterleri

Bir arrayi yada aralığı temsil eder.

"CSharp[SNY]edir" deseni, CSharpSedir, CSharpNedir ve CSharpYedir stringlerini döndürür.

"CSharp[a-z]edir" şeklindeki kullanım aralık belirtmeye yarar.

"CSharp[0-9]edir" şeklindeki kullanım ise sayısal aralık belirtmeye yarar.

3.) "?" Karakteri

Kendinden önceki karakterin stringte olması yada olmamasını sağlar.

"CSharpn?edir" deseni CSharpedir yada CSharpnedir döndürür.

4.) "\" Karakteri

Kendinden sonraki özel karakterin stringe dahil edilmesini sağlar.

"CSharpnedir\" deseni CSharpnedir? Stringini döndürür. (Eğer "\" karakterini kullanmamış olsaydık CSharpnedi yada CSharpnedir dönerdi.)

5.) "*" Karakteri

Kendinden önceki karakterin yada stringin hiç olmaması yada istediği sayıda olmasını sağlar.

"CSharpnedir*" deseni, CSharpnedi, CSharpnedir, CSharpnedirr, CSharpnedirrr, ... döndürür. "CSharp(nedir)*" deseni ise CSharp, CSharpnedir, CSharpnedirnedir, ... döndürür.

6.) "{" Karakterleri

Kendinden önce gelen karakterin belirtilen sayıda tekrar etmesini sağlar.

"C{4}Sharpnedir" deseni, CCCCSharpnedir stringini döndürür.

7.) "^" Karakteri

Satır başını ifade eder.

"^CSharpnedir" deseni, satır başında "CSharpnedir" stringi varsa bunu döndürür.

8.) "\$" Karakteri

Satır sonunu ifade eder.

"CSharpnedir\$" deseni, satır sonunda "CSharpnedir" stringi varsa bunu döndürür.

Basit Bir Tarih Deseni Yapalım

Şimdi işin pratiğine geelim ve adım adım tarih deseni oluşturalım. Daha sonra ise oluşturduğumuz bu tarih desenini bir konsol programında kullanalım.

Tarih desenimiz bir string içerisindeki, GG/AA/YYYY formatlarındaki tarihleri yakalayacak yapıda olsun.

Önce desenimizin GG yani tarihin gün belirtilen kısmını tanımlayalım :

```
"(0?[1-9])"  
// 1, 2, .., 9, 01, 02, ..., 09 gibi yazılmış günleri tanımlar.  
  
"([12][0-9])"  
// 10, 11, ..., 29 gibi yazılmış günleri tanımlar.  
  
"(3[01])"  
// 30, 31 günlerini tanımlar.  
  
// Bu üç tanımlı OR (|) işlemiyle  
// birleştirirsek gün tanımını elde ederiz.  
  
// Gün tanımı :  
"((0?[1-9])|([12][0-9])|(3[01]))"
```

Şimdi desenimizin AA yani tarihin ay belirtilen kısmını tanımlayalım :

```
"(0?[1-9])"  
// 1, 2, .., 9, 01, 02, ..., 09 gibi yazılmış ayları tanımlar.  
  
"(1[0-2])"  
// 10, 11, 12 aylarını tanımlar.  
  
// Bu iki tanımlı OR işlemiyle  
// birleştirirsek ay tanımını elde ederiz.  
  
// Ay Tanımı :  
"((0?[1-9])|(1[0-2]))"
```

Şimdi desenimizin YYYY yani tarihin yıl belirtilen kısmını tanımlayalım :

```
"([12][0-9][0-9][0-9])"  
//1000 ile 2999 yılları arasındaki tüm yılları içerir.
```

Ve son olarak tanımladığımız gün, ay ve yıl desenlerini "/" ile birleştirelim :

```
"((0?[1-9])|([12][0-9])|(3[01]))(/)(0?[1-9]|1[0-2])(/)([12][0-9][0-9][0-9])"
```

Basit Bir Test Programı Yazalım

Şimdi elde ettiğimiz tarih desenini test edebileceğimiz basit bir konsol programı yazalım. Ek bilgi olarak “?” şeklindeki bir ifadeyi desenin önüne ilave ederseniz, bir desen grubu ifade etmiş olursunuz ve birkaç deseni aynı anda kontrol edebilirsiniz.

```
using System;

using System.Text.RegularExpressions;

class Test
{
    public static void Main()
    {
        // Regular Expression için bir desen (pattern) tanımlıyoruz :

        string tarihDeseni=@"(?((0?[1-9])|([12][0-9])|(3[01]))(\/)(0?[1-9]|1[0-2])(\/)([12][0-9][0-9][0-9]))";

        // Regular Expression'umuzu tanımlıyoruz :

        Regex benimRegex=new Regex(tarihDeseni);

        // Kullanıcıdan tarih içeren metni talep ediyoruz :

        Console.WriteLine("Lütfen içinde tarih olan bir metin giriniz :");

        // Tarih arayacağımız metni konsoldan alıyoruz :

        string metin=Console.ReadLine();

        // Metin içerisindeki tarihleri (birden fazla olabilir) Collection nesnesine atıyoruz :

        MatchCollection benimMatchCollection=benimRegex.Matches(metin);

        // Metin içindeki her bir tarihi ekrana yazdırıyoruz :

        foreach(Match benimMatch in benimMatchCollection)
        {
            Console.WriteLine(benimMatch.Groups["tarih"]);
        }

        Console.Read();
    }
}
```

Bir Web Sitesindeki E-Mail Adreslerini Yakalamak (Düzenli İfadeler 2)

Gene şu sıkıcı düzenli ifadeler değil mi? Bu makalede düzenli ifadelerin ne kadar etkileyici olduğunu bir örnek üzerinde anlatmaya çalışacağım. Web sitelerinin birebir kopyasını kendi bilgisayarınıza kopyalayan (Teleport gibi) programları bilirsiniz. Önce bir sayfanın kaynak kodunu indirir. İçindeki linkleri ve resim dosyalarını belirler. Sonra sıra bu link ve resim dosyalarına gelir.

Hiç elinizdeki kaynak kodun linklerini nasıl ayıklayacağınızı düşündünüz mü? İşte düzenli ifadeler burada gerçekten harikalar yaratır. Fazla uzatmadan örneğimize geçelim...

E-Mail Yakalayıcı



Şimdi yapacağımız örnekte önce bir web sitesinin kaynak kodunu indirip, içerisindeki e-mail linklerini elde edeceğiz.

Öncelikle sayfanın kaynak kodunu indirelim

Bunun için iki basit metod yazdım. İlki sayfanın adresini kullanıcıdan alıyor, ikincisi sayfaya bağlanıp kaynak kodu indiriyor :

```
// Adresi Alan Metod :
private string AdresiAl()
{
    string adres="http://" + txtAdres.Text;
    return adres;
}

// Sayfanın Kaynak Kodunu Döndüren Metod :
private string KaynakAl(string adres)
{

```

```

lblStat.Text="Siteye Bağlanıyor...";
WebResponse benimResponse=null;
try
{
    WebRequest benimWebRequest=WebRequest.Create(adres);
    benimResponse=benimWebRequest.GetResponse();
}
// Eğer internet bağlantısı yoksa yada site adresi yanlış ise :
catch(WebException e)
{
    lblStat.Text="Siteye Bağlanamıyor.";
    return null;
}
// Site içeriği stream olarak alınıyor :
Stream str=benimResponse.GetResponseStream();
StreamReader reader=new StreamReader(str);
string kaynak=reader.ReadToEnd();
// Tüm içerik küçük harfle döndürülüyor.
//Daha fazla kontrol yapmamak için bir önlem
return kaynak.ToLower();
}

```

Sayfanın kaynak kodunu indirdikten sonra içindeki e-mail linklerini bulup bir diziye atan bir başka fonksiyon daha yazdım. Öncelikler e-mail linkini yakalayan deseni (pattern) açıklamaya çalışalım.

Bildiğiniz gibi e-mail linkleri
[](mailto:aaa@bbb.ccc) gibi ifade edilir.

O zaman desenimiz (href=) ifadesi ile başlamalıdır.
 Ardından (') yada (") karakterleri gelebilir.

```
"(href=)(('|")|("'))"
```

Sonra "mailto:" ifadesi gelir :

```
"(href=)(('|")|("'))(mailto:)"
```

"mailto:" ifadesinden sonra istediğimiz ifade yani e-mail adresi gelir. Bunu "mail" isminde bir grup tanımlayarak elde edeceğiz.

```
"(href=)(('|")|("'))(mailto:)(?<mail>(.*))"
// (.*) ifadesi kendinden sonra gelen desene kadar her karakteri alan bir desendir.
```

Şimdi desenimizi sonlandıralım :

```
"(href=)(('|")|("'))(mailto:)(?<mail>(.*))(('|")|("'))"
```

Kısaca, "mailto:" ile tırnak karakterleri arasındaki her ifade bizim için mail grubuna dahil oldu.

Şimdi E-mail adreslerini dizi şeklinde döndüren metodumuzu yazalım :

```
// Sayfanın içindeki mail adreslerini dizi şeklinde döndüren metod :
private string[] MailAl(string kaynak)
{
    lblStat.Text+= "Kaynak kod alındı... " + "Mailler ayrıştırılıyor... ";
    // Desenimiz :
    string mailDeseni=@"(href=)('|\"")(mailto:)(?<mail>(.*))('|\"")";
    int i=0;
    // Regular Expressionumuzu tanımlıyoruz :
    Regex benimRegex=new Regex(mailDeseni);
    Match str=benimRegex.Match(kaynak);
    // Oluşturduğumuz deseni sitenin kaynak kodunda karşılaştırıyoruz :
    MatchCollection mailCol=benimRegex.Matches(kaynak);
    string[] mail=new string[mailCol.Count];
    // Bulunan her e-mail adresini mail[] dizisine atıyoruz :
    foreach(Match mailMatch in mailCol)
    {
        mail[i]=mailMatch.Groups["mail"].ToString();
        i++;
    }
    return mail;
}
```

Şimdi Yakala butonuna basılınca icra edilecek olay kodunu yazalım :

```
// Şimdi e-mail yakalamak için bu yazdığımız metodları button_Click olayı ile
// birleştirelim :
private void btnYakala_Click(object sender, System.EventArgs e)
{
    lblStat.Text="";
    if(txtAdres.Text=="")
    {
        MessageBox.Show("Lütfen Bir Adres Girin !");
    }
    else
    {
        // Sitenin Adresini alıyoruz :
        string adres=AdresiAl();
        // Sitenin Kaynak Kodunu alıyoruz :
        string kaynak=KaynakAl(adres);
        // E-Mail adreslerini alıyoruz :
        if(kaynak!=null)
        {
            string[] mail=MailAl(kaynak);
            lblStat.Text+="İşlem sona erdi." + mail.Length + " tane mail adresi
yakalandı.";
            foreach(string yakalananMail in mail)
            {
                // Her e-mail adresi listbox'a giriliyor :
                lblEmail.Items.Add(yakalananMail);
            }
        }
    }
}
```


Açıklama

Bazı sitelerde frameset kullanıldığından sayfada e-mail linki görülse bile programımız bunları döndürememekte. Bu sayfaların framelerinin linkleri verilerek e-mail adresleri elde edilebilir.

Yine bazı sitelerde linkler javascript kodu ile erişildiğinden bu adreslerde programımız tarafından erişilememektedir.

C# ile Taskbarda Çalışan Program Hazırlamak

Bu makalemdede size NotifyIcon ve ContextMenu kullanarak bir taskbara yerleşen program nasıl yapılır, onu göstereceğim. Daha fazla uzatmadan hemen kodlarımızı yazmaya başlayalım.

İlk olarak Visual Studio'yu açalım ve yeni bir proje yaratalım. Bu projenin adına istediğiniz gibi bir isim verebilirsiniz. Projemiz "C# Windows Application" olmalıdır.

Projemizi yarattıktan sonra Add / New Item diyerek yeni bir Icon ekleyelim. Iconumuzun Build Action'ı mutlaka "Embedded Resource" olmalı. Daha sonra Form1'ın kod kısmına gecerim.

Sınıfımızın içine

```
private NotifyIcon notifyicon;  
private ContextMenu menu;
```

kodlarını ekleyelim. Formumuza iki kere tıklayalım ve aşağıdaki metotları kaynak kodumuza ekleyelim.

```
private void Form1_Load(object sender, System.EventArgs e)  
{  
    notifyicon = new NotifyIcon(); //Yeni bir NotifyIcon tanımladık  
  
    notifyicon.Text = "NotifyIcon Ornegimiz"; //Mouse ile uzerine geldiğimizde olusacak  
yazi  
  
    notifyicon.Visible = true; //Gorunur ozelligi  
  
    notifyicon.Icon = new Icon("Icon1.ico"); //Iconumuzu belirledik  
  
    menu = new ContextMenu(); //Yeni bir ContextMenu tanımladık  
  
    menu.MenuItems.Add(0, new MenuItem("Goster", new  
System.EventHandler(Goster_Click))); //Menuye eklemeler yapıyoruz.  
  
    menu.MenuItems.Add(1, new MenuItem("Gizle", new  
System.EventHandler(Gizle_Click)));  
  
    menu.MenuItems.Add(2, new MenuItem("Kapat", new  
System.EventHandler(Kapat_Click)));  
  
    notifyicon.ContextMenu = menu; //Menumuz notifyiconun menu su olarak tanımladık  
}  
  
protected void Goster_Click(object sender, System.EventArgs e)  
{  
    Show(); //Formumuzu normal ebatlara getirecek  
}  
  
protected void Gizle_Click(object sender, System.EventArgs e)  
{  
    Hide(); // Formumuzu minimize edecek
```

```

}

protected void Kapat_Click(object sender, System.EventArgs e)
{
    Close(); //Formumuzu kapatacak
}

```

Evet, şimdi programımızı çalıştırmaya hazırız. E o zaman çalıştıralım ve sonucu görelim. Gordüğünüz gibi programımız çalıştı. Programı kapatalım. O da ne! Iconumuz hala taskbarda duruyor. Peki bunu nasıl düzelteceğiz? Hemen cevap verelim. Kaynak kodumuzun biraz üstlerine bakıyoruz ve şu satırları görüyoruz:

```

protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        if(components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose( disposing );
}

```

Bu satırları aşağıdaki gibi değiştirdiğimizde program kapatıldığında taskbar daki icon da silinecektir.

```

protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        this.notifyicon.Dispose();
        components.Dispose();
    }
    base.Dispose( disposing );
}

```

Ve bir sorunumuz daha var. Programımızı açtığımız anda Form1 de gözüküyor. Peki Form1 gözükmeden sadece Iconumuzun gözükmesini nasıl sağlayacağız? Bunun da çözümü var. Biraz daha yukarılara bakıp

```
InitializeComponent();
```

satırından bir sonraki satıra su kodları koyuyoruz:

```

this.WindowState = FormWindowState.Minimized;
this.ShowInTaskbar = false;

```

Bunları yaptıktan sonra yapmamız gerek bir değişiklik daha var. O da Goster_Click ve Gizle_Click'i su şekilde değiştirmek:

```

protected void Goster_Click(object sender, System.EventArgs e)
{
    this.WindowState = FormWindowState.Normal; //Formumuzu normal ebatlara getirecek
}

```

```
}  
  
protected void Gizle_Click(object sender, System.EventArgs e)  
{  
this.WindowState = FormWindowState.Minimized; // Formumuzu minimize edecek  
}
```

Ve şimdi herşey tamam. Programımız artık çalışmaya hazır durumda. Hemen çalıştırıp sonucu görebiliriz.

Umarım herkes için faydalı bir yazı olmuştur. Benim için kod yazmak yazı yazmaktan daha kolay, bunu herkesin tatmasını isterim :)) Yeni yazılarda görüşmek dileğiyle hoşçakalın.

"Singleton" Tasarım Deseninin(Pattern) C# ile Gerçekleştirilmesi

Yazılım mühendisliğinin sanatsal yönü ağır olan "design pattern" kavramını bir çoğumuz mutlaka duymuşuzdur, ama rutin işlerimizden kendimizi boşa çıkarıp bir türlü inceleme fırsatı bulamamışızdır, Bu ve bundan sonraki bir kaç makalelik dizide size kendi terminolojik yapısı içinde deyimleşmiş olan "design pattern" yani "desen tasarımı" kavramını açıklamaya çalışacağım. Elbette açıklamalarımı en çok bilinen tasarım desenleri ile destekleyeceğim. Bu konudaki ilk makalede en basit ve en popüler tasarım desenlerinden biri olan "Singleton" deseninden bahsedip "pattern design" sanatına daha farklı bir bakış açısıyla yaklaşmanızı sağlamaya çalışacağım. O halde işe basit tanımlarla başlayalım.

"Design Pattern" Nedir ?

Bildiğiniz üzere günümüzde yazılım mühendisliği alanında en fazla ses getiren kurgu yazılımın gerçek dünya ile olan ilişkisinin sağlanabilmesidir. Bu ilişki elbette nesne yönelimli programlama tekniği ile sağlanmaktadır. Nesne yönelimli programlama tekniği bilgisayar uygulamalarını soyut bir olgudan çıkararak insanların daha kolay algılayabileceği hale getirmiştir. Öyle görünüyorki, makro düzeyde gerçek hayatı modelleme ile başlayan bu ilişki mikro düzeydede büyük ses getirecektir. Nitekim son yıllarda geliştirilen yapay sinir ağları ile makinelerin çalışma sistemlerinin gün geçtikçe insanların veya canlıların çalışma şekline yaklaştığı görülmektedir. Artık bilgisayarlardan sadece verilen komutları yerine getirmek değil, belirli olaylar ve durumlar karşısında bazı hükümlere varabilmeleride istenmektedir. Burada vurgulamak istediğim nokta şudur : bizler yazılım mühendisleri veya programcılar olarak gerçek hayatı ne kadar iyi modelleyebiliyorsak o kadar başarılı sayılırız. Peki "desgin pattern" konusu nerede devreye girmektedir? İşte benimde gelmek istediğim nokta budur; "desgin patterns" bir modelin tasarımın ustaca tasarlanmasını sağlayacak çeşitli desenlerin oluşturulmasını ve bu desenlerin ihtiyaç dahilinde herhangi bir modelin inşasında kullanılabilmesini sağlar. "Design pattern" kavramı bir kurallar topluluğundan ziyade bir işi nasıl ve en güzel ne şekilde yapabileceğimiz gösteren yöntemler topluluğudur. Öyleki iyi bir yazılım modelleyicisiyseniz kendi tasarım desenlerinizi oluşturabilir ve bunları diğer ustaların kullanımına sunabilirsiniz. Tasarım desenleri tecrübe ile oluşturulan yapılarıdır. Bazıları olmazsa olmaz yapılar olmasına rağmen bazıları tamamen yazılımın sanatsal yönünü göstermek için tasarlanmıştır . Örneğin bu yazımın ana konusunu belirleyen "Singleton" tasarım deseni yıllardan beri bir çok kişi tarafından kullanılmıştır. Sizde bu yazıda bu desenin amacını ve nasıl uygulandığını öğrendiğinizde eminimki projelerinizde mutlaka kullanacaksınız. Hemen şunuda belirtiyimki bu tasarım deseni sizi uzaya götürmeyecektir, bu yüzden beklentilerinizi biraz daha azaltmanızda fayda var.

O halde "design pattern" yada "tasarım deseni" ni şu şekilde tanımlayabiliriz : Bir tasarım problemini en basit ve en efektif bir şekilde çözüme kavuşturacak yöntemdir.

"Design Pattern" Kaynakları

"Design Pattern" konusunda yazılmış en güzel ve en popüler kaynaklardan biri Erich Gamma, Richard Helm, Ralph Johnson ve John Vlissides tarafından kaleme alınmış "*Design Patterns: Elements of Reusable Object-Oriented Software*" kitabıdır. Bu kitapta en popüler tasarım desenleri anlatılmış ve bu desenlerin çeşitli uygulamalarına yer verilmiştir. "Desgin pattern" guru'ları olarak anılan bu 4 kişi "Gangs Of Four(GOF)" olarak ta bilinmektedir. Zaten bahsi geçen kitapta anlatılan tasarım desenlerine de genel olarak GoF tasarım desenleri denilmektedir. Bu yazı ile başlayan yazı dizisinde GOF olarak anılan tasarım desenlerini sizlere aktarıp çeşitli kullanım alanlarını açıklayacağım.

GOF tasarım desenleri genel olarak 3 ana grup altında incelenir. Bu gruplar ve herbir gruptaki tasarım desenlerinin isimleri aşağıda verilmiştir.

1 - Creatinal Patterns Bu desenler bir yada daha fazla nesnenin oluşturulması ve yönetilmesi ile ilgilidir. Örneğin bu yazıda anlatacağım ve bir uygulamanın ömrü boyunca belirli bir nesneden sadece bir adet bulunmasını garantileyen Singleton deseni bu gruba girmektedir. Bu gruptaki diğer desenler ise

- ☐ Abstract Factory
- ☐ Builder
- ☐ Factory Method
- ☐ Prototype

olarak bilinmektedir. Bu desenlerin bir çoğunu ilerleyen yazılarımda ele alacağım. Şimdilik sadece bir giriş yapıyoruz.

2 - Behavioral Patterns Bu gruptaki desenlerin amacı belirli bir işi yerine getirmek için çeşitli sınıfların nasıl birlikte davranabileceğinin belirlenmesidir. Bu gruptaki desenler ise aşağıdaki gibidir.

- ☐ Chain of responsibility
- ☐ Command
- ☐ Interpreter
- ☐ Iterator
- ☐ Mediator
- ☐ Memento
- ☐ Observer
- ☐ State
- ☐ Strategy
- ☐ Template method
- ☐ Visitor

3 - Structural Patterns Bu gruptaki desenler ise çeşitli nesnelerin birbirleri ile olan ilişkileri temel alınarak tasarlanmıştır. Bu gruptaki tasarım desenleri ise şunlardır:

- ☐ Adapter
- ☐ Bridge
- ☐ Composite
- ☐ Decorator
- ☐ Façade
- ☐ Flyweight
- ☐ Proxy

Bu giriş bilgisinden sonra şimdi nesnelerin yaratılması ile ilgili grup olan "Creatinal Patterns" grubunda bulunan "Singleton" desenini açıklamaya başlayabiliriz.

Singleton Deseni

Singleton deseni bir programın yaşam süresince belirli bir nesneden sadece bir örneğinin(instance) olmasını garantiler. Aynı zamanda bu desen, yaratılan tek nesneye ilgili sınıfın dışından global düzeyde mutlaka erişilmesini hedefler. Örneğin bir veritabanı uygulaması geliştirdiğinizi düşünelim. Her programcı mutlaka belli bir anda sadece bir bağlantı nesnesinin olmasını isteyecektir. Böylece her gerektiğinde yeni bir bağlantı nesnesi yaratmaktansa varolan bağlantı nesnesi kullanılarak sistem kaynaklarının daha efektif bir şekilde harcanması sağlanır. Bu örnekleri dahada artırmak mümkündür. Siz ne zaman belli bir anda ilgili sınıfın bir örneğine ihtiyaç duyarsanız bu deseni kullanabilirsiniz.

Peki bu işlemi nasıl yapacağız.? Nasıl olacakta bir sınıftan sadece ve sadece bir nesne

yaratılması garanti altına alınacak? Aslında biraz düşünürseniz cevabını hemen bulabilirsiniz! Çözüm gerçekten de basit : statik üye elemanlarını kullanarak.

Singleton tasarım desenine geçmeden önce sınıflar ve nesneler ile ilgili temel bilgilerimizi hatırlayalım. Hatırlayacağınız üzere bir sınıftan yeni bir nesne oluşturmak için yapıcı metot(constructor) kullanılır. Yapıcı metotlar C# dilinde new anahtar sözcüğü kullanılarak aşağıdaki gibi çağrılabilir.

Sınıf nesne = new Sınıf();

Bu şekilde yeni bir nesne oluşturmak için new anahtar sözcüğünün temsil ettiği yapıcı metoduna dışarıdan erişimin olması gerekir. Yani yapıcı metodun public olarak bildirilmiş olması gerekir. Ancak "Singleton" desenine göre belirli bir anda sadece bir nesne olabileceği için new anahtar sözcüğünün ilgili sınıf için yasaklanması gerekir yani yapıcı metodun protected yada private olarak bildirilmesi gerekir. Eğer bir metodun varsayılan yapıcı metodu(default constructor- parametresiz yapıcı metot) public olarak bildirilmemişse ilgili sınıf türünden herhangi bir nesnenin sınıfın dışında tanımlanması mümkün değildir. Ancak bizim isteğimiz yalnızca bir nesnenin yaratılması olduğuna göre ilgili sınıfın içinde bir yerde nesnenin oluşturulması gerekir. Bunu elbette statik bir özellik(property) yada statik bir metotla yapacağız. Bu statik metot sınıfın kendi içinde yaratılan nesneyi geri dönüş değeri olarak bize gönderecektir. Peki bu nesne nerde ve ne zaman yaratılacaktır? Bu nesne statik metodun yada özelliğin içinde yaratılıp yine sınıfın private olan elemanına atanır. Tekil olarak yaratılan bu nesne her istendiğinde eğer nesne zaten yaratılmışsa bu private olan elemanın referasına geri dönmek yada nesneyi yaratıp bu private değişkene atamak gerekmektedir. Sanırım bu deseni nasıl uygulayabileceğimizi kafanızda biraz canlandırdınız. O halde daha fazla uzatmadan desenimizi uygulamaya geçirelim.

Singleton Deseninin 1. versiyonu

```
public class SingletonDeseni
{
    private static SingletonDeseni nesne = new SingletonDeseni();

    private SingletonDeseni()
    {
    }

    public static SingletonDeseni Nesne
    {
        get
        {
            return nesne;
        }
    }
}
```

Yukarıdaki sınıf örneğinde SingletonDeseni sınıfı belleğe yüklendiği anda statik olan SingletonDeseni nesnesi yaratılacaktır. Bu nesne yaratılışının new anahtar sözcüğü ile yapıldığına dikkat edin. Eğer siz Main() gibi bir metodun içinden bu nesneyi yaratmaya kalksaydınız derleme aşamasında hata alırdınız. Çünkü public olan herhangi bir yapıcı metot bulunmamaktadır. Ayrıca Siz Main() gibi bir metodun içinden yaratılan bu nesneye

SingletonDeseni nesne = SingletonDeseni.Nesne;

şeklinde erişmeniz mümkündür. Böylece yukarıdaki deyim her kullandığınızda size geri dönen nesne, sınıfın belleğe ilk yüklendiğinde yaratılan nesne olduğu garanti altına alınmış oldu. Dikkat etmeniz gereken diğer bir nokta ise nesneyi geri döndüren özelliğin yalnızca get bloğunun olmasıdır. Böylece bir kez yaratılan nesne harici bir kaynak tarafından hiç bir şekilde değiştirilemeyecektir.

Yukarıdaki SingletonDeseni sınıfını aşağıdaki gibi de yazmamız mümkündür.

Singleton Desenin 2. versiyonu

```
public class SingletonDeseni
{
    private static SingletonDeseni nesne = new Singleton();

    private SingletonDeseni()
    {

    }

    public static Singleton Nesne()
    {
        return nesne;
    }
}
```

Dikkat ederseniz iki sınıfın tek farkı oluşturulan nesneye erişme biçimidir. İlk versiyonda nesneye özellik üzerinden erişilirken ikinci versiyonda metot üzerinden erişilmektedir. Değişmeyen tek nokta ise her iki erişim aracının da statik olmasıdır.

Yukarıdaki her iki versiyonda da biz yaratılan nesneyi

SingletonDeseni nesne = SingletonDeseni.Nesne;

yada

SingletonDeseni nesne = SingletonDeseni.Nesne();

şeklinde istediğimizde nesne zaten yaratılmış durumda olmaktadır. Oysa bu sınıfı daha efektif bir hale getirerek yaratılacak nesnenin ancak biz onu istediğimizde yaratılmasını sağlayabiliriz. Bu durumu uygulayan Singleton deseninin 3 versiyonunu olarak aşağıda görebilirsiniz.

Singleton Desenin 3. versiyonu

```
public class SingletonDeseni
{
    private static SingletonDeseni nesne;

    private SingletonDeseni()
    {

    }

    public static Singleton Nesne()
    {
        if(nesne == null)
```



```
        nesne = new SingletonDeseni();  
    }  
    return nesne;  
}  
}
```

Gördüğümüz üzere nesne ilk olarak sınıf belleğe yüklendiğinde değil o nesneyi ilk defa kullanmak istediğimizde yaratılıyor. İlgili nesneyi her istediğimizde yeni bir nesnenin yaratılmaması içinde

```
if(nesne == null)
```

şeklinde bir koşul altında nesnenin yaratıldığına dikkat edin.

Not : 3.versiyonda nesneyi yaratan bir metod olabileceği gibi 1. versiyondaki gibi sadece get bloğu olan özellikte olabilir.

Herşeye rağmen yukarıdaki 3 versiyonda bazı durumlar için tek bir nesnenin oluşmasını garanti etmemiş olabiliriz. Eğer çok kanallı(multi-thread) bir uygulama geliştiriyorsanız farklı kanalların aynı nesneyi tekrar yaratması olasıdır. Ancak eğer çok kanallı çalışmıyorsanız(çoğunlukla tek thread ile çalışırız) yukarıdaki sade ama öz olan 3 versiyondan birini kullanabilirsiniz. Ama eğer çok kanallı programlama modeli söz konusu ise ne yazıkki farklı kanalların aynı nesneden tekrar yaratmasını engellemek için ekstra kontroller yapmanız gerekmektedir. Ne yazıkki diyorum çünkü bu yapacağımız kontrol performansı büyük ölçüde düşürmektedir.

O halde çok kanallı uygulamalarda kullanabileceğimiz Singleton desenini yazalım.

Singleton Deseninin 4. versiyonu

```
public class SingletonDeseni  
{  
    private static SingletonDeseni nesne;  
    private static Object kanalKontrol = new Object;  
    private SingletonDeseni()  
    {  
    }  
    public static Singleton Nesne()  
    {  
        if(nesne == null)  
        {  
            lock(kanalKontrol)  
            {  
                if(nesne == null)  
                {  
                    nesne = new SingletonDeseni();  
                }  
            }  
        }  
        return nesne;  
    }  
}
```

Yukarıdaki desendeki püf nokta lock anahtar sözcüğünün kullanımıdır. Eğer nesne ilk defa yaratılacaksa yani daha önceden nesne null değere sahipse lock anahtar sözcüğü ile işaretlenen blok kitlenerek başka kanalların bu bloğa erişmesi engellenir. Böylece kilitlenme işlemi bittiğinde nesne yaratılmış olacağı için, kilidin kalkmasını bekleyen diğer kanal lock bloğuna girmiş olsa bile bu bloktaki ikinci if kontrolü nesnenin yeniden oluşturulmasını engelleyecektir. Böylece çok kanallı uygulamalar içinde tek bir nesnenin oluşmasını ve bu nesneye erişimi garanti altına alan Singleton desenini tasarlamış olduk.

Son olarak lock anahtar sözcüğünü kullanmadan çok kanallı uygulamalar içinde tek bir nesneyi garanti altına alacak deseni yazalım. Aşağıda Singleton desenin 5. versiyonu bulunmaktadır.

Singleton Deseninin 5. versiyonu

```
public class SingletonDeseni
{
    private static SingletonDeseni nesne = new SingletonDeseni ();

    private static SingletonDeseni()
    {

    }

    private SingletonDeseni()
    {

    }

    public static SingletonDeseni Nesne
    {
        get
        {
            return nesne;
        }
    }
}
```

Bu versiyonun birinci versiyondan tek farkı yapıcı metodunda statik olmasıdır. C# dilinde statik yapıcı metotlar bir uygulama domeninde ancak ve ancak bir nesne yaratıldığında yada statik bir üye eleman referans edildiğinde bir defaya mahsus olmak üzere çalıştırılır. Yani yukarıdaki versiyonda farklı kanalların(thread) birden fazla SingletonDeseni nesnesi yaratması imkansızdır. Çünkü static üye elemanlar ancak ve ancak bir defa çalıştırılır.

Son versiyon basit ve kullanışlı görünmesine rağmen kullanımının bazı sakıncaları vardır. Örneğin Nesne Özelliği dışında herhangi bir statik üye elemanınız var ise ve ilk olarak bu statik üye elemanını kullanıyorsanız siz istemedğiniz halde SingletonDeseni nesnesi yaratılacaktır. Zira yukarıda da dediğimiz gibi bir statik yapıcı metot herhangi bir statik üye elemanı kullanıldığı anda çalıştırılır. Diğer bir sakıncalı durumda birbirini çağıran statik yapıcı metotların çağırılması sırasında çelişkilerin oluşabileceğidir. Örneğin her static yapıcı metot ancak ve ancak bir defa çalıştırılır dedik. Eğer çalıştırılan bir static metot diğer bir statik metodu çağırıyor ve bu statik metotta ilkini çağırıyorsa bir çelişki olacaktır.

Kısacası eğer kodunuzun çelişki yaratmayacağından eminseniz 5. deseni kullanmanız doğru olacaktır. Eğer çok kanallı uygulama geliştiriyorsanız 4. versiyonu, çok kanallı uygulama geliştirmiyorsanızda 3. versiyonu kullanmanız tavsiye edilmektedir.

Singleton deseni konulu makalenin sonuna gelmiş bulunmaktayız. Eğer ileride bir gün yukarıdaki desenlerin birini kullanma ihtiyacı hissederseniz hangi deseni ne amaçla kullandığınızı bizimle paylaşılırsanız seviniriz.

Bir diğler "Creational Patterns" deseni olan "Abstract Factory" desenini anlatacağım yazıda görüşmek üzere.

XP Stilinde Kontroller ile Çalışma

Bu yazımızda Windows Form kontrollerinin veya nesnelerinin Windows XP stili görünümlerini nasıl elde edebileceğimizi göreceğiz.

Microsoft Framework v1.1' de bu özellik henüz pratik bir şekilde yok. Bu yüzden yolumuz biraz uzun.

Elde edeceğimiz bu görünüm Windows Xp' den önceki işletim sisteminde haliyle görünmeyecek, o işletim sisteminin default haliyle görünecektir(mesela butonlar önceki işletim sistemlerinde gri renkli görünüyordu).

Herhangi bir karışıklık çıkmaması için yönergeleri beraber takip edelim.

Hemen işlem adımlarımıza başlayalım:

Microsoft Visual Studio.NET' i açın.

File/New/Project' i tıklayın.

Açılan Pencerede **Project Type** alanında **Visual C# Project** seçili olsun.

Aynı pencerede **Templates** alanında **Windows Application** seçili olsun.

Aynı pencerede **Name** alanına **XPStyle** yazın.

Aynı pencerede **Location** alanında mevcut yolun sonundaki klasör isminide **XPStyle** yapın.

Projenin açılması için **Okey** butonuna tıklayın.

Şimdi **Form1.cs[Design]** görünümüne sahipsiniz.

Formumuza ; **Button, radioButton, checkBox, textBox, progressBar** ve **trackBar** ekleyin.

Button, radioButton, ve checkBox nesnelerinin **Properties** penceresinde **Flat Style** kısmını **System** yapın.

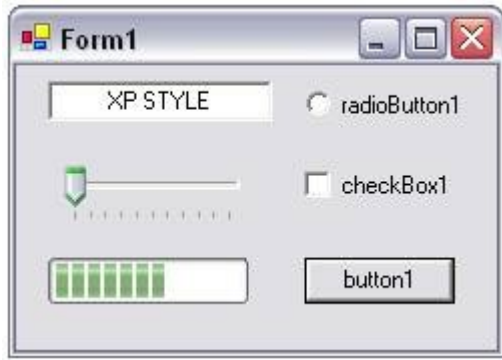
Düğherleri için bunu yapmaya gerek yok.

Menüden File/Save All tıklayın ve Menüden Build/Build Solutin tıklayın.

Elimizde mevcut bir görünüm oluştu, Projeyi bu haliyle çalıştırırsanız(Debug/Start),

Form elemanlarında XP Stilini göremeyeceksiniz,

Mesela Buton hala aşağıdaki gibi gri renkte görünecek.



Şimdi XP Stil görünümünü elde etmek için yönergeleri izleyin:

Menüden, Project/Add Class tıklayın.

Açılan pencerede **Templates** kısmında **XML File**(herhangi bir .cs dosyası da olabilir) seçin.

Aynı pencerede **Name** kısmındaki alanı tamamen temizleyin.

"[Proje Adı].exe.manifest" yazım biçiminde **XPStyle.exe.manifest** yazın.

Bu projenin adını **XPStyle** olarak belirlemiştik

Dosyamızın oluşması için **OK** butonuna tıklayın.

Oluşturduğumuz **XPStyle.exe.manifest** adlı dosyayı açın ve içine aşağıdaki kodları yapıştırın.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">

<assemblyIdentity version="1.0.0.0" processorArchitecture="X86"
name="Microsoft.Winweb.<Executable Name>" type="win32"/>

<description>.NET control deployment tool</description><dependency>

  <dependentAssembly>

    <assemblyIdentity type="win32" name="Microsoft.Windows.Common-Controls"
version="6.0.0.0" processorArchitecture="X86" publicKeyToken="6595b64144ccf1df"
language="*" />

  </dependentAssembly>

</dependency>

</assembly>
```

NOT: "<" ">" karakterlerinin < ve > haline dönüşmesi söz konusu olabilir. Bu yüzden bu kodları önce bir NotePad' e yapıştırın sonra buradan Select All deyip tekrar kopyalayın ve XPStyle.exe.manifest dosyamıza yapıştırın.

Bu kodda <Executable Name> kısmına Projemizin adı olan XPStyle yazın

Yeni hal: name="Microsoft.Winweb.XPStyle" şeklinde olacak.

Menüden File/Save All tıklayın ve Menuden Build/Build Solutin tıklayın.

Microsoft Visual Studio.NET' i indirin ve projenizin bulunduğu klasöre geçin

Bu klasörün içinde XPStyle.exe.manifest dosyasını göreceksiniz.

Bu dosyayı kopyalayıp, Obj klasörünün içine girin, buradan da Debug klasörüne tıklayıp içine girin ve dosyayı buraya yapıştırın.

Çalışma esnasında faydalanmak içi bu dosyayı bin\debug klasörüne de kopyalayabilirsiniz

Microsoft Visual Studio.NET' i açın

Menüden, File/Open/File tıklayın.

Açılan pencerede Obj\Debug klasörüne ulaşın.

Buradan XPStyle çalıştırılabilir dosyanızı seçin ve Open butonuna tıklayın.

Açılan XPStyle.exe dosyasının içindeyken sağ tıklayın.

Açılan menuden Add Resource tıklayın.

Açılan pencereden import butonuna tıklayın.

Açılan pencrenden Files of type alanında All Files seçin.

Görünen dosyalardan XPStyle.exe.manifest dosyasını seçip Open butonuna tıklayın.

Açılan Custom Resource Type penceresinde Resource Type alanına "RT_MANIFEST" yazın ve Okey butonuna tıklayın.

XPStyle.exe(101-Data) dosyası açıldı. Bu dosyadayken Properties penceresinden ID alanının 101 olan değerini 1 yapın

Menüden File/Save All tıklayın ve Menuden Build/Build Solutin tıklayın.

Bu dosyayı kapatın.

Projeyi çalıştırın.


Karşınızda aşağıdaki gibi XP stilli bir pencere göreceksiniz.

Form1

XP STYLE

☐ radioButton1

☐ checkBox1



button1

Sayıları Yazıya Çevirme Örneği

Bu yazıda bir sayının yazıya nasıl çevirebileceğimiz hakkında bir yol göstereceğim, dil olarak C# kullanılacaktır. Öncelikle belirteyim ki programlama ve C# konusunda çok yeniyim. Hemen hemen tüm bildiklerimi bu siteye borçluyum.

Aşağıdaki kodda bulunan **Oku fonksiyonu** kendisine string olarak gönderilen tam sayıyı yazıya çevirmektedir. Kodun çalışma mantığı şöyledir.

oku fonksiyonuna gönderilen string başına "0" eklemek suretiyle önce 15 haneye tamamlanır, sonra yeni string 3 erli kümeler halinde 5 eşit parçaya bölünür ve her bir üçlü küme tek tek **rakam** dizisine yüklenir. Böylece 5 elemanlı rakam dizisinin her bir elemanında 3 karakterli bir string yüklü olur.

1.Aşama

sayımız 32313234 olsun. ilk olarak sayımızın hane sayısını başına 0 eklemek sureti ile 15 e çıkarırız.

Böylece yeni stringimiz 000000032313234 şeklini alır.

2.Aşama

Stringimiz 3 erli kümeler halinde 5 eşit parçaya bölünür.

1. küme : 000
2. küme : 000
3. küme : 032
4. küme : 313
5. küme : 234

3.aşama

her bir küme 5 elemanlı rakam isimli araya yüklenir ve sonuçta

```
rakam[0] = "000"  
rakam[5] = "234"
```

olur

```
rakam[5][0]="2" 5. kümenin yüzler basamağı;  
rakam[5][1]= "3" 5. kümenin onlar basamağı;  
rakam[5][2]= "4" 5. kümenin birler basamağı;
```

olur.

4.aşama

10 elemanlı **yüzler, onlar, birler** string dizileri tanımlanır ve i **çleri doldurulur**.

örn:


```
yuzler.SetValue("ikiyuz",2);
onlar.SetValue("otuz",3);
birler.SetValue("dört",4);
```

yani yuzler[2]+onlar[3]+birler[4] = ikiyüzotuzdört olur.

```
int x =Convert.ToInt16(rakam[5][0].ToString()); yüzler
```

```
int y =Convert.ToInt16(rakam[5][1].ToString()); onlar
```

```
int z =Convert.ToInt16(rakam[5][2].ToString()); birler
```

```
yuzler[x]+onlar[y]+birler[z] = ikiyüzotuzdört
```

bir döngü ile her bir kümeye bu işlemi uygularsanız, 1 ve ikinci kümelerin bütün elemanları sıfır olduğu için sonuçta

```
otuzdört
üçyüzonuç
ikiyüzotuzdört
```

ü elde edersiniz

5.Aşama

hane isimli 5 li array tanımlanır ve elemanları trilyon, milyar, milyon, bin ve sonuncusu da boş olacak şekilde ayarlanır. aynı döngü içerisinde her bir kümenin sonuna eklenir

```
string sonuc = "";
```

```
for(int i = 0 ; i < 5;i++)
```

```
{
    sonuc = sonuc +

    yuzler[Convert.ToInt16(rakam[i][0].ToString())]+

    onlar[Convert.ToInt16(rakam[i][1].ToString())]+

    birler[Convert.ToInt16(rakam[i][2].ToString())]+

    hane[i];
}
```

Burada ayarlanması gereken durum eğer bir kümenin bütün elemanları sıfırsa (yukarıdaki gibi) hanein gözükmemesi gerekir. Yani

000 = yuzler[0]+onlar[0]+birler[0]+hane[0] dersek sonuç trilyon olur bu durumda

```
if(rakam[0].ToString() != "000")
    hane.SetValue("trilyon ",0);
if(rakam[1].ToString() != "000")
    hane.SetValue("milyar ",1);
if(rakam[2].ToString() != "000")
    hane.SetValue("milyon ",2);
if(rakam[3].ToString() != "000")
    hane.SetValue("bin ",3);
```

yani rakam[0] (trilyon kümesi) "000" değilse hane[0] = "trilyon" olsun demeliyiz.

Yalnız bir sorun daha var. eğer sayı 1000 ise fonksiyon bize haklı olarak "birbin" i döndürür. Bir milyar var, Bir Milyon var ama bir bin ve bir yüz yok. Ben bu sorunu BirSorunu isimli fonksiyonla hallettim.

Aşağıdaki kodu inceleyebilirsiniz.

```
using System;

namespace numbereader
{
    public class SayiOkuma
    {
        private string[] yuzler = new string[10];
        private string[] onlar = new string[10];
        private string[] birler = new string[10];
        private string[] hane = new string[5];
        private string[] rakam = new string[5];
        // arrayları tanımlıyoruz

        public SayiOkuma()
        {
            // içlerini dolduruyoruz

            yuzler.SetValue("dokuzyüz",9);
            yuzler.SetValue("sekizyüz",8);
            yuzler.SetValue("yediyüz",7);
            yuzler.SetValue("altıyüz",6);
            yuzler.SetValue("beşyüz",5);
            yuzler.SetValue("dört yüz",4);
            yuzler.SetValue("üç yüz",3);
            yuzler.SetValue("iki yüz",2);
            yuzler.SetValue("yüz",1);
```

```
yuzler.SetValue("",0);

onlar.SetValue("doksan",9);
onlar.SetValue("seksen",8);
onlar.SetValue("yetmiş",7);
onlar.SetValue("altmış",6);
onlar.SetValue("elli",5);
onlar.SetValue("kırk",4);
onlar.SetValue("otuz",3);
onlar.SetValue("yirmi",2);
onlar.SetValue("on",1);
onlar.SetValue("",0);

birler.SetValue("dokuz",9);
birler.SetValue("sekiz",8);
birler.SetValue("yedi",7);
birler.SetValue("altı",6);
birler.SetValue("beş",5);
birler.SetValue("dört",4);
birler.SetValue("üç",3);
birler.SetValue("iki",2);
birler.SetValue("bir",1);
birler.SetValue("",0);

hane.SetValue("",0);
hane.SetValue("",1);
hane.SetValue("",2);
hane.SetValue("",3);
hane.SetValue("",4);
/* ilk olarak bu arrayın elemanlarını boş olarak ayarlıyoruz eğer küme elemanları
000 değilse trilyon,milyar,milyon bin değerleri ile dolduruyoruz
*/

}

public string oku(string sayi)

{

int uzunluk = sayi.Length;
if(uzunluk > 15)

return "Hata girilen değerin uzunluğu en fazla 15 olmalı";
// uzunluk 15 karakterden fazla olmamalı. si

try
{

long k = Convert.ToInt64(sayi);

}

catch(Exception ex)
```

```
{

return ex.Message.ToString();

}

sayi = "0000000000000000"+sayi;
sayi = sayi.Substring(uzunluk,15);

rakam.SetValue(sayi.Substring(0,3),0);
rakam.SetValue(sayi.Substring(3,3),1);
rakam.SetValue(sayi.Substring(6,3),2);
rakam.SetValue(sayi.Substring(9,3),3);
rakam.SetValue(sayi.Substring(12,3),4);

if(rakam[0].ToString()!="000")
    hane.SetValue("trilyon ",0);
if(rakam[1].ToString()!="000")
    hane.SetValue("milyar ",1);
if(rakam[2].ToString()!="000")
    hane.SetValue("milyon ",2);
if(rakam[3].ToString()!="000")
    hane.SetValue("bin ",3);

string sonuc = "";

for(int i = 0 ; i < 5;i++)

{

sonuc = sonuc + yuzler[Convert.ToInt16(rakam[i][0].ToString())]+
birsorunu(onlar[Convert.ToInt16(rakam[i][1].ToString())]+birler[Convert.ToInt16(rakam[i][2].ToString())])
}

return sonuc;

}

privatestring birsorunu (string sorun)

{

string cozum = "";
if (sorun == "birbin ")
cozum = "bin ";

else
    cozum = sorun;

return cozum;

}
```

}

}

}

Herkese kolay gelsin. Bu arada dileyene DLL i gönderebilirim. İyi çalışmalar.

C#'ta Temsilci (Delegate)ve Olay(Event) Kullanımı

Bu yazımda sizlere C#'ta delegate(delege veya elçi) ve event(olay) kavramlarından bahsedeceğim ve elbette bunları kullanmanın yararlarını anlatacağım. Kuşkusuz .Net Framework ile birlikte gelen en büyük yeniliklerden biri delege ve event kullanımıdır. Bu yanı ile .Net Framework diğer uygulama geliştirme ortamlarının bir kısmından daha üstün niteliklere sahip olmaktadır. Şimdi delege nedir biraz ondan bahsedelim. Delege basit anlamda düşünürsek aracılık yapandır. Burada ise delege aracılık yapan metod olarak karşımıza çıkar. Yani başka bir metodun tetiklenmesine yardım eden metottur. Bu sayede metodların tetiklenmesi dinamik olarak gerçekleşir. Olay(event) ise her zaman –belki farkında olmadan– kullandığımız kontroller gibidir. Yalnız delege ile olay arasındaki fark delegelerin "invoke" edilmesi olayların ise "raise" edilmesidir. Daha basit olarak ifade edersek olaylar delegelerin özel bir halidir. Delegeler olmadan olayların bir anlamı yoktur. Aslında bir örnekle bunların nasıl kullanıldığını açıklarsam daha iyi anlaşılacaktır. Şimdi çayın kaynamasını düşünün. Çay kaynadığı anda haberdar olmak istiyoruz ancak çayın başında da beklemek istemiyoruz. Bu yüzden bizi haberdar edecek zil çalma metodunu çayın kaynama metoduna bağlıyoruz. Ve çay kaynadığı anda zil çalma metodu da otomatik olarak tetiklenmiş oluyor. Delege ile olayın implementasyonunu aşağıda göstereceğim ve bunlarla ilgili bir program de koyarak anlaşılmasını kolaylaştıracam. Çünkü bazen programlarla anlatmak istediklerimizi daha iyi anlatırız.

I. Olayın(event) yapılışı: Mouse'ın tıklanması, form üzerindeki bir tuşa basılması, form üzerindeki bir linkin tıklanması vs... bunların hepsi birer olaydır aslında. Şimdi mouse tıklanması ile ilgili bir kodu inceleyelim:

```
private void FareTikla()
{
    // Buraya mouse'un sol tıklanması durumunda yapılması gereken

    // işlemler gelecek.
}

// Burada fareye sol tıklandığı zaman çalışacak olan metodu bağlıyoruz. Bu işlemi de
doğal olarak delegelerle yapıyoruz. Buradaki '+=' operatörüne dikkat ettiniz mi? Bu '+='
operatörü fareye sol tıklanması olayı karşısında raise edilecek eventi bağlamamıza
yarıyor.
```

```
Mouse.MouseClicked += new MouseClickedEventHandler(FareTikla);
```

```
// Şimdi de fare sol tıklandığı zaman çalışacak olan metodu devreden çıkarıyoruz. Bu
sayede fare sol tıklandığı zaman hiç bir metod //devreye girmeyecektir. Yani farenin sol
tıklanmasına herhangi bir tepki verilmeyecektir.
```

```
Mouse.MouseClicked -= new MouseClickedEventHandler(FareTikla);
```

II. Delegenin yapılışı: Delege kullanılacağı zaman tek başına da kullanılabilirdiği gibi olaylara bağlanarak da kullanılabilir. Bu yüzden ilk önce olayı(event) anlarsak delegenin anlaşılması daha da kolaylaşacaktır. Şimdi basit bir delege nasıl tanımlanır ona bakalım.

```
// Tanımladığımız delege double tipinde parametre alıyor
```

```
// ve int tipinde değer döndürüyor.

public delegate int myDelegate(double D);

// Delegenin temsil edeceği metodu yazalım.

public int ReturnInt(double D)

{

//Buraya metod çalışınca yapılması gereken kod gelecek

}

// Şimdi de delegenin bir örneğini yaratalım.

public void amethod(

{

myDelegate aDelegate = new myDelegate(ReturnInt);

}
```

//Şimdi de delegenin temsil ettiği metodun nasıl çalıştırılacağını sağlayan kodu yazalım.

```
aDelegate(3333);
```

Şimdi bu delege ve olay yapısı bizim ne işimize yarayacak dersenez; Şöyle bir senaryo üretelim: Birden fazla formda aynı anda çalıştığımızı ve bu formların da birbirinin aynı olduğunu düşünelim(şekilsel olarak tabii ki) ve bu formlardan herhangi birinin üzerindeki bir tuşa basıldığı zaman bütün formlarda hangi formun tıklandığına dair bir yazı yazmasını isteyelim.

Bunu eğer uzun yoldan yapmak isterseniz tek tek bütün formlar içinde birer tane değişken tanımlayıp onu kontrol etmeniz gerekirdi. Sonra da tek tek diğer formlara mesaj gönderilmesini sağlamak gerekirdi. Bu şekilde yapmak uzun ve zahmetli bir iş olurdu.

İkinci yöntem ise delege(temsilci) yapısını kullanmaktır. Şimdi her formumuz için bir tane forma mesaj yazacak olan bir metod olmalı ve bu metod hangi formdaki yazdırma tuşunun tetiklendiğini belirtmek için bir tane de parametre almalı. Şimdi de yukarıda anlattığımız senaryonun kodunu inceleyelim.

Ana formumuz içinde tanımladığımız delege ve olay yapısına bakalım. Burada delegemiz tamsayı(int) bir parametre alıyor. Bu parametre sayesinde hangi formdaki tuşa basılarak bu delegenin harekete geçirildiğini anlayabileceğiz. Olayımıza bakacak olursak olayımızın bir üst satırda tanımlanan delege tipinde olduğunu görebiliriz.

sayac değişkeni ise forma numara vermek için tanımlanmış bir değişken oluyor.

```
private int sayac = 0;

public delegate void userDelegate(int formNo);

public event userDelegate userEvent;
```

Yeni form oluşturma tuşuna basılınca ise yeni form oluşturulacak ve bu formun içindeki etikete yazma metodu olaya eklenecek.

```
sayac = sayac+1;
```

```
FrmUser frmUser=new FrmUser(sayac,this);
```

```
this.userEvent += new userDelegate(frmUser.writeFormNo);
```

Olaylara doğrudan başka sınıflar altından erişilebilse de bu yöntem pek doğru bir yol olmayacağı için olayımızı bir metodun içinde kamufle ederek kullanıyoruz. O yüzden olayımızı metodTetikle adında bir metodun içinde kamufle ederek kullanıyoruz. Başka bir formdan içinden bu metod çağrıldığı an otomatik olarak olayımızı tetikleyip bütün formlara(ana formumuz hariç) hangi formdan tuşa basıldığını yazacak.

```
public void metodTetikle(int formNo)
{
    userEvent(formNo);
}
```

Bir de şimdi olaydan belli bir delegenin çıkarılması işlemi var değil mi? Öyle ya yaratılan formlardan bir tanesi kapatılırsa yazdırma metodunun tetiklenmesi istisna'ya (exception) yol açar. Bu yüzden bu formun metodunun listeden çıkarılması gerekir. Aşağıdaki kod bu işi yaparak bizi istenmeyen hatalardan korumuş olur.

```
public void elemanCikart(FrmUser frmUser)
{
    this.userEvent -= new userDelegate(frmUser.writeFormNo);
}
```

Yukarıdaki kodun tamamı ana formumuz içinde yer alan kodlardı. Şimdi de ana formda tuşa basılınca oluşturulan formun içinde koda bakalım.

Formlara yazı yazdırmak için tuşa bastığımızda ana formumuzdaki metodTetikle metodunu çağırarak olayın tetiklenmesini sağlıyoruz.

```
private void btnActive_Click(object sender, System.EventArgs e)
```

```
{
```

```
form1.metodTetikle(formNo);
```

```
}
```

Ancak her formun içinde forma mesajı yazacak olan kısım aşağıdaki metoddakigibidir. Şimdi diyeceksiniz, niye aynı form içindeki metodu başka bir formu kullanarak çağırıyorsunuz. Ama buradaki fark ekrana mesaj yazacak olan form sayısının bir tane olmamasıdır. N tane forma aynı anda mesaj yazdırmak istediğimiz için bir başka sınıfta veya formda tanımlanmış olan bir olayı tetikleyerek bu işi yapıyoruz. Aksi halde N tane formu tek tek dolaşmamız gerekirdi. Bu ise performans kaybına yol açtığı gibi daha büyük programlarda kod karmaşasına yol açabilmektedir.

```
public void writeFormNo(int tetiklenenNo)
{
    lblAciklama.Text="Tuşa Basılan Form Numarası:"+tetiklenenNo;
}
```


Galiba bir tek sonradan oluşturulmuş bir form kapatılınca ana formumuzdaki delege çıkartma metodunun nasıl çağrılacağı kaldı. Onu da sonradan oluşturulan formumuzun Dispose metodu içine aşağıdaki gibi yazarsak hiç bir şekilde hatayla karşılaşmayız.

```
form1.elemanCikart(this);
```

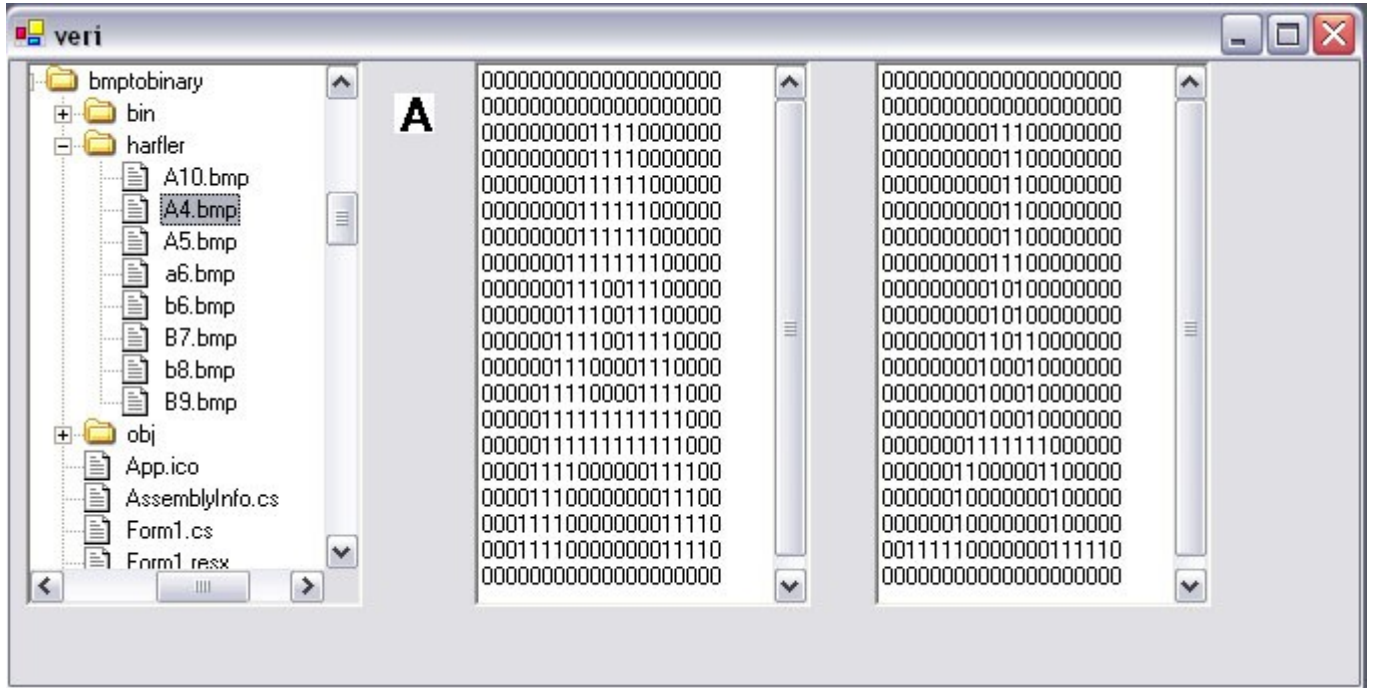
Evet yukarıda delege ve olayın nasıl bir arada kullanıldığını basit bir örnek üzerinde anlatmış oldum. Yukarıda delege ve olayın bir arada kullanıldığı örnek programı [buraya](#) tıklayarak indirebilirsiniz.

Son olarak iyi anlayamadığınız veya kafanıza takılan kısımlar için mail adresim aytacozy@msakademik.net

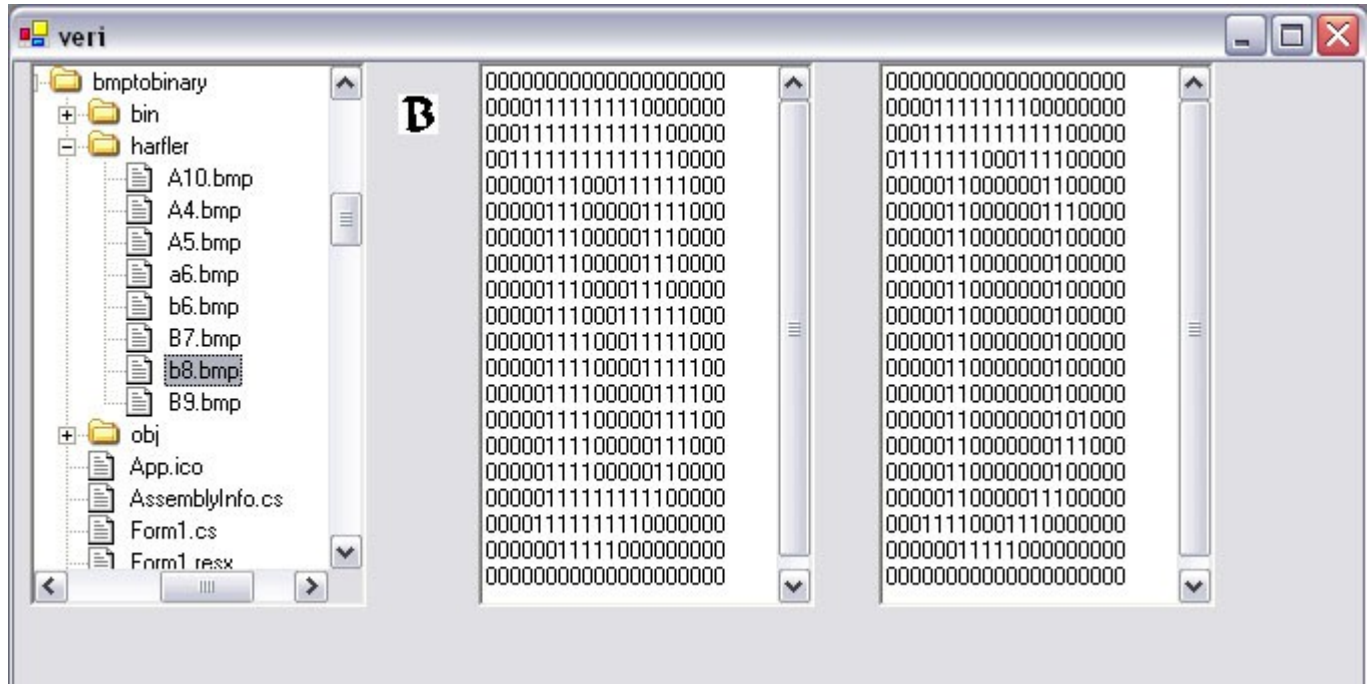
C# ile "Bitmap to Binary" ve "Bitmap İnceltme" Algoritması

Bu yazıda monochrome(siyah beyaz) Bitmap resimlerinin binary(0,1) formata dönüştürülmesi ve dönüştürülen binary resim üzerinde inceltme yapma algoritmasının ne şekilde uygulandığına ilişkin örnek uygulama vereceğim.

Yazdığım örnek uygulamanın örnek ekran çıktıları aşağıdaki gibidir.



Siyah Beyaz formatındaki A4.bmp resminin ilk görüntüsü binary formatındaki halidir, ikinci görüntü ise bu binary bilgilerin inceltilmesinden elde edilmiştir.



Siyah Beyaz formatındaki B4.bmp resminin ilk görüntüsü binary formatındaki halidir, ikinci görüntü ise bu binary bilgilerin inceltilmesinden elde edilmiştir.

Not : İki resim arasındaki farkı göremiyorsanız ekranınıza biraz daha uzaktan şaşı şekilde bakın.

Yukarıdaki resim dosyalarını projedeki harfler klasöründe bulabilirsiniz.

Projeye ait kaynak dosyaları indirmek için tıklayınız.

Kaynak kodlarla ilgili her türlü sorunuzu bana iletebilirsiniz.

Ref ve Out Anahtar Sözcüklerinin Kullanımı

Bu yazıda C#'ın önemli iki anahtar kelimesi olan ve değer türleride dahil olmak üzere bütün veri türlerini referans yolu ile metotlara aktarmamızı sağlayan **ref** ve **out**'un kullanımını göstereceğim.

C# dilinde temel olarak iki veri türü vardır. Bunlardan birincisi referans türleri ikincisi ise değer türleridir. Referans türleri bir ifade (metot çağırımı, atama ifadesi vs.) içinde kullanıldığı zaman nesnenin bellekteki adresi üzerinden işlem yapılır. Yani nesnenin bütün verisi ayrıca kopyalanmaz. Değer türlerinde ise durum daha farklıdır. Değer türleri yani int, byte, bool gibi veri türleri herhangi bir ifade içinde kullanılırsa değişkenin yeni bir kopyası çıkarılır ve işlemler bu yeni kopya üzerinden gerçekleştirilir. Dolayısıyla orjinal nesnenin değeri hiç bir şekilde değiştirilemez. Asıl konumuza geçmeden önce değişkenlerin referans yolu ile aktarımının ne demek olduğunu ve bu iki tür arasındaki farkı daha iyi anlamak için basit bir örnek vermekte fayda görüyorum.

Aşağıdaki programdaki gibi **x** ve **y** gibi iki int türünden değişkenimiz olsun. Bu değişkenlerin değerlerini değiştirmek (swap) için Degistir() isimli bir metot yazmak istediğimizi düşünelim. Bu metot ilk akla gelebilecek şekilde aşağıdaki gibi yazılır.

```
using System;

namespace ConsoleApplication2
{
    class Class1
    {
        static void Main()
        {
            int x = 10;
            int y = 20;

            Degistir(x,y);

            Console.WriteLine("X = " + x.ToString());
        }
    }
}
```

```

        Console.WriteLine("Y = " + y.ToString());
    }

    static void Degistir(int x, int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
}

```

Yukarıdaki programı derleyip çalıştırdığımızda x ve y değişkenlerinin değerlerinin değiştirilmediğini ve aynı değerde kaldıklarını görürüz. Bunun sebebi Degistir() metoduna gelen x ve y değişkenleri ile Main() metodunda tanımlamış olduğumuz x ve y değişkenlerinden tamamen bağımsız yeni değişkenler olmasıdır. Dolayısıyla Degistir() metodunda yaptığımız değişiklikler orjinal değişkenlerimizi hiç bir şekilde etkilememiştir. Değişkenlerin bu şekilde aktarılmasına "değer yolu ile aktarma" yada "pass by value" denilmektedir.

Yukarıdaki programı aşağıdaki gibi biraz değiştirip x ve y değişkenleri bir sınıf aracılığıyla metota gönderirsek değiştirme işlemini yapabiliriz.

```

using System;

namespace ConsoleApplication2
{
    class C
    {
        public int deger;
        public C(int x)
        {
            deger = x;
        }
    }

    class Class1
    {
        static void Main()
        {
            C x = new C(10);
            C y = new C(20);

            Degistir(x,y);

            Console.WriteLine("X = " + x.deger.ToString());
            Console.WriteLine("Y = " + y.deger.ToString());
        }

        static void Degistir(C x, C y)
        {
            int temp = x.deger;
            x.deger = y.deger;
            y.deger = temp;
        }
    }
}

```

```
}  
}
```

Bu programı derleyip çalıştırdığımızda x ve y nesnelerinin değer üye elamanlarının yer değiştirildiğini göreceksiniz. Yer değiştirmeyi yapabilmemizin sebebi x ve y değişkenlerinin referans yolu ile metoda geçirilmesidir. Yani Degistir() metodundaki x ve y değişkenleri ile Main() metodundaki x ve y değişkenleri bellekteki aynı bölgeyi temsil etmektedirler. Dolayısıyla Degistir() metodunda yaptığımız bir değişiklik Main() metodundaki değişkene de yansıtacaktır. Nesnelerin metotlara bu şekilde aktarılmasına ise "referans yolu ile aktarım" yada "pass by reference" denilmektedir.

Bir referans tipi olan string türü ile ilgili önemli bir istisna vardır. string referans türü olmasına rağmen metotlara string türünden değişkenler geçirilirken değer tiplerinde olduğu gibi kopyalanarak geçirilirler. Yani int türünden bir değişken ile string türünden bir değişken metotlara değer yolu ile aktarılırlar. Bunu test etmek için birinci programdaki int türü yerine string türünü kullanabilirsiniz.

Ref Anahtar Sözcüğü

Yukarıda denildiği gibi değer tipleri(int, double, byte vs.) metotlara kopyalanarak geçirilirler yani değişkenin birebir yeni bir kopyası oluşturulur. Ancak bazı durumlarda değer tiplerini de referansları ile metotlara geçirmek isteyebiliriz. C ve C++ dillerinde değer tiplerini referans yolu ile geçirmek için göstericilerden faydalanır. Yani metotlara değişkenlerin adresleri geçirilir. C# ta bu işlemi yapmak için gösterici yerine yeni bir anahtar sözcük olan **ref** kullanılır. ref anahtar sözcüğü değer türlerinin metotlara referans yolu ile geçirilmesini sağlar. Referans türleri zaten referans yolu ile geçirildiği için bu türler için ref anahtar sözcüğünü kullanmak gereksizdir. Ancak kullanımı tamamen geçerli kılınmıştır.

ref sözcüğü metot çağrımında ve metot bildiriminde aynı anda kullanılmalıdır. Yani metot bildiriminde ref ile birlikte kullanılan bir değişken, metot çağrılırken ref ile çağrılmalıdır. Yukarıda yazdığımız birinci programı ref sözcüğünün kullanımı ile yeniden düzenlersek Degistir() metodunun istediğimiz şekilde çalışmasını sağlayabiliriz.

```
using System;  
  
namespace ConsoleApplication2  
{  
    class Class1  
    {  
        static void Main()  
        {  
            int x = 10;  
            int y = 20;  
  
            Degistir(ref x, ref y);  
  
            Console.WriteLine("X = " + x.ToString());  
            Console.WriteLine("Y = " + y.ToString());  
        }  
  
        static void Degistir(ref int x, ref int y)  
        {  
            int temp = x;
```

```
        x = y;  
        y = temp;  
    }  
}
```

Bu programı derleyip çalıştırdığımızda x ve y değişkenlerinin değerlerinin değiştirildiğini göreceksiniz. Çünkü Degistir() metotundaki x ve y değişkenleri ile Main() metodundaki x ve y değişkenleri aynı bellek bölgesindeki değeri temsil etmektedirler. Birinde yapılan değişiklik diğerinde etkilemektedir. Yani ilk durumdan farklı olarak ortada iki değişken yoktur, tek bir değişken vardır.

Unutmamamız gereken nokta metot çağırımının da ref anahtar sözcüğü ile birlikte yapılması zorunluluğudur. Eğer Degistir() metodunu Degistir(ref x, ref y) yerine Degistir(x,y) şeklinde çağırmış olsaydık derleme aşamasında

Argument '1': cannot convert from 'int' to 'ref int'
Argument '2': cannot convert from 'int' to 'ref int'

hatalarını alırdık.

ref sözcüğünün kullanımı ile ilgili diğer bir önemli nokta ise ref ile kullanılacak değişkenlere mutlaka değer atanmış olma zorunluluğudur. Herhangi bir değer verilmemiş değişkeni ref ile de olsa kullanamayız. Kullandığımız takdirde ise derleme aşamasında "Use of unassigned local variable" hatasını alırız. Bu durum ref sözcüğünün bir kısıtı olarak düşünülebilir. Ancak birazdan göreceğimiz out sözcüğü ile bu kısıtı ortadan kaldıracak şekilde göreceğiz.

Out Anahtar Sözcüğü

Out anahtar sözcüğünün kullanım amacı ref anahtar sözcüğünün kullanımı ile tamamen aynıdır. Yani out ile de değer tipleri referans yolu ile aktarılır. Aralarındaki tek fark out ile kullanılacak değişkenlere ilk değer verme zorunluluğunun olmamasıdır. Yani ref sözcüğünün kullanımındaki kısıt, out ile birlikte ortadan kaldırılmıştır. out anahtar sözcüğünü genellikle bir metottan birden fazla geri dönüş değeri bekliyorsak kullanırız.

Yukarıdaki Degistir() metodunu out ile kullanılabilecek şekilde değiştirdiğimizde x ve y değişkenlerine ilk değer verme zorunluluğumuz kalkacaktır.

ref ve out sözcüklerinin C# dilinde kullanılmasının küçük bir farkı olsada IL dilinde ref ve out aynı şekilde implemente edilmiştir. ILDASM aracı ile Degistir() metodunun hem ref hemde out versiyonlarının bildiriminin aşağıdaki gibi olduğunu görürüz.

```
.method private hidebysig static void Degistir(int32& x, int32& y) cil managed  
{  
    .maxstack 2  
    .locals init (int32 V_0)  
    IL_0000: ldarg.0  
    IL_0001: ldind.i4  
    IL_0002: stloc.0  
    IL_0003: ldarg.0  
    IL_0004: ldarg.1  
    IL_0005: ldind.i4  
    IL_0006: stind.i4  
    IL_0007: ldarg.1
```

```
IL_0008: ldloc.0
IL_0009: stind.i4
IL_000a: ret
}
```

Bu durum ref ve out sözcüklerinin kullanımındaki farkın C# derleyicisi ile sınırlı olduğunu göstermektedir. Yani CLR ile ilgili bir fark değildir.

Yazıyı bitirmeden önce out anahtar sözcüğünün kullanımına bir örnek vermek istiyorum. İki sayıdan büyük olanına geri dönen bir metot yazmak istediğimizi düşünelim. Aynı zamanda da bu iki sayıdan birincisinin mi ikincisinin mi büyük olduğunda bu metotla öğrenmek istiyoruz. Her metodun tek bir geri dönüş değeri olabileceğine göre klasik yöntemlerle bunu ideal(!) bir şekilde gerçekleştiremeyiz. Bunun için ilk değer verilmemiş yeni bir parametreyi daha Max() isimli metoda göndereceğiz. Bu parametre metot içinde değiştirilerek kendisini çağıran metoda iletilecektir.

```
using System;

namespace Out
{
    class Class1
    {
        static void Main()
        {
            bool b;

            int max = Max(9,2,out b);

            Console.WriteLine(b);
        }

        static int Max(int x,int y, out bool b)
        {
            if(x > y)
                b = true;
            else
                b = false;

            return Math.Max(x,y);
        }
    }
}
```

Yukarıdaki işlemi ref anahtar sözcüğü ile de yapabildik ancak bir metodun içinde değeri belirlenecek bir değişkene ilk değer vermek gereksiz ve mantıksızdır. Dolayısıyla bir metodun birden fazla değer geri vermesini istediğimiz durumlarda out anahtar sözcüğünü kullanmamız daha okunabilir ve daha düzenli programcılık açısından önemlidir.

Bir sonraki yazıda görüşmek üzere...

Abstract Factory Tasarım Deseni(Design Pattern)

Singleton deseni ile başladığım "design pattern" yazı dizisine "Abstract Factory" deseni ile devam ediyoruz. Bu yazıda "Creational" desenler grubunun en önemli ve en sık kullanılan deseni olan Abstract Factory(Soyut Fabrika) tasarım deseninin C# ile ne şekilde uygulandığını bir örnek üzerinden göstereceğim.

İlk yazımda da bahsettiğim gibi "Creational" grubundaki desenler bir yada daha çok nesnenin çeşitli şekillerde oluşturulması ile ilgili desenlerdir. Bu kategoride ele alınan "Abstract Factory" ise birbirleriyle ilişkili yada birbirlerine bağlı olan nesnelerin oluşturulmasını en etkin bir şekilde çözmeyi hedefler. Bu hedefe ulaşmak için soyut sınıflardan(abstract class) veya arayüzlerden(interface) yoğun bir şekilde faydalanmaktadır. "Abstract Factory" deseninin ana teması belirli sınıfların içerdiği ortak arayüzü soyut bir sınıf yada arayüz olarak tasarlamaktır. Böylece nesneleri üreten sınıf, hangi nesnenin üretileceği ile pek fazla ilgilenmesi gerekmez. İlgilenmesi gereken nokta oluşturacağı nesnenin hangi arayüzleri desteklediği yada uyguladığıdır. Bahsi geçen mekanizmalarla deseni oluşturduğumuz anda çalışma zamanında hangi nesnenin oluşturulması gerektiğini bilmeden nesnelerin oluşturulmasını yönetebiliriz.

Eğer bir nesne oluşturacaksanız ve tam olarak hangi nesnenin oluşturulacağına bir switch yada if deyimi ile karar veriyorsanız muhtemelen her nesneyi oluşturduğunuzda aynı switch yapısını kullanmak zorunda kalacaksınız. Bu tür tekrarları önlemek için "Abstract Factory" deseninden faydalanılabilir. Bu elbetteki nesnelerin ortak bir arayüzü uygulamış olma zorunluluğunun getirdiği bir faydadır.

Şimdi de gerçek dünyadan bir örnek vererek "Abstract Factory" deseninin hangi durumlarda kullanabileceğimizi ve soyut fabrika mantığını netleştirelim. Bir CD sürücüsü düşünün. CD sürücüsü kendisine sürülen CD leri okumakla sorumludur. Hiç bir zaman sürülen CD nin şekli ve biçimiyle ilgilenmez. Ama sürülen CD nin okunabilmesi için de belirli şartların yerine getirildiğini farzeder. Yani siz CD sürücüsüne CD olmayan ama CD ye benzeyen bir cisim yerleştirirseniz onu da okumaya çalışır.(eğer CD sürücünüz bozulmazsa!) Çünkü okumaya çalıştığı cismin ne olduğu ile pek ilgilenmez CD sürücüsü. Buradaki örnekte CD sürücüsünün okuma yapabilmesi için gereken şartları bir soyut fabrika sınıfı ile modelleyebiliriz. Kare yada daire şeklindeki gerçek CD ler ise bu soyut

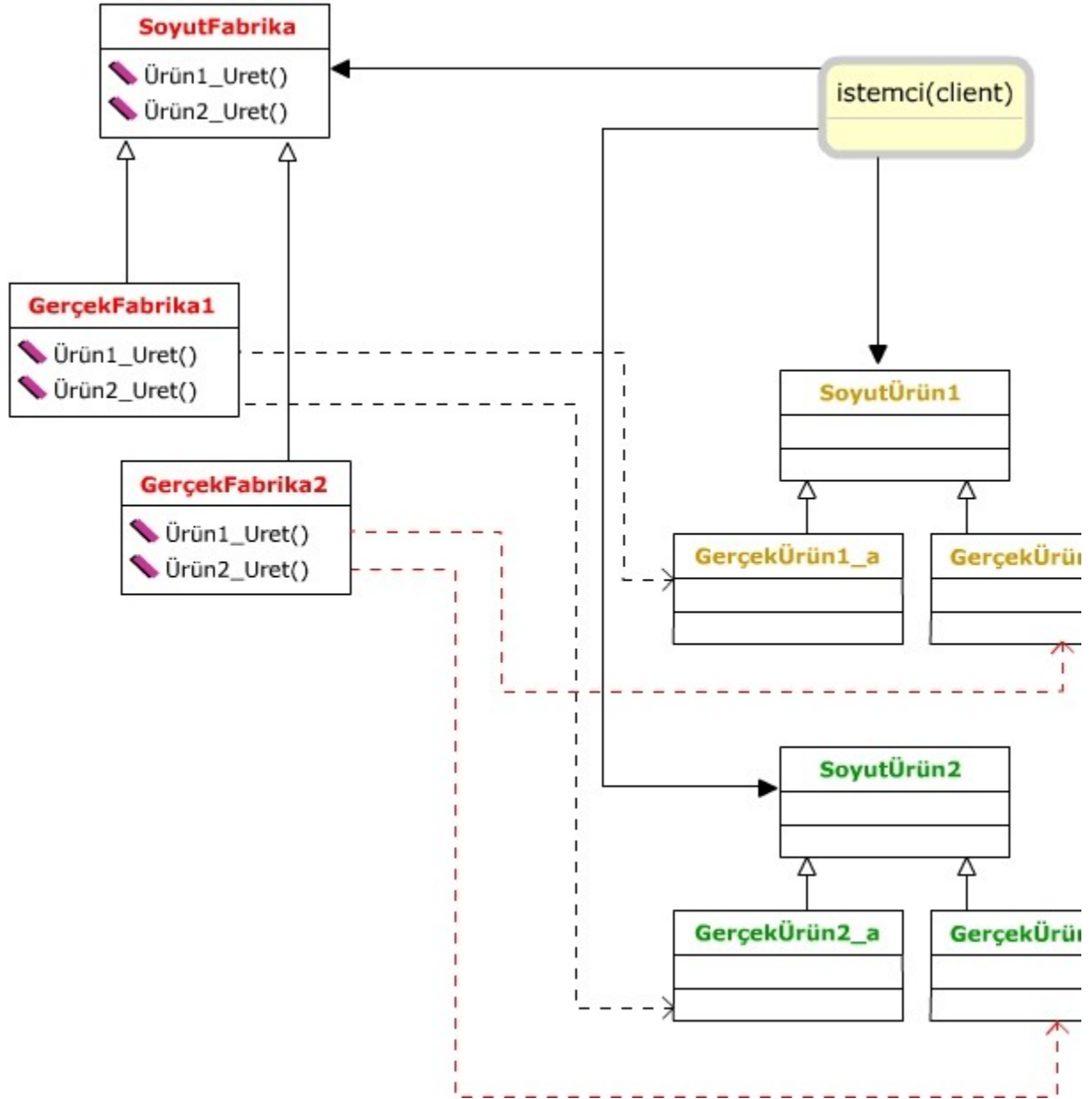
fabrika sınıfı tarafından belirlenen şartları destekleyen gerçek nesnelerdir. CD sürücüsünün kendisi ise soyut fabrika tarafından belirlenen standartlar çerçevesi içerisinde CD nin ne tür bir CD olduğundan bağımsız bir şekilde bilgiyi okuyan birimdir. Bu, "abstract factory" desenindeki client yani istemci sınıfa denk düşer ki bu sınıf nesnelerin yaratılmasından sorumludur.

Bu giriş bilgilerinden sonra "abstract factory" deseninin temel özelliklerini kısaca özetleyelim.

- "Abstract Factory", nesneleri oluşturan bir sınıftır. Oluşturulan bu nesneler birbirleriyle ilişkili olan nesnelerdir. Diğer bir deyişle aynı arayüzü uygulamış olan nesnelerdir.
- Üretilen nesnelerin kendisiyle ilgilenilmez. İlgilenilen nokta oluşturulacak nesnelerin sağladığı arayüzlerdir. Dolayısıyla aynı arayüzü uygulayan yeni nesneleri desene eklemek çok kolay ve esnektir.
- Bu desende üretilcek nesnelerin birbirleriyle ilişkili olması beklenir.

UML Modeli

Aşağıdaki şekil "abstract factory" tasarım deseninin yapısal UML diagramını göstermektedir. Şemadaki her bir şekil desendeki bir sınıfı modellemektedir. Ayrıca desendeki sınıflar arasındaki ilişkilerde detaylı bir şekilde gösterilmiştir.



Yukarıda şemayı kısaca açıklamakta fayda var. Şemadan da görüleceği üzere "abstract factory" deseninde 3 ana yapı vardır. İlk yapı nesnelerin oluşturulmasından sorumlu soyut ve gerçek fabrikalar, ikinci yapı soyut fabrikadan türeyen gerçek fabrikaların ürettiği ürünleri temsil eden soyut ve gerçek ürün sınıflar, son yapı ise herhangi bir ürünü, kendisine parametre olarak verilen soyut fabrikaları kullanarak üreten istemci(client) sınıfıdır.

SoyutFabrika sınıfı gerçek fabrikaların uygulaması gereken arayüzü temsil eder. Bu sınıf, bütün metotları soyut olan sınıf olabileceği gibi bir arayüz de olabilir. Uygulamanızın ihtiyacına göre dilediğinizi kullanabilirsiniz. SoyutFabrika sınıfında ürün1 ve ürün2'nin üretilmesinden sorumlu iki tane metot bulunmaktadır. Dolayısıyla bütün gerçek fabrikaların hem ürün1'i hemde ürün2'yi ürettiği kabul edilmektedir. Her bir ürünün ortak özelliklerini belirlemek ve ana yapıda toplamak için SoyutUrun1 ve SoyutUrun2 sınıfları oluşturulur. Bu sınıflarda herhangi bir ürüne özel bilgi bulunmamaktadır. Asıl bilgi bu soyut ürünlerden türeyen GercekUrun sınıflarında bulunmaktadır. Her bir fabrikanın ürettiği ürünleri modelleyen sınıflarda yukarıdaki şekilde gösterilmiştir. Asıl önemli mesele

ise gerçek fabrikaların üretimden sorumlu metotlarının ne şekilde geri döneceğidir. Yukarıdaki şemadan da görüleceği üzere bu metotlar üreteceği ürünün soyut sınıfına dönmektedir. Yani üretim sonucunda geri dönen gerçek ürün nesnesi değildir. Şemada Client olarak gösterilen sınıfın yapısı ise şu şekildedir : Client sınıfı yapıcı metoduna bir soyut fabrika nesnesi alır. Ve soyut fabrikanın üretimden sorumlu metotlarını kullanarak soyut ürünleri üretir. Dikkat ederseniz Client sınıfı hangi gerçek fabrikanın üretim yaptığından ve üretilen ürünün gerçek özelliklerinden haberi yoktur. Client sadece soyut fabrikanın içerdiği temel özelliklerin farkındadır. Bunu şemadaki kalın ve kesikli oklardan görmek mümkündür.

Desenin C# ile Gerçekleştirilmesi

Yukarıdaki yapısal örneği verdikten sonra gerçek bir örnek ile bu deseni nasıl gerçekleştirebileceğimizi inceleyelim. Bu örnekte araba kasası ve araba lastiği üreten farklı iki firmanın üretimi modellenmektedir.

Önce örneği kabaca inceleyin, ardından açıklamaları okuyun.

```
using System;

namespace DesignPattern
{
    abstract class SoyutArabaFabrikasi
    {
        abstract public SoyutArabaKasasi KasaUret();
        abstract public SoyutArabaLastigi LastikUret();
    }

    class MercedesFabrikasi : SoyutArabaFabrikasi
    {
        public override SoyutArabaKasasi KasaUret()
        {
            return new MercedesE200();
        }

        public override SoyutArabaLastigi LastikUret()
        {
            return new MercedesLastik();
        }
    }

    class FordFabrikasi : SoyutArabaFabrikasi
    {
        public override SoyutArabaKasasi KasaUret()
        {
            return new FordFocus();
        }

        public override SoyutArabaLastigi LastikUret()
        {
            return new FordLastik();
        }
    }
}
```

```
abstract class SoyutArabaKasasi
{
    abstract public void LastikTak(SoyutArabaLastigi a );
}

abstract class SoyutArabaLastigi
{
}

class MercedesE200 : SoyutArabaKasasi
{
    public override void LastikTak(SoyutArabaLastigi lastik)
    {
        Console.WriteLine( lastik + " lastikli MercedesE200");
    }
}

class FordFocus : SoyutArabaKasasi
{
    public override void LastikTak(SoyutArabaLastigi lastik)
    {
        Console.WriteLine( lastik + " lastikli FordFocus");
    }
}

class MercedesLastik : SoyutArabaLastigi
{
}

class FordLastik : SoyutArabaLastigi
{
}

class FabrikaOtomasyon
{
    private SoyutArabaKasasi ArabaKasasi;
    private SoyutArabaLastigi ArabaLastigi;

    public FabrikaOtomasyon( SoyutArabaFabrikasi fabrika )
    {
        ArabaKasasi = fabrika.KasaUret();
        ArabaLastigi = fabrika.LastikUret();
    }

    public void LastikTak()
    {
        ArabaKasasi.LastikTak( ArabaLastigi );
    }
}
```

```

    }

    class UretimBandi
    {
        public static void Main()
        {
            SoyutArabaFabrikasi fabrika1 = new MercedesFabrikasi();
            FabrikaOtomasyon fo1 = new FabrikaOtomasyon( fabrika1 );
            fo1.LastikTak();

            SoyutArabaFabrikasi fabrika2 = new FordFabrikasi();
            FabrikaOtomasyon fo2 = new FabrikaOtomasyon( fabrika2 );
            fo2.LastikTak();
        }
    }
}

```

Yukarıdaki örnekte SoyutArabaFabrikasi sınıfı iki metot içermektedir. Bu metotlar SoyutArabaFabrikasi sınıfından türeyecek sınıfların uygulaması gereken metotlardır. Çünkü metotlar abstract olarak bildirilmiştir. Bu metotlar gerçek fabrika sınıflarının araba kasası ve araba lastiği üretmesi gerektiğinin belirtisidir. Zira görüldüğü üzere SoyutArabaFabrikasi sınıfından türeyen MercedesFabrikasi ve FordFabrikasi kendilerine has lastikleri ve kasaları üretmek için soyut fabrika sınıfının metotlarını kullanmaktadır. Bu metotlar geri dönüş değeri olarak soyut ürün sınıflarını temsil eden sınıfları döndürmektedirler. Örneğin KasaUret() metodu her bir fabrika için farklı ürün üretmesine rağmen her bir ürün SoyutArabaKasasi sınıfındaki metotları uyguladığı için gerçek ürünler birbirleriyle ilişkili hale gelir. Mercedes fabrikası KasaUret() metodu ile MercedesE200 ürününü döndürmesine rağmen Ford fabrikası aynı metotla FordFocus ürününü döndürmektedir. Ancak her iki fabrikanın da ürettiği ürün SoyutArabaKasasi sınıfından türediği için herhangi bir çelişki olmamaktadır.

SoyutArabaKasasi sınıfındaki LastikTak() sınıfı fabrikadan üretilen ürünlerin birbirleriyle karıştırılmadan esnek bir şekilde nasıl ilişkilendirildiğini gösterilmektedir. Bu metot parametre olarak gerçek lastik ürünü yerine soyut lastik ürünü alır. Dolayısıyla herhangi bir fabrikadan üretilen lastik ürünü bu metoda parametre olarak geçirilebilir.

FabrikaOtomasyon sınıfı kendisine verilen bir soyut fabrika nesnesi üzerinden kasa ve lastik üretir ve üretilen lastiği, lastiğin gerçek türünü bilmeden üretilen araba kasası ile ilişkilendirir. Dikkat ederseniz bu sınıf üretimin yapılacağı fabrikanın hangi fabrika olduğu ve üretilen ürünlerin gerçekte hangi ürünler olduğu ile ilgilenmez.

Son olarak tasarladığımız bütün bu sınıfları test edecek UretimBandi sınıfını inceleyelim. Bu sınıf içerisinde ürünleri üretilecek fabrikanın soyut nesnesi oluşturulur ve FabrikaOtomasyonu nesnesine parametre olarak verilir. SoyutFabrika nesnesini alan FabrikaOtomasyonu bu nesnenin standart üretimden sorumlu metotlarını kullanarak kasa ve lastik üretir. Ardından SoyutArabaKasasi sınıfının LastikTak() metodunu kullanarak kasa ve lastik ürünlerini ilişkilendirir.

Bu örnekte kullanılan soyut sınıfların yerine arayüzleride kullanmak mümkündür. Daha önce de dediğim gibi siz uygulamanızın durumuna göre herhangi birini seçebilirsiniz.

Diğer bir "Creational" deseni olan "Builder" desenini anlatacağım yazıda görüşmek üzere...

C# ile .NET Ortamında Threading'e Giriş

İnsan vücudunda aynı anda bir çok iş birlikte yapılır, mesela kalbimiz tüm vücuda kan pompalarken midemiz yediğimiz bir şeyi sindirmek için gerekli enzimleri salgılar: Bilgisayarların zaman içinde çok hızlı gelişmeleri sonucunda insanlar bu aletlerden daha fazla verim ve hız beklediler ve ortaya atılan birçok çözümden biri de iş parçacıklarını (threads) kullanmak olmuştur.

İş parçacıkları ilk defa Ada programlama dilinde Amerikan ordusunun stratejik yazılımları için kullanılmıştır. Daha sonra C++ dilinde iş parçacıklarını kullanmak için kütüphaneler geliştirilmiştir. Bu kütüphaneler sayesinde zaman içinde C++ dilinde yazılmış programlarda iş parçacıklarını kullanmak bir takım faydalar sağlamıştır.

İş parçacıklarını .NET ortamında nasıl kullanacağımızı öğrenmeden önce işin teorik temellerini bilmek gerekir. Ayrıca iş parçacıklarını ne zaman ve nasıl programlarımıza katmayı da öğrenmek daha sağlıklı programlar geliştirmeye yardımcı olacaktır. Makalemizin kalan kısmını MSDN kütüphanesindeki iş parçacıkları konusunun baş kısımları oluşturacaktır.

İşletim sistemlerinde aynı anda birden fazla programın çalışması günümüzde mümkün hale gelmiştir. Aslında bir işlemcide aynı anda sadece bir işlem gerçekleşebilir. Fakat başlayan bir işlemin tamamını bitirmeden başka bir işlemin yapılması ile multitasking başarılabılır. Mesela bir taraftan Ms Word diğer taraftan Ms Explorer açık olabilir. İşlemlerin bir alt parçası olan iş parçacıkları (threadler) aynı zamanda bilgisayar ortamında yapılacak olan en küçük görev birimleridir denilebilir. Bir işlem(process) içinde birden fazla iş parçacığı bulunabilir. Her bir iş parçacığı için hata yönetimi(exception handler), öncelik çizelgesi (scheduling priority) ve bir takım yapılar bulunur. Bir önceki cümlede bahsettiğimiz yapılar iş parçacığı hakkında işletim sisteminin tuttuğu bilgilerdir. Bu bilgiler ile iş parçacıklarının sorunsuz olarak çalıştırılması sağlanır.

.NET platformu işlemleri(process) daha küçük bir birim olan **application domain**'lere ayırır. Application domain'ler System.AppDomain sistem alanındaki sınıflar tarafından işlenirler. .NET'te bir veya daha fazla iş parçacığı birden farklı application domain için çalışabilir. Her ne kadar bir application domain sadece bir tane iş parçacığı ile çalışmaya başlasa da zaman içinde birden fazla application domain ve iş parçacığı aynı application domain için çalışabilir. Ayrıca tek bir iş parçacığı birden farklı application domain'ler arasında gidip gelebilir.

Eğer bir işletim sistemi preemptive multitasking'i destekliyorsa bilgisayarda birden fazla programın aynı anda çalışıyormuş hissi yaratılabilir. Bunun için işletim sistemi her bir işlemin belirli bir süre (time slicing) işlemciyi meşgul etme hakkı tanır. İşlemci kullanım süresi dolan işlem bekletilmeye alınır ve bu işlem hakkındaki bilgiler bir yere not edilir. Sonra sırada bekleyen (thread queue) başka bir işlemin belirli bir süre işlemciyi kullanmasına izin verilir. İkinci işlemin de süresi dolunca bu işlem hakkında bilgiler bir yere kaydedilir ve tekrar kuyruğa geri döner. Sonra sıradaki diğer işleme başlanır... Ve bu şekilde devam eder.

İşlemlerin işlemciyi kullanma aralıkları işletim sistemine ve işlemciye göre değişir. İşlemciyi kullanma aralıkları o kadar küçük ve işlemciler o kadar hızlıdır ki bir çok iş parçacığının çalıştırıldığı bir işletim sisteminde aynı anda birden fazla programın çalıştığı hissi kullanıcıya uyanır.

Ne Zaman Birden Fazla İş Parçacığı ile Çalışmalı?

Eğer geliştirdiğimiz programlar kullanıcı ile sık sık etkileşime geçiyor ve kullanıcılara sistemin cevabının çok hızlı olması gerekiyorsa iş parçacıklarını kullanmak yerinde olacaktır. Eğer sizin programınızda sadece bir iş parçacığı yeterli oluyorsa ve .NET remoting veya XML Web servisleri kullanıyorsanız yine iş parçacıklarından faydalanmak suretiyle programınızın kullanıcıya vereceği tepkiyi daha kısa sürede üretebilirsiniz. Son olarak yoğun bir biçimde I/O işlemleri gerektiren programlarda iş parçacıklarından faydalanmak uygun olacaktır.

Çoklu İş Parçacıklarının Avantajları

Birden fazla iş parçacığı kullanmakla hem programın kullanıcıya olan cevap süresi (Kullanıcı arayüzünde) kısılır hem de aynı anda arka planda verilerin hızlıca işlenip sonuca ulaşılması sağlanır. Mesela biz bir taraftan Excel çalışma sayfasına verileri giriyorken diğer taraftan Excel çalışma sayfasında tanımlanan formüllere göre diğer hücrelerin değerlerini hesaplayıp yazar.

Bir programda hem iş parçacıkları kullanılır hem de bu program birden fazla işlemcisi olan bir makinede çalıştırılırsa kullanıcıların programdan memnuniyetlerinde çok ileri seviyede artışlar olur. Geliştirdiğimiz bir programda birden fazla iş parçası kullanmakla:

- Ağ üzerinde, web sunucu ile yada veritabanı ile veri alışverişi
- Çok uzun süren hesaplamalı işlemleri
- Farklı öncelikteki görevlerde. (Mesela yüksek öncelikli iş parçacıkları ile hemen bitirilmesi gereken işlemler yapılırken diğer iş parçacıkları başka işleri yapabilir.)
- Grafik arayüzünde kullanıcıya daha hızlı cevap verilirken arka planda diğer veri işleme işleri gerçekleşir.

Çoklu İş Parçacıklarının Dezavantajları

Mümkün olduğunca az sayıda iş parçacığını aynı anda kullanmak tavsiye edilir. Bu şekilde işletim sisteminin daha az kaynağını kullanır ve performansı artırabiliriz. Ayrıca iş parçacıkları için hem ek kaynak gereksimi hem de programda çakışma ihtimalleri vardır. İş parçacıkları için gerekli ek kaynaklar şunlardır:

- İşletim sistemleri işlemler, AppDomain nesneleri ve iş parçacıkları hakkında bilgileri tutmak zordundadırlar. Yani, işlemler, AppDomain nesneleri ve iş parçacıklarının oluşturulmasında kullanılabilir hafıza sınırlayıcı bir etken olabilir.
- Çok sayıda iş parçacıkları ile çalışma durumlarında, işlemci iş parçacıklarının gerektirdiği işleri yapmaktan çok iş parçacıkları arasında geçiş için meşgul olur. Ayrıca bir işlemin içinde çok sayıda iş parçacığı varsa bu iş parçacıklarının işlenmesi için daha az sayıda şans doğacaktır.
- Birden fazla iş parçacığı aynı anda işlemeye çalışmak çok karmaşık bir durum yaratır ve bir çok hataya sebep olabilir.
- Bir iş parçacığının işi bittiğinde onu yok etmek için yine çok karmaşık kodlar yazmak ve sonuçlarını tahmin etmek gerekir.

Kaynakları paylaşmak ve birden fazla işlem için aynı anda kullanmaya çalışmak sistemde çakışmalara yol açabilir. Muhtemel çakışmaların önüne geçmek için senkronizasyon yapmak veya paylaşılan kaynaklara erişimi kontrol altına almak gerekir. Aynı veya farklı AppDomain'lerde erişimleri senkronize etmede başarısızlık durumumda **deadlock** (aynı anda iki iş parçacığının boş durdukları halde birbirlerini sonsuza kadar beklemleri) problemi ortaya çıkabilir. Fakat sistemin sağladığı senkronize nesneleri ile aynı kaynağın farklı farklı iş parçacıkları tarafından kullanılması koordine edilebilir. Tabiki iş

paracıklarının sayısını azaltmak da kaynakların senkronize olarak kullanılmasını kolaylařtırır.

Senkronize edilmesi gereken kaynaklar řunlardır:

- Sistem kaynakları (iletiřim portları gibi)
- Birden fazla iřlem tarafından kullanılan kaynaklar.(dosya yneticileri)
- Tek bir AppDomain'e ait (global, static ve rnek veri alanları) fakat farklı iř paracıkları tarafından kullanılan kaynaklar.

İř Paracıkları ve Uygulama Tasarımı

Genelde kısa srecek grevler ve zel olarak zamanlama gerektirmeyen iř paracıkları iin ThreadPool sınıfını kullanmak en kolay yoldur. Fakat bir ok sebepten dolayı kendimize ait thread sınıfını yazmak daha iyi olacaktır. Bunun en nemli nedenleri řunlardır:

- Eęer zel ncelięe sahip bir grev tanımlayacak ve kullanacaksak.
- Eęer uzun srebilecek bir iř yapmak gerekiyorsa.
- Eęer iř paracıklarınızı tek-iř paracıęı apartmanına koymak gerekiyorsa (ThreadPool sınıfındaki tm iř paracıkları oklu-iřparacıęı apartmanına koyulur.)
- Eęer bir iř paracıęı iin sabit bir kimlik kullanmak gerekiyorsa .

İleriki yazılarda .NET ortamında C# ile iř paracıklarının kullanımlarını daha detaylı olarak inceleyeceęiz ve bir ok rnek kod zerinde duracaęız.

C# ile İlgili Sık Sorulan Sorular (SSS)

Bu yazıda C# dili ilgili sık sorulan sorulara yanıt verilmiştir.

Aşağıdaki C# ile ilgili sık sorulan sorular www.msdn.com adresinde faaliyet gösteren Microsoft Visual C# ekibi tarafından hazırlanmıştır.

S - 1 : DllImport niteliğini neden çalıştıramıyorum?

C - 1 : DllImport ile işaretlenen bütün metotlar **public static extern** olarak bildirilmelidir.

S - 2 : Yazdığım switch ifadeleri farklı bir biçimde çalışıyor. Neden?

C - 2 : C# case blokları için "explicit fall through" özelliğini desteklemez. Buna göre aşağıdaki kod parçası geçersizdir ve C#'ta derlenemez.

```
switch(x)
{
    case 0:
        // bir şeyler yap
    case 1:
        // 0 case'indekine ek olarak birşeyler daha yap
    default:
        // 0 ve 1 durumlarına ek olarak birşeyler daha yap

        break;
}
```

Yukarıdaki kodun verdiği etkiyi C# ile aşağıdaki gibi gerçekleştirebiliriz. (Case' ler arasındaki akışın açıkça belirtildiğine dikkat edin!)

```
class Test
{
    public static void Main()
    {
        int x = 3;

        switch(x)
        {
            case 0:
                // bir şeyler yap
                goto case 1;
            case 1:
                // 0 case'indekine ek olarak birşeyler daha yap
                goto default;
            default:
                // 0 ve 1 durumlarına ek olarak birşeyler daha yap
                break;
        }
    }
}
```

S - 3 : const ve static readonly arasındaki farklar nelerdir?

C - 3 : static readonly elemanlar bulundukları sınıfın üye elemanları tarafından değiştirilebilir(!), fakat const olan üye elemanlar asla değiştirilemez ve derleme zamanı sabiti olarak ilk değerleri verilmelidir.

static readonly üye elemanlarının değiştirilebilmesini biraz açacak olursak, static readonly üyeyi içeren sınıf bu üyeyi aşağıdaki durumlarda değiştirebilir :

- değişken ilk değer verilen durumda
- static yapıcı metotlar içinde

S - 4 : trace ve assert komutlarını nasıl gerçekleyebilirim?

C - 4 : Metotlarla birlikte Conditional niteliğini kullanarak gerçekleyebiliriz.

```
class Debug
{
    [conditional("TRACE")]
    public void Trace(string s)
    {
        Console.WriteLine(s);
    }
}

class MyClass
{
    public static void Main()
    {
        Debug.Trace("hello");
    }
}
```

Yukarıdaki örnekte Debug.Trace() metodu ancak ve ancak TRACE ön işlemci seöbolü tanımlanmışsa çağrılacaktır. Komut satırından ön işlemci sembollerini tanımlamak için /D parametresi kullanılabilir. Conditional niteliği ile bildirilen metotların geri dönüş değerinin void olma zorunluluğu vardır.

S - 5 : C#'ta dll oluşturmak için ne yapmalıyım?

C - 5 : Derleyicinin /target:library argümanını kullanmanız gerekir.

S - 6 : checked isimli bir değişken tanımladığımda neden derleme zamanında "syntax error" hatası alıyorum?

C - 6 : Çünkü **checked** C#'ta bir anahtar sözcüktür.

S - 7 : Bir yapıcı metot içinde aşırı yüklenmiş başka bir yapıcı metot nasıl çağrılır (this() ve yapıcımetotadı() şeklindeki çağrılar derlenmiyor)?

C - 7 : Diğer bir yapıcı metot aşağıdaki gibi çağrılabilir.

```
class B
{
    B(int i)
    { }
}

class C : B
{
    C() : base(5)    //
    B(5) i çağırır.
    { }

    C(int i) : this() // C()
    yi çağırır.
    { }

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

S - 8 : C#'ta Visual J++ ta bulunan instanceof operatörünün karşılığı varmıdır?

C - 8 : Evet, **is** operatörü bunun karşılığıdır. Kullanımı aşağıdaki gibidir :

*ifade **is** tür*

S - 9 : C#'ta enum sabitleri nasıl kullanılır.

C - 9 : enum türlerinin kullanımına bir örnek :

```
namespace Foo
{
    enum Colors
    {
        BLUE,
        GREEN
    }

    class Bar
    {
        Colors color;
        Bar() { color = Colors.GREEN;}

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

S - 10 : Geri dönüş değeri olmayan bir metot bildirimi yaptığımda neden (CS1006) hatası almaktayım?

C - 10 : Bir metodun geri dönüş değerini yazmadan bildirirseniz derleyici onu sanki bir yapıcı metot bildiriyormuşsunuz gibi davranır. O halde geri dönüş değeri olmayan bir metot bildirimi için void anahtar sözcüğünü kullanın. Aşağıda bu iki kullanıma örnek verilmiştir.

```
// Bu bildirim CS1006 hatası verir.  
public static staticMethod (mainStatic obj)  
  
// Bu metot ise istenildiği gibi çalışır.  
public static void staticMethod (mainStatic obj)
```

S - 11 : Her birinde farklı Main() metodu olan birden fazla kaynak kod dosyam var: derleme sırasında hangi Main() metodunun kullanılacağını nasıl bildirebilirim?

C - 11 : Programınızın giriş noktası(metodu) Main isimli herhangi bir parametre almayan yada string türünden bir dizi parametresi alan geri dönüş değeri void yada int olan static bir metot olmalıdır.

C# derleyicisi programınızda birden fazla Main metodu bildirmenize izin verir fakat hangi Main() metodunu kullanacağınızı derleme zamanında bildirmeniz gerekir. Main() metodunu belirtirken Main metodunun bulunduğu sınıfın tam yolunu belirtmeniz gerekir. Komut satırından kullanılan /main argümanı bu işe yarar.(Örn : csc /main:MainSınıfı *.cs)

S - 12 : Console.WriteLine() metodu bir string içinde NULL karakteri gördüğünde ekrana yazma işlemini durdururmu?

C - 12 : Çalışma zamanı için string türleri NULL ile sonlandırılmış türler değildir. Dolayısıyla bir string içine NULL karakteri gömebilirsiniz. Console.WriteLine() ve buna benzer metotlar string değişkeninin sonuna kadar işlem yaparlar.

S - 13 : C# ta "Multicast Delegate"(çoklu temsilciler) bildirmek mümkünmüdür, mümkünse sentaksı nasıldır?

C - 13 : Bütün temsilciler varsayılan olarak multicast olarak bildirilir. Dolayısıyla Visual J++ taki gibi ayrıca multicast anahtar sözcüğü yoktur.

S - 14 : Delegate/MulticastDelegate (Temsilciler) nasıl bildirilir?

C - 14 : C# ta temsilci bildirimi için sadece bir parametreye ihtiyacımız vardır : metot adresi. Diğer dillerden farklı olarak C# ta metodun adresi aynı zamanda bu metodun hangi nesne üzerinden de çağrılacağını tutabilir, diğer dillerde ise temsilcilerin temsil ettiği metodu çağırabilmek için ayrıca nesnelere ihtiyaç duyulur. Örneğin System.Threading.ThreadStart() metodunun kullanımına bakalım.

```
Foo MyFoo = new Foo();  
ThreadStart del = new ThreadStart(MyFoo.Baz);
```

Bu, static ve instance metotlarının aynı sentaks ile çağrılabilceğini göstermektedir.

S - 15 : Yaptığım windows pencere uygulamasını her çalıştırdığımda neden pop up şeklinde konsol ekranı gösteriliyor.

C - 15 : Proje ayarlarında "Target Type" özelliğinin Console Application yerine Windows Application olduğuna emin olun. Eğer komut satırı derleyicisini kullanıyorsanız /target:exe argümanı yerine /target:winexe argümanını kullanın.

S - 16 : Gereksiz çöp toplayıcısı(Garbage Collection) zorla çağırmanın bir yolu var mı?

C - 16 : Evet; Bütün referansları null değer atayın ve System.GC.Collect() statik metodunu çağırın.

Yıkılması(destruct) gereken nesneleriniz var ve GC nin bunu yapmadığını düşünüyorsanız nesneleri null değere atayarak onların sonlandırıcı metotlarının çağrılmasını sağlayın ver ardından System.GC.RunFinalizers() metodunu çağırın

S - 17 : C#, C dilindeki makroları destekliyormu?

C - 17 : Hayır, C# ta makro yoktur.

__LINE__ ve __FILE__ gibi C dilinde önceden tanımlanmış bazı makroların System.Diagnostics isim alanındaki StackTrace ve StackFrame gibi COM+ ile ilgili sınıflardan elde edilebileceğini unutmayın. Fakat bunlar sadece Debug moddaki derleme için çalışacaktır.

S - 18 : C# derleyicisine bazı dll leri referans vermememe rağmen neden kendisi referans verir.

C - 18 : "csc.rsp" dosyasında bulunan bütün assembly lere C# derleyicisi otomatik olarak referans verir. Bu dosyanın içerdiği assembly leri /r argümanı ile belirtmek zorunda değilsiniz. csc.rsp dosyasının kullanımını komut satırından /noconfig argümanını belirterek engelleyebilirsiniz.

Not : Visual Studio IDE si hiç bir zaman csc.rsp dosyasını kullanmaz.

S - 19 : Delegate/MulticastDelegate (Temsilciler) nasıl bildirilir?

C - 19 : Aşağıda DllImport niteliğinin kullanımına bir örnek verilmiştir.

```
using System.Runtime.InteropServices;
```

```

class C
{
    [DllImport("user32.dll")]
    public static extern int MessageBoxA(int h, string m, string c, int type);

    public static int Main()
    {
        return MessageBoxA(0, "Hello World!", "Caption", 0);
    }
}

```

Yukarıdaki örnek kod yönetilmeyen(unmanaged) DLL deki doğal(native) bir fonksiyonu C# ta bildirmek için minimum gereksinimleri gösterir. C.MessageBoxA() metodu static ve extern sözcükleri ile bildirilmiş, DllImport niteliği ile bu metodun user32.dll dosyasında MessageBoxA ismiyle uygulanmış olduğu belirtilmektedir.

S - 20 : COM+ runtime'ında tanımlanan bir arayüzü uygulamaya çalışıyorum ancak "public * Object GetObject{...}" çalışmıyor gibi. Ne yapmalıyım?

C - 20 : Managed C++'ta "Object * GetObject()" (object türünden gösterici) sentaksı geçerlidir. C# ta ise "public Object GetObject()" biçiminde kullanmak yeterlidir.

S - 21 : C# şablon(template) yapılarını destekliyormu?

C - 21 : Hayır, fakat bir tür şablon olan generics yapılarının C# diline eklenilmesi planlanmaktadır. Bu türler sentaks olarak şablonlara benzerler fakat derleme zamanı yerine çalışma zamanında oluşturulurlar. Bu türlerle ilgili detaylı bilgi için [tıklayın](#).

S - 22 : Item özelliğini kullandığımda neden CS0117 hatası almaktayım?

C - 22 : C# özellikleri destekler ancak Item özelliğinin sınıflar için özel anlamı vardır. Item özelliği aslında varsayılan indeksleyici olarak yer alır. Bu imkanı C# ta elde etmek için Item sözcüğünü atmak yeterlidir. Aşağıda örnek program gösterilmiştir.

```

using System;
using System.Collections;

class Test
{
    public static void Main()
    {
        ArrayList al = new ArrayList();
        al.Add( new Test() );
        al.Add( new Test() );
        Console.WriteLine("First Element is {0}", al[0]);
    }
}

```

WriteLine metodunda .Items[0] 'ın kullanılmadığına dikkat edin.

S - 23 : Herhangi bir fonksiyonumu "out int" parametresi alacak şekilde tasarlmaya çalışıyorum. Bu metoda göndereceğim int değişkenini nasıl bildirmeliyim?

C - 23 : Değişken bildirimi int türünden yapmalısınız fakat bu değişkeni fonksiyona parametre olarak gönderirken aşağıdaki gibi "out" anahtar sözcüğünü de kullanmalısınız.

```
int i;  
foo(out i);  
foo metodu aşağıdaki gibi bildirilmiştir.  
[return-type] foo(out int o) { }
```

S - 24 : C++'taki referanslara benzer bir yapı C#' ta varmıdır? (Ör : void foo(int &x) gibi)

C - 24 : C#'ta bunun karşılığı ref parametreleridir.

```
class Test  
{  
    public void foo(ref  
int i)  
    {  
        i = 1;  
    }  
  
    public void bar()  
    {  
        int a = 0;  
        foo(ref a);  
        if (a == 1)  
  
Console.WriteLine("It  
worked");  
    }  
  
    public static void  
Main() {}  
}
```

Not: Metot çağrımında da ref sözcüğünün kullanıldığına dikkat edin!

S - 25 : C#'ta *inout* argümanları nasıl bildirilir?

C - 25 : inout'un C# taki karşılığı **ref**'tir. Örneğin :

```
public void MyMethod (ref String str1, out String str2)  
{  
    ...  
}
```


Bu metot aşağıdaki biçimde çağrılmalıdır.

```
String s1;  
String s2;  
s1 = "Hello";  
MyMethod(ref s1, out s2);  
Console.WriteLine(s1);  
Console.WriteLine(s2);
```

Not : Hem metot çağırımı hemde metot bildirimi sırasında ref sözcüğünün kullanıldığına dikkat edin.

S - 26 : Yıkıcı metotlar(destructors) ve GC C#'ta ne şekilde çalışır?

C - 26 : C# ta sonlandırıcı metotlar vardır ve kullanımı aşağıdaki gibidir. (Bu sonlandırıcı metotlar C++ taki yıkıcı metotlara benzer, tek farkı çağrılacağı garanti altına alınmamıştır.)

```
class C  
{  
    ~C()  
    {  
        // your code  
    }  
  
    public static void Main() {}  
}
```

Bu metotlar object.Finalize() metodunu aşırı yüklerler ve GC nesneyi yok ederken bu metodu kullanır.

S - 27 : Derleme sırasında neden "CS5001: does not have an entry point defined - tanımlanmış giriş noktası yok- " hatasını alıyorum?

C - 27 : Bu hata en çok Main metodunu main şeklinde yazdığınızda karşınıza çıkar. Giriş noktası olan bu Main metodunun bildirimi aşağıdaki gibi olmalıdır :

```
class test  
{  
    static void Main(string[] args) {}  
}
```

S - 28 : Visual J++ ta "synchronized" olarak bildirilen metotları C# diline nasıl taşıyım?

C - 28 : Orjinal Visual J++ kodu:

```
public synchronized void Run()  
{
```

```
// function body  
}
```

C# diline taşınmış hali

```
class C  
{  
    public void Run()  
    {  
        lock(this)  
        {  
            // function body  
        }  
    }  
  
    public static void Main() {}  
}
```

S - 29 : Kanal(thread) senkronizasyonu(Object.Wait, Notift ve CriticalSection) C#'ta nasıl sağlanır?

C - 29 : lock ile işaretlemiş bloklar bu işe yarar :

```
lock(obj)  
{  
    // code  
}
```

kod parçasının karşılığı

```
try  
{  
    CriticalSection.Enter(obj);  
    // code  
}  
finally  
{  
    CriticalSection.Exit(obj);  
}
```

S - 30 : Statik yapıcı metotların sentaksı nasıldır?

C - 30 : Aşağıda MyClass adlı sınıfın statik yapılandırıcısının bildirimi gösterilmiştir.

```
class MyClass  
{  
    static MyClass()  
    {  
        // initialize static variables here  
    }  
}
```

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

S - 31 : Bir özelliğin get ve set bloklarını farklı erişim belirleyicileri ile bildirmek mümkünmüdür?

C - 31 : Hayır, bir özelliğin belirtilen erişim belirleyicisi aynı zamanda hem get hem de set bloklarının erişim belirleyicisidir. Fakat yapmak istediğinizi muhtemelen sadece get bloğu olan yani readonly olarak bildirip set bloğunu private yada internal olan bir metot yapacak şekilde gerçekleştirebilirsiniz.

S - 32 : Tek bir assembly de çoklu dil desteğini nasıl sağlayabilirim?

C - 32 : Bu şu an için Visual Studio.NET tarafından desteklenen bir özellik değildir.

S - 33 : C# dizi türünden olan özellikleri destekliyor mu?

C - 33 : Evet, aşağıda buna bir örnek verilmiştir:

```
using System;  
  
class Class1  
{  
    private string[] MyField;  
  
    public string[] MyProperty  
    {  
        get { return MyField; }  
        set { MyField = value; }  
    }  
}  
  
class MainClass  
{  
    public static int Main(string[] args)  
    {  
        Class1 c = new Class1();  
  
        string[] arr = new string[] { "apple", "banana" };  
        c.MyProperty = arr;  
        Console.WriteLine(c.MyProperty[0]); // "apple"  
  
        return 0;  
    }  
}
```

S - 34 : Birden fazla assembly ile çoklu dil desteği sağlanabilir mi?

C - 34 : Malesef şu an için IDE de bu desteklenmiyor. Bunu yapabilmek için komut satırından projenizi /target:module argümanı ile derleyip modüllere ayırmanız gerekir. Ve oluşturduğunuz bu modülleri birleştirmek için yine komut satırından al(alink) aracını çalıştırarak bu modüllerin birleştirilmesini sağlayın.

S - 35 : COM nesnelere erişmek için opsiyonel olan parametreleri nasıl simule edebilirim?

C - 35 : Opsiyonel parametreler için System.Reflection altında bulunan Missing sınıfı kullanılır. Her bir parametre için Missing.Value değeri kullanılabilir.

S - 36 : C++'taki varsayılan metot argümanlarının bir karşılığı C#'ta var mı?

C - 36 : Varsayılan argüman desteği yoktur ancak aynı etkiyi metot yükleme ile rahatlıkla yapabilirsiniz.

Bu problem için metot yüklemeyi tercih etmemizin sebebi ileriki zamanlarda kaynak kodu yeniden derlemeden varsayılan argümanı değiştirme imkanı vermesidir. C++ taki varsayılan argümanlar derşenmiş kodun içine gömüldüğü için sonradan bu argümanı kaynak kodu derlemeden değiştirmek mümkün değildir.

S - 36 : İç içe geçmiş bloklarda yada döngülerde hangi bloğun sonlandırıldığını belirtmek için kolay bir yol varmıdır?

C - 36 : Bu işin en kolay yolu goto atlama deyimini aşağıdaki gibi kullanmaktır.

```
using System;
class BreakExample
{
    public static void Main(String[] args)
    {
        for(int i=0; i<3; i++)
        {
            Console.WriteLine("Pass {0}: ", i);
            for( int j=0 ; j<100 ; j++ )
            {
                if ( j == 10) goto done;
                Console.WriteLine("{0} ", j);
            }
            Console.WriteLine("This will not print");
        }
        done:
        Console.WriteLine("Loops complete.");
    }
}
```

S - 37 : C#'ta deterministik sonlandırmayı nasıl sağlayabilirim.

C - 37 : GC mekanizması gerçek anlamda deterministik yapıda değildir. Yani ne zaman gereksiz nesnelerin toplanacağı kesin olarak belirlenmemiştir. Bu yüzden kritik kaynaklara sahip olan nesneleri tasarlamak için IDisposable arayüzünü uygulamış sınıflar tasarlamakta fayda vardır. Aşağıdaki tasarım deseni sınıf için ayrılan kaynağın blok sonunda bırakılacağını bildirmektedir.

```
using(FileStream myFile = File.Open(@"c:\temp\test.txt", FileMode.Open))
{
    int fileOffset = 0;

    while(fileOffset < myFile.Length)
    {
        Console.Write((char)myFile.ReadByte());
        fileOffset++;
    }
}
```

Akış, using bloğunun sonuna geldiğinde myFile nesnesi üzerinden Dispose() metodu çağrılacaktır. Nesneleri bu şekilde using ile kullanabilmek için ilgili sınıfın IDisposable arayüzünü uygulaması gerektiğini unutmayın.

S - 38 : C#'ta metotlar için değişken sayıda argüman (vararg) desteği varmıdır?

C - 38 : params anahtar sözcüğü bir metodun değişken sayıda parametre alabileceğini belirtir. Metot bildiriminde params anahtar sözcüğünden sonra herhangi bir metot parametresi bildirilemez. Ve bir metot için sadece bir tane params anahtar sözcüğünün kullanımına izin verimiştir. Aşağıda params'ın kullanımına bir örnek verilmiştir.

```
using System;

public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for ( int i = 0 ; i < list.Length ; i++ )
            Console.WriteLine(list[i]);
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for ( int i = 0 ; i < list.Length ; i++ )
            Console.WriteLine((object)list[i]);
        Console.WriteLine();
    }

    public static void Main()
    {
        UseParams(1, 2, 3);
        UseParams2(1, 'a', "test");

        int[] myarray = new int[3] {10,11,12};
        UseParams(myarray);
    }
}
```

```
}  
}
```

Çıktı

```
1  
2  
3  
  
1  
a  
test  
  
10  
11  
12
```

S - 39 : C#'ta string türünden bir değişkeni int türüne nasıl dönüştürebilirim?

C - 39 : Aşağıda bu duruma bir örnek verilmiştir.

```
using System;  
  
class StringToInt  
{  
    public static void Main()  
    {  
        String s = "105";  
        int x =  
Convert.ToInt32(s);  
        Console.WriteLine(x);  
    }  
}
```

S - 40 : C# ile yazılmış uygulamalardan çıkmak için exit() gibi bir fonksiyon var mıdır?

C - 40 : Evet, uygulamadan çıkmak için System.Environment.Exit(int exitCode) metodunu yada uygulamanız bir windows uygulaması ise Application.Exit() metodlarını kullanabilirsiniz.

S - 41 : Bütün assembly için özel bir nitelik nasıl belirtilir?

C - 41 : Global nitelikler en tepedeki using deyiminden sonra ve herhangi bir sınıf yada isim alanı bildiriminden önce yapılmalıdır. Örnek :

```
using System;
```

```
[assembly : MyAttributeClass]
```

```
class X {}
```

Not : IDE tarafından yaratılan projelerde bu nitelik bildirimleri AssemblyInfo.cs dosyasında yapılmıştır.

S - 42 : C# ile yazılmış kodu klasik COM istemcilerinin kullanımına sunmak için nasıl kayıt(register) etmeliyim?

C - 42 : regasm aracını kullanarak eğer gerekliyse type library leri oluşturun. Sınıf windows registry ye kayıt edildikten sonra bu sınıfa COM istemcileri tarafından sanki bir COM bileşeniymiş gibi erişilebilir.

S - 43 : Birden fazla derlenecek kaynak kod aynı anda derlendiğinde, çalıştırılabilir dosyanın ismi nasıl belirleniyor?

C - 43 : Çalıştırılabilir dosyanın adı Main metodunun bulunduğu kaynak dosyanın adı ile aynı olur. Komut satırından /out argümanı ile çalıştırılabilir dosyanın adını kendiniz de belirleyebilirsiniz. Örneğin:

```
C:\csc /out:Uygulama.exe dosya1.cs dosya2.cs
```

komutu çalıştırılabilir dosyanın adını Uygulama.exe yapacaktır.

S - 44 : C#'ta String türünden nesneler nasıl karşılaştırılır?

C - 44 : Geçmişte iki stringi == ve != operatörleri ile karşılaştırmak için ToString() metodu kullanılmalıydı. Şu anda ise yine bu yöntem geçerli olmasına rağmen string ler == ve != operatörü ile karşılaştırıldıklarında değişkenlerin referansları yerine onların değerleri karşılaştırılır.

Eğer gerçekten string değişkenlerinin referanslarını karşılaştırmak istersek aşağıdaki kodu kullanabiliriz.

```
if ((object) str1 == (object) str2) { ... }
```

Aşağıda string değişkenlerinin nasıl çalıştığına bir örnek verilmiştir.

```
using System;
public class StringTest
{
    public static void Main(string[] args)
    {
        Object nullObj = null;
        Object realObj = new StringTest();
        int i = 10;
    }
}
```

```

        Console.WriteLine("Null Object is [" + nullObj + "]\n" +
            "Real Object is [" + realObj + "]\n" +
            "i is [" + i + "]\n");

        // Show string equality operators
        string str1 = "foo";
        string str2 = "bar";
        string str3 = "bar";

        Console.WriteLine("{0} == {1} ? {2}", str1, str2, str1 == str2 );
        Console.WriteLine("{0} == {1} ? {2}", str2, str3, str2 == str3 );
    }
}

```

Çıktı

```

Null Object is []
Real Object is [StringTest]
i is [10]

foo == bar ? False
bar == bar ? True

```

S - 45 : C++'taki typedef komutunun yaptığı gibi farklı türler için takma isimleri kullanılabilmirmi?

C - 45 : Tam olarak değil ama bir dosyada using deyimini aşağıdaki gibi kullanarak herhangi bir türe takma isim verebilirsiniz.

```

using System;
using Integer =
System.Int32; // takma
ismi

```

using deyiminin kullanımı hakkında ayrıntılı bilgi için C# standartlarını incelyin.

S - 46 : C#'ta sınıf ile yapı arasındaki farklar nelerdir?

C - 46 : Sınıflar ve yapılar arasındaki farkların listesi oldukça fazladır. Yapılar sınıflar gibi arayüzleri uygulayabilir ve aynı üye elemanlara sahip olabilirler. Yapılar, sınıflardan bir çok önemli noktada ayrılır; yapılar değer tipleri sınıflar ise referans tipleridir ve türetme yapılar için desteklenmez. Yapılar stack bellek bölgesinde saklanır. Dikkatli programcılar bazen yapıları kullanarak uygulamanın performansını artırabilirler. Mesela Point yapısı için sınıf yerine yapı kullanmak çalışma zamanında tahsis edilen bellek açısından oldukça faydalıdır.

S - 47 : Bir karakterin ASCII kodunu nasıl elde edebilirim?

C - 47 : char türden değişkeni int türüne dönüştürürseniz karakterin ASCII kodunu elde edersiniz.

```
char c = 'f';  
System.Console.WriteLine((int)c);
```

yada bir string deki herhangi bir karakter için

```
System.Console.WriteLine((int)s[3]);
```

kodunu kullanabilirsiniz.

.NET kütüphanesindeki Convert ve Encoding sınıflarının yardımıyla da bu işlemi gerçekleştirmek mümkündür.

S - 48 : Versiyonlama bakış açısıyla arayüz türetmenin sınıf türetmeye karşı getirileri nelerdir?

C - 48 : Versiyonlama bakış açısıyla arayüz türetmenin sınıf türetmesine göre daha az esnek olduğunu söylemek mümkündür.

Sınıflarda farklı versiyonlara yeni üye elemanlar örneğin yeni metod eklemeniz mümkündür. Bu metod abstract olmadığı sürece yeni türetilen sınıflar bu metodun fonksiyonallitesine sahip olacaktır. Arayüzler uygulanmış kodların türetilmesini desteklemediği için bu durum arayüzler için geçerli değildir. Bir arayüze yeni bir metod eklemek sınıflara yeni bir abstract metod gibidir. Bu arayüzü uygulayan bir sınıf bu metodu aynen uygulayıp kendine göre anlamlandırmalıdır.

S - 49 : C# koduna inline assembly yada IL kodu yazmak mümkün müdür?

C - 49 : Hayır.

S - 50 : Bir metodu yada herhangi bir üye elemanının kullanımını sadece belirli bir isim alanı için sınırlayabiliyor muyuz?

C - 50 : İsim alanları için bir kısıtlama yapılamaz çünkü isim alanları koruma amaçlı olarak kullanılmamaktadır ancak internal erişim belirleyicisi ile bir türün sadece ilgili assembly dosyası içinde kullanılabilecek durumuna getirebiliriz.

S - 51 : try bloğu içerisinde return ile metodu sonlandırırsam finally bloğundaki kodlar çalıştırılır mı?

C - 51 : Evet, finally bloğundaki kodlar siz return ile metodu sonlandırırsanız da try bloğunun sonuna gelsenizde her zaman çalışacaktır. Örneğin :

```

using System;
class main
{
    public static void
Main()
    {
        try
        {
            Console.WriteLine
("In Try block");
            return;
        }
        finally
        {
            Console.WriteLine
("In Finally block");
        }
    }
}

```

programında hem "In try block" hemde "In Finally block" yazısı ekrana yazdırılacaktır. Performans açısından return sözcüğünü try bloğunda yada finally bloğundan sonra kullanmanın bir farkı yoktur. Derleyici yukarıdaki durumda return ifadesinin sanki finally bloğunun dışındaymış gibi davranır. Eğer yukarıda olduğu gibi return deyimi herhangi bir ifade ile kullanılmıyorsa her iki durumdada IL olarak üretilen kodlar aynıdır. Fakat eğer return deyimi bir ifade ile kullanılıyorsa try bloğundaki return ifadesinde ekstradan store ve load deyimlerinin IL de olacağı açıktır.

S - 52 : C# try-catch-finally vloklerini destekliyormu?

C - 52 : Evet destekliyor, aşağıda bu blokların kullanımına bir örnek verilmiştir.

```

using System;
public class TryTest
{
    static void Main()
    {
        try
        {
            Console.WriteLine("In Try block");
            throw new ArgumentException();
        }
        catch(ArgumentException n1)
        {
            Console.WriteLine("Catch Block");
        }
        finally
        {
            Console.WriteLine("Finally Block");
        }
    }
}

```

Çıktı

In Try Block
Catch Block
Finally Block

S - 53 : Statik indeksleyici tanımlamak mümkünmüdür?

C - 53 : Hayır. Statik indeksleyici tanımlamaya izin verilmemiştir.

S - 54 : Derleyiciyi /optimize+ argümanı ile çalıştırdığımızda ne gibi optimizasyonlar yapar.

C - 54 : C# derleyicisini yazan takımın bu soruya verdiği cevap:

Kullanılmayan lokal değişkenleri atıyoruz. (örnek, hiç okunmayan lokal değişkenler - kendisine değer verilmiş olsa bile-).

Hiç bir şekilde erişilemeyecek(unreachable) kodları atıyoruz.

try bloğu boş olan try/catch bloklarını kaldırıyoruz.

try bloğu boş olan try/finally bloklarını kaldırıyoruz.(normal koda çevrilir)

finally bloğu boş olan try/finally bloklarını kaldırıyoruz.(normal koda çevrilir)

Dallanmalarda diğer dallanmaları optimize ediyoruz. Örneğin

```
gotoif A, lab1  
goto lab2:  
lab1:
```

kodu

```
gotoif !A, lab2  
lab1:
```

koduna dönüştürülür.

We optimize branches to ret, branches to next instruction, branches to branches.
Dallanmaları "ret"e, "next instruction" lara veya diğer "branch" lara dönüştürüyoruz.

S - 55 : C# ile registry'ye nasıl erişebilirim?

C - 55 : Microsoft.Win32 isim alanındaki Registry ve Registry sınıflarını kullanarak bu alana erişmek mümkündür. Aşağıdaki program bir registry anahtarını okuyup değerini yazdırmaktadır.

```
using System;  
using Microsoft.Win32;
```

```

class regTest
{
    public static void Main(String[] args)
    {
        RegistryKey regKey;
        Object value;
        regKey = Registry.LocalMachine;
        regKey =
regKey.OpenSubKey("HARDWARE\\DESCRIPTION\\System\\CentralProcessor\\0");
        value = regKey.GetValue("VendorIdentifier");
        Console.WriteLine("The central processor of this machine is: {0}.", value);
    }
}

```

S - 56 : C# global sabitleri tanımlamak için #define komutunu destekler mi?

C - 56 : Hayır. Eğer C dilindeki aşağıdaki koda benzer bir kullanım elde etmek istiyorsanız

```
#define A 1
```

bu kodu kullanabilirsiniz.

```

class
MyConstants
{
    public
const int A =
1;
}

```

Böylece A makrosuna her erişmek istediğinizde MyConstants.A şeklinde bir kullanıma sahip olursunuz.

MyConstants.A şeklindeki kullanım ile 1 sayısının kullanımı arasında bir fark yoktur. Yani aynı kod üretilecektir.

S - 57 : Yeni bir proses çalıştırıp bu prosesin sonlanmasını nasıl bekleyebilirim?

C - 57 : Aşağıdaki kod argüman olarak verilen çalıştırılabilir programı çalıştır ve çalışan bu programın kapatılması için bekler.

```

using System;
using System.Diagnostics;

public class ProcessTest {
    public static void Main(string[] args) {
        Process p = Process.Start(args[0]);
        p.WaitForExit();
    }
}

```

```
        Console.WriteLine(args[0] + " exited.");  
    }  
}
```

S - 58: Bir metod obsolete olarak nasıl işaretlenir?

C - 58 : using System; yazdığınızı varsayarak

```
[Obsolete]  
public int Foo() {...}
```

yada

```
[Obsolete("Bu mesaj metodun neden Obsolete olduğunu açıklar.")]  
public int Foo() {...}
```

Not: Obsolete kelimesindeki O harfi büyüktür.

S - 59: using deyimini kaynak koduma eklememe rağmen derleyici tanımlanmamış türlerin bulunduğunu söylüyor. Nerede yanlış yapıyorum acaba?

C - 59 : Büyük bir ihtimalle isim alanının bulunmuş assembly dosyasını referans vermeyi unutmuşsunuzdur. using deyimi sadece bir sentaksdır. Assembly nin fiziksel olarak konumunu da ayrıca belirtmeniz gerekir. IDE yi kullanarak project menüsünden add reference seçeneği seçip istediğiniz assembly ye referans verebilirsiniz. Komut satırı derleyicisi kullanıyorsanız /r argümanını kullanmalısınız.

S - 60 : Basit bir çok kanallı uygulama için örnek kod var mı?

C - 60 : Evet. örnek :

```
using System;  
using System.Threading;  
  
class ThreadTest  
{  
    public void runme()  
    {  
        Console.WriteLine("Runme Called");  
    }  
  
    public static void Main(String[] args)  
    {  
        ThreadTest b = new ThreadTest();  
        Thread t = new Thread(new ThreadStart(b.runme));  
        t.Start();  
    }  
}
```

S - 61: Override edilmiş bir metodun temel sınıftaki versiyonunu nasıl çağırabilirim?

C - 61 : Aşağıdaki gibi base anahtar sözcüğünün kullanarak çağırabilirsiniz.

```
public class MyBase
{
    public virtual void meth()
    {
        System.Console.WriteLine("Test");
    }
}

public class MyDerived : MyBase
{
    public override void meth()
    {
        System.Console.WriteLine("Test2");
        base.meth();
    }

    public static void Main()
    {
        MyDerived md = new MyDerived();
        md.meth();
    }
}
```

S - 62: C# geliştiricilerini düzenli ifadeler(regex) desteği sunulmuş mudur?

C - 62 : Evet, .NET sınıf kütüphanesi programcılara düzenli ifadelerle çalışmak için System.Text.RegularExpressions isim alanında bir takım sınıflar sağlamaktadır.

S - 63 : C# ile yazmış olduğum uygulamayı çalıştırdığımda neden güvenlik hatası alıyorum?

C - 63 : Bazı güvenlik hataları ağ üzerinde paylaşıma açılmış kaynaklar üzerinde çalışırken alınır. Roaming profilleri, mapped diskler gibi kaynaklar üzerinde çalışmayan bazı sınıflar vardır. Bunun olup olmadığını kontrol etmek için uygulamanızı lokal diskinize alıp yeniden çalıştırmayı deneyin.

Bu tür durumlarda genellikle System.Security.SecurityException istisnai durumu meydana gelir.

Bu tür sorunların üstesinden gelmek için caspol.exe aracı yardımıyla intranet için güvenlik policy nizi codegroup 1.2 ye ayarlayabilirsiniz.

S - 64: try-catch bloklarında faaliyet alanı (scope) problemlerinin üstesinden nasıl gelirim?

C - 64 : try bloğu içinde yarattığınız nesneye catch bloğu içinden erişemezsiniz çünkü try bloğunun sonunda ilgili nesnenin faaliyet alanı bitecektir. Bunun önüne geçmek için aşağıdaki kod bloğu kullanılabilir.

```
Connection conn = null;
try
{
    conn = new Connection();
    conn.Open();
}
finally
{
    if (conn != null) conn.Close();
}
```

try bloğundan önde değişkeni null değere atamakla derleyicinin CS0165 (Use of possibly unassigned local variable 'conn') hatasını vermesini engellemiş oluruz.

S - 65: .NET geliştirme ortamında regsvr32 ve regsvr32 /u komutlarının karşılığı nedir?

C - 65 : RegAsm aracını kullanabilirsiniz. .NET SDK içinde bu aracın kullanımı hakkında detaylı bilgiyi bulmanız mümkündür.

S - 65: C#, parametreleri özellikleri destekliyor mu?

C - 65 : Hayır, fakat dilin temel yapısında indeksleyici diye ayrı bir kavram vardır.

Bir indeksleyici bir türün dizi gibi indek operatörü ile kullanılabilmesini sağlar. Kısaca özellikler field benzeri erişimi indeksleyiciler ise dizi benzeri erişimi sağlarlar.

Örnek olması açısından daha önce yazdığımız Stack sınıfını düşünün. Bu sınıfı tasarlayan sınıfın üye elemanlarına bir dizi gibi erişilmesini isteyebilir ve böylece gereksiz Pop ve Push çağrımları yapılmamış olur. Yani stack bir bağlı liste gibi tasarlanmış olmasına rağmen bir dizi gibi kullanılabilir.

İndeksleyici bildirimi özellik bildirimine benzemektedir. İki bildirim arasındaki en büyük fark indeksleyicilerin isimlerinin olmamasıdır.(indeksleyici bildiriminde is yerine this anahtar sözcüğü kullanılır.) Diğer bir fark ise indeksleyicilerin indeks parametresi alabilmesidir. Bu indeks parametresi köşeli parantezler içinde yazılır.

C# ile XMLQuery Örneği

Bu yazımda sizlere, XML dokümanlarında nasıl sorgulama yapabileceğimizi basit bir örnek ile anlatmaya çalışacağım.

Dilerseniz hemen uygulamaya geçelim. Yeni bir Asp.Net Web Application açın ve adını XmlQuery olarak ayarlayın. Ardından projeye yeni bir xml doküman ekleyin ve adını Kayitlar.Xml olarak ayarlayın. Xml dokümanın yapısını aşağıdaki gibi düzenleyin.

```
<?xml version="1.0" encoding="utf-8" ?>
<Kayitlar>
  <Kayit id="1" tip="A">
    <Adi>Tolga</Adi>
    < Soyadi>Güler</Soyadi>
    <Numarası>1544747</Numarası>
  </Kayit>
  <Kayit id="2" tip="B">
    <Adi>Utku</Adi>
    <Soyadi>Selen</Soyadi>
    <Numarası>4577877</Numarası>
  </Kayit>
  <Kayit id="3" tip="B">
    <Adi>Murat</Adi>
    <Soyadi>Kula</Soyadi>
    <Numarası>8787878</Numarası>
  </Kayit>
```



```
<Kayit id="4" tip="C">
  <Adi>Argun</Adi>
  <Soyadi>Çelikten</Soyadi>
  <Numarası>7454621</Numarası>
</Kayit>
</Kayitlar>
```

WebForm1.aspx.cs dosyasına aşağıdaki kodları ekleyin.

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;
using System.Xml;
namespace XmlQuery
{
    public class WebForm1 : System.Web.UI.Page
    {
        private void Page_Load(object sender, System.EventArgs e)
        {
            XmlNodeList Isimler,Isimler2,Isimler3,Isimler4,Isimler5;
            //XmlNodeList türünden değişkenlerimizi tanımlıyoruz.

            XmlTextReader rdr = new XmlTextReader("http://localhost/XmlQuery/kayitlar.xml");
            // XmlTextReader sınıfı yardımı ile xml dökümanına erişiyoruz.
            XmlDocument MyXmlDoc = new XmlDocument();
            MyXmlDoc.Load(rdr);
            //XmlDocument sınıfını xml dökümanı üzerinde işlem yapabilmek için kullanıyoruz

            // Xml domüanından id si 1 olan isimleri seçmek için
            Isimler = MyXmlDoc.SelectNodes("/Kayitlar/Kayit[@id='1']/Adi");
            /* XmlDocument.SelectNodes metoduna parametre olarak verdiğimiz XPATH
            e dikkat edin.
            */
            for(int i = 0;i < Isimler.Count;i++)
                Response.Write(Isimler.Item(i).InnerXml.ToString()+"<br>");
            // Sonuç "Tolga" olacaktır.

            // id si 1 veya 2 olan kayıtlar için
            Isimler2 = MyXmlDoc.SelectNodes("/Kayitlar/Kayit[@id='1' or @id='2']/Adi");
            for(int i = 0;i < Isimler2.Count;i++)
                Response.Write(Isimler2.Item(i).InnerXml.ToString()+"<br>");
            // Sonuç "Tolga" ve "Utku" olacaktır.
```

```

// id si 1 ve tipi A olan kayıtlar için
Isimler3 = MyXmlDoc.SelectNodes("/Kayitlar/Kayit[@id='1' and @tip='A']/Adi");
for(int i = 0; i < Isimler3.Count; i++)
    Response.Write(Isimler3.Item(i).InnerXml.ToString()+"<br>");
// Sonuç "Tolga" olacaktır.

// tipi B olan kayıtların adının ilk iki harfi "Ut" olanlar
Isimler4 = MyXmlDoc.SelectNodes("/Kayitlar/Kayit[@tip='B']/Adi[substring(.,1,2)='Ut']");
for(int i = 0; i < Isimler4.Count; i++)
    Response.Write(Isimler4.Item(i).InnerXml.ToString()+"<br>");
// Sonuç "Utku" olacaktır.

// tipi B olan kayıtların adında "ura" geçenler
Isimler5 = MyXmlDoc.SelectNodes("/Kayitlar/Kayit[@tip='B']/Adi[contains(.,'ura')]");
for(int i = 0; i < Isimler5.Count; i++)
    Response.Write(Isimler5.Item(i).InnerXml.ToString()+"<br>");
// Sonuç "Murat" olacaktır.

}

}

}

```

Siz örnekleri istediğiniz gibi geliştirip çoğaltabilirsiniz.

2003 – 2005 Microsoft Yazılım Geliştirme Araçları Yol Haritası – 1

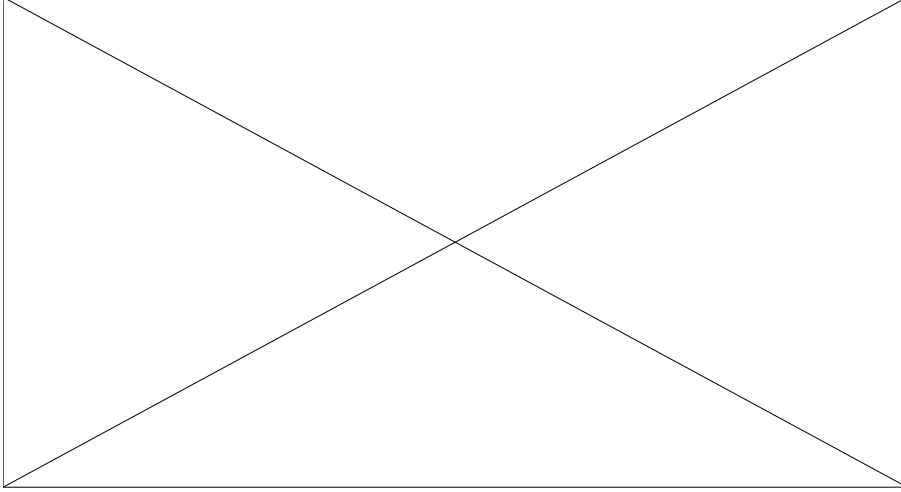
Şubat 2002’de Visual StudioNet ve dotNET Platformu dünyadaki tüm yazılım geliştiricilerin hizmetine sunuldu; bu önemli olay sayesinde programcılar çok değişik alanlarda program geliştirme işlerini dotNET platformu ve Visual StudioNET ile yapabilir hale geldi. Visual Studio.NET 2003 ile programcılar müşterilerine başarıları kanıtlanmış, yüksek performanslı ve güvenilir yazılımlar geliştirmeye devam ediyorlar.

İş dünyasındaki değişikliklerle birlikte ihtiyaç duyulan yazılımların da gelişmesi ve değişmesi gerekiyor. Böyle bir ortamda Microsoft kendisinin yazılım geliştirme araçlarını kullanan geliştiricilere devrim niteliğinde ve iş dünyasının değişen ihtiyaçlarına en kısa ve en iyi çözümlerini üretecek yazılım geliştirme araçlarını sunmaya devam ediyor. Kurumların gelecekteki yazılım ihtiyaçlarının planlamasını yaparken onlara yardımcı olmak amacıyla Microsoft bu yol haritasını sunmaktadır. Bu belge özellikle şu ürünler üzerinde yoğunlaşmıştır:

- **Microsoft Ofis 2003 için Visual Studio araçları:** Şu anda beta aşamasında olan bu teknoloji sayesinde, Microsoft Office Word 2003 ve Microsoft Excel 2003’ü .Net ortamında programlayabileceğiz.
- **“Whidbey” kod adlı Visual Studio 2004:** Visual Studio.NET ve .NET platformunun bu versiyonunda birçok yenilikler ve değişikliklerle geliyor. Başlıca yenilikler sınıf kütüphanesinde, ortak dil çalışma (CLR) kısmında, programlama dillerinde ve Visual studio.NET’in arayüzünde (IDE) olacaktır. Ayrıca SQL Server’ın

yeni versiyonu olan SQL Server "Yukon" ile büyük bir entegrasyon sağlanacaktır. Bu sayede C# ve Visual Basic.Net ile saklı yordamları (stored procedures) yazıp **Yukon** üzerinde çalıştırabileceğiz.

- **"Orcas" kod adlı Visual Studio 2005:** Bu versiyonda ise **"Longhorn "** isimli Windows işletim sistemiyle daha iyi entegrasyon ve programlama alt yapısı sağlanacak.



Microsoft yazılım geliştirme araçları her zaman Windows platformunun en son özelliklerine erişmeyi ve onları programlamayı programcılara sunmuştur. Yukarıda da görüldüğü gibi Microsoft bu geleneği sürdürmeye devam edecektir. Bu bağlamda Microsoft Ofis Sistem 2003'ü, SQL Server Yukon'u ve Windows işletim sistemlerini programlamak için bir çok kolaylıklara sahip olacağız biz yazılım geliştiriciler olarak.

Microsoft Ofis 2003 için Visual Studio Araçları

"Yazılım geliştiriciler hem Visual Studio hem de Microsoft'un başarısındaki öncül güç olmuşlardır."

- Eric Rudder, Sunucu ve Araçlardan sorumlu Genel başkan yardımcısı.

Visual Studio 2003'ün hemen ardından Microsoft, Ofis 2003 için Visual Studio araçlarını piyasaya sürdü. Bu yeni teknoloji sayesinde .NET platformundan yönetilen kod sayesinde Microsoft Word 2003 ve Microsoft Excel 2003 için kod yazılabilecek. Tıpkı VBA ve COM tabanlı otomasyon projeleri gibi. Microsoft Ofis 2003 için Visual Studio Araçları biz yazılımcılara şu önemli avantajları da getiriyor:

- **Tanıdık programlama deneyimi:** Microsoft Ofis 2003 için Visual Studio Araçları ile programcılar .Net sınıf kütüphanelerini kullanabilirler. Böylelikle bir çok zahmetli iş için çok daha az satır kod yazmak zorunda kalacağız. Mesala stringleri işlemede, veri yapılarında, veri tabanı işlemlerinde ve dosya yönetiminde büyük kolaylıklar sağlar. Dahası Visual Studio.NET ile daha güçlü ofis uygulamaları geliştirme şansına da sahibiz. Microsoft Ofis 2003 için Visual Studio Araçları ile Word ve Excel dosyalarının nesne modellerine tam olarak erişim ve onları programlama hakkımız doğuyor.
- **Kolaylaştırılmış program kurulumu ve bakımı:** Microsoft Ofis 2003 için Visual Studio Araçları ile yazdığımız kodlar DLL olarak derlenebilir. Bu DLL(ler) genelde ağ üzerinde paylaşımda olan bir yerde dururlar ve Excel veya Word açıldığında ilgili dll makineye indirilir ve çalıştırılır. Eğer kodda bir değişiklik olursa yeni derlenmiş kod otomatik olarak istemci makineye indirilir.

- **Gelişmiş güvenlik:** Microsoft Ofis 2003 için Visual Studio Araçları ile daha güvenli bir çalışma ortamına sahip olacağız. Hem güvenlik kod (trusted code) çalıştıracamız hem de güvenliğin sistem yöneticisi tarafından denetim altına alınması sağlanacak.

“Whidbey” kod isimli Visual Studio 2004

“ Gelişmiş araçlar, tüm kritik zamanlarda, uygulamalar için çok önemli dönemeçler olmuştur Aynı şekilde uygulamadaki bu kritik dönemeçler bilgi işlem alanında bir sonraki aşamayı getirmiştir.”

-Bill Gates

2004 yılında piyasaya sunulacak olan Visual Studio.NET ve .NET altyapısı yazılım geliştirmenin tüm alanlarında çok önemli değişiklikleri beraberinde getirecektir. Geliştiricilerden alınan geribildirimler (feedback) ve bunların dikkatlice değerlendirilmesiyle programcıların daha verimli olmalarını ve IDE içinden diğer yazılım geliştiricilere ulaşmayı ve destek hizmetlerine ulaşmayı mümkün kılacaktır. Yenilikler programlama dillerindeki gelişmeler, .NET Platformundaki değişiklikler ve kurumsal yazılım geliştirme projelerine destek ve yardımların artırılmasıdır.

Diğer göze çarpan gelişme ise Microsoft tarafından üretilen yazılım geliştirme araçlarının planlı olarak birbiri ile ve sistemle daha uyumlu hale gelmesidir. Whidbey’in SQL Server Yukon ile çok iyi entegrasyonu bu uyumluluk planlarının başında geliyor. Tıpkı Windows Server 2003’ün daha sisteminize kurulurken .NET Platformunun varsayılan olarak kurulması gibi. Bu sayede SQL Server Yukon CLR ortamına tam olarak adapte olmuş hale gelecektir. Yukarıda da belirtildiği gibi Whidbey ortamında SQL Server Yukon üzerinde çalışan saklı yordamlar (stored procedures) yazabileceğiz. Tabi ki Whidbey ile veri tabanı işlemlerimizi daha az kod yazarak gerçekleştirme şansımız vardır.

Yukarıdaki geniş değişikliklerin yanında yenilikler başlıca şu konularda olmuştur:

- **Programlama Dilleri:** Bu versiyonda Microsoft Visual Studio içerisinde tam destek verdiği 4 dilde (Visual Basic, Visual C#, Visual C++ ve Visual J#) önemli değişiklikler yapacak. Bu değişiklikler dillerin güçlerini artıracakları gibi dillerin özellikleri ve ortak çalışabilmesine en ufak bir yan etkisi olmayacaktır.
- **.NET Platformu:** Whidbey ile .NET Platformundaki sınıf kütüphanelerinde önemli değişiklikler olacak. Değişiklikler daha güçlü ve hoş Windows uygulamaları geliştirmeyi sağlayacağı gibi ASP.NET programlama ve ADO.NET veri işlemleri daha verimli olacaktır. Ayrıca en son web servisleri standartlarını destekleyecek ve daha geniş çaplı cihaz tabanlı (Mobil veya diğer programlanabilir cihazlar için) programlama imkanları gelecek.
- **Kurumsal Yazılım Geliştirme:** Bu yeni versiyon ile sistem tasarımcılarına ve kurumsal yazılım geliştiren yazılım mühendislerine kapsamlı ve etkili çözümler için yeni araçlar sunulacak. Bu araçlar geliştirilmiş proje analizi ve tasarımı, yazılım ayarları yönetme ve yazılımın dağıtılması (deployment) gibi kritik noktalar için düşünülmüştür.

Programlama Dilleri

.NET Platformunda yazılım geliştirmek için 20’den fazla değişik dil kullanabiliriz. Bunun yanında Microsoft resmi olarak .Net platformunda 4 dili Whidbey’de destekliyor olacak. Microsoft Whidbey’de bu 4 dil için gerekli tüm araçları ve desteği en güvenilir yazılım geliştirmek için bizlere sunuyor.

Visual Basic

Whidbey ile gelecek olan Visual Basic versiyonunda programcılarının verimliliğini inanılmaz seviyede artıracak yenilikleri göreceğiz. Tabi bu yenilikler Visual Basic programlama dili ile .NET ortamında yazılım geliştirmek için bize sunulan tüm özellikleri de sonuna kadar kullanacağız. Visual Basic Whidbey'deki kritik değişiklikler temel olarak şunlardır:

- 1 Sık sık yazmak zorunda kaldığımız bazı kodları yazmak çok daha hızlı olacaktır.**
- 2 Program tasarım halindeyken dahi hataları minimize etmek için alınan önlemler ve yollar.**
- 3 Veri ve Veritabanlarına daha kolay erişim.**
- 4 Geliştirilmiş RAD hata ayıklama**
- 5 Çok ileri seviyede Visual Basic programları yazabilme.**

1. Çoğu programda sık sık yazmak zorunda kaldığımız kodların yazımı Visual Basic Whidbey'de en az iki katı hızlı bir biçimde yazılabilir. Programcı verimliliğinin artması için çalışma zamanı nesnelere ve metodlarına direk olarak erişim ve bunları getirdiği esneklik diğer bir güzel haber. Kod editöründeki gelişmeler sayesinde sık sık yazılan kodları hızlıca yazmak için sadece belirli boşlukları doldurmak yetecektir. Bu sayede dilin söz dizimi yerine geliştirilen projenin mantığı üzerinde yoğunlaşma fırsatı bulacağız.

2. Yeni kod editörü sayesinde her seviyedeki programcılarının hatalarını en aza, daha tasarım aşamasında, indirmek mümkün. Microsoft Word'ta bulunan gramer ve yazım hatalarını kontrol ve düzeltmeye yarayan aracın bir benzeri Visual Basic Whidbey ile gelecek. Visual Basic derleyicisi de daha iyi bir kod denetimi yaptıktan sonra programı derleyecek böylece çalışma anında ortaya çıkması muhtemel hataların önüne geçilecek.

3. Visual Basic Whidbey ile veriye erişim ve veri üzerinde değişiklikler yapmak çok daha kolay hale geliyor. Kolaylaştırılan işlerin başında, yerel ve uzaktaki veriye, işle ilgili veri taşıyan nesnelere ve uzaktaki XML Web servislerine erişim geliyor. Whidbey ayrıca sadece veriler üzerinde çalışan (databound) programlar geliştirmeyi de inanılmaz kolay hale getiriyor. Bu tür programları tek satır dahi kod yazmadan dahi geliştirme imkanı bulacağız. Çok sık kullanılan veriye erişim senaryoları için tasarlanan bu yöntemlerle programları veri kaynağındaki tabloları ve sütunları sürükleyip bırakarak programı geliştirebileceğiz.

4. Whidbey ile gelen hata ayıklama yöntemleri için araçlar hem daha güçlü hem de Visual Basic programcılarının aşına oldukları bir biçimde tasarlandı. Edit ve Continue komutlarının baştan tasarımı sayesinde programda hata ayıklarken tekrar tekrar programı derlemeyi ve hata ayıklamaya devam etmeyi unutun. Ayrıca break modundaki değişiklikler ile daha önce görülmemiş en güçlü ve esneklikte hata ayıklama araçlarına sahip olacağız.

5. Son olarak, ileri seviyedeki Visual Basic programları için dilde bir çok iyileştirmeler yapıldı. Bunlar işlemlere aşırı yüklenme (operator overloading), işaretli veri tipleri (unsigned data types), kod içinde XML tabanlı kod dokümantasyonu yazımı (inline XML-based code documentation) ve kısmi veri tipleri (partial types). Dildeki bu gelişmeler sayesinde Visual Basic programcılarının tip güvenli (type -safe), yüksek performanslı, derleme zamanında onaylanmış (compile time-verified) olan **generics** yazabilecekler. Bu sayede kodun tekrar tekrar farklı veri tipleriyle birlikte kullanılmasını beraberinde getirecektir.

Önceki versiyonları gibi Visual Basic Whidbey'de hızlı bir biçimde program geliştirmeyi mümkün kılmak üzerine yoğunlaşmıştır. Planlı olan yenilikler ile Visual Basic Programcılarının

daha güvenli, daha sağlam ve daha hoş programları kolay bir biçimde geliştirip onları aynı kolaylıkla web, çalışma grubu ve kurumsal ortamlarda dağıtmayı/kurmayı garantiliyor.

Visual C++

Visual C++ Whidbey önceki versiyonlarından daha güçlü olarak sistem programlama ve yazılım geliştirme görevlerini hem Windows hem de .NET'i tercih eden programcılar hedef alıyor. Planlı olarak yapılan yenilikler derleyiciyi, geliştirme ortamını, programlama dilini ve temel kütüphaneleri kapsıyor. Ek olarak Visual C++ Whidbey ile mobil cihazlar için native C++ uygulamalarında geliştirmek mümkün olacak.

C++ derleyicisindeki gelişmelerden biri Profile Guided Optimization (PGO)'dır. PGO teknolojisi derleyicinin bir uygulamayı inceleyip onun nasıl kullanıldığı hakkında bilgi toplamasıdır. Bu bilgiler ile Visual C++ kodu daha iyi biçimde optimize edecek. Son hali olmasada 64-Bit PGO teknolojisinin Pre-release versiyonu ücretsiz olarak indirilebilir. Whidbey de ise bu teknoloji daha gelişmiş olarak 32-bit derleyici için hazır olarak gelecektir.

CLR'nin ön sürümlerinde Visual C++ Managed Extensions ile gelecek ve programcılar .NET'in tüm tüm olanaklarına ulaşabilecekler. Whidbey sürümünde ise Visual C++ geliştiricileri C++'a has özelliklere, mesela **generics'e** sahip olacak. Diğer gelişmeler ile C++'ı CLR ortamında yazılım geliştirme aracı olarak kullanmak daha kolay bir hal alacaktır.

Visual C++ Whidbey C++ temel kütüphanelerinde bir çok gelişmeyi beraberinde getiriyor. Bildiğimiz gibi C++'ta kullanabileceğimiz dünya çapında yaygın kütüphaneler bulunuyor. Bunlar en çok öne çıkanlarından biri de Microsoft Foundation Class (MFC)'dir. Visual C++ Whidbey ile gelen MFC'de bir yönden yeni gelişmeler olacak. Bunların en dikkat çekenini ise Windows Fusion teknolojisine destektir. Windows Fusion DLL'lerin çıkardığı sorunları aza indirmek için yaratılan ileri seviye bir teknolojidir. Diğer önemli gelişme ise kolayca MFC tabanlı uygulamaların .NET platformu tarafından desteklenmesidir.

Visual C#

Microsoft Visual C#'a değişik dillerden çok hoş özellikleri Whidbey'de eklemeyi planlıyor. Bu değişiklikler ile programcılara "Kod odaklı RAD" olanakları sağlanacak. Yani, C# programcılarını daha verimli bir biçimde tekrar kullanılabilir nesne yönelimli bileşenler ve iş taslakları geliştirecekler. Ekleniecek yenilikler generics, itaretörler, anonymous metodlar ve kısmi tiplerdir.

Bir yazılım projesinin karmaşıklığı artıkça programcılar daha fazla oranda hazır olan program bileşenlerini direk kullanmaya veya onların özerinde az bir değişiklikle kullanma eğilimi gösterirler. Böyle yüksek seviyede kodun yeniden kullanılmasını başarmak için **generics** ismi verilen yöntemi tercih ederler. Whidbey'de CLR içine yerleştirilen özellikler sayesinde yüksek performanslı, tip güvenli ve derleme zamanında onaylanmış generics'leri C++'ta bulunan template'lere benzer biçimde geliştirebiliriz. Generic ler programcılara kodu bir kere yazıp bir çok değişik veri tipleriyle birlikte hiç bir performans kaybı olmadan kullanmayı vaad eder. CLR de yazılan genericlerin benzerlerine göre daha sade koda, bu sayede daha kolay okunabilir ve bakımı yapılabilir olmaları büyük bir avantajdır.

C# ile kodun tekrar kullanılması yönünde bir çok kolaylıkların gelmesine ek olarak tekrar tekrar yapmamız gereken bazı karmaşık kod parçaları için de yeni yeni çözümler

üretiştir. Mesela enum sabitleri için yenileciler(iterators). Yenileyiciler sayesinde enum sabitleri ile çalışmak daha rahat bir hal almıştır. Bilgisayar bilimlerinde araştırmalarda kullanılan CLU, Sather ve Icon programlama dillerindeki özelliklerden esinlenerek foreach blokları içinde hangi veri tiplerinin nasıl ne şekilde yenileyicilerin kullanılmasının tanımlanması mümkün hale gelmiştir.

Anonim metodlar (anonymous methods) da C# diline Whidbey ile girecek. Bu tür metodlar ile yazmış olduğumuz bir kod parçasını bir delege içine koyup daha sonra kullanacağız. Anonim metodlar programlama dillerinin incelendiği derslerde geçen **lamda function** fikri üzerine kurulmuştur ve Lisp ve Phyton dillerinde uygulanmıştır. Bu tür metodlar kullanılacakları anda ve yerde tanımlanırlar. Normalde bir fonksiyon daha önce tanımlanır ve derleyici onun imzasını (method signature) bilmek ister. Böylelikle anonim metodlar, özellikle metodun yaptığı iş veya metodun imzasının çalışma anında değişmesinin gerektiğinde bazı işlemlerin yapılmasını daha uygun ve kolay hale gelir.

Son olarak Whidbey C# ile programcılar bir veri tipinin tamamını tek bir yerde değil değişik kaynak dosyalarında tanımlayabilecekler. Bu tür tipler parçalı tip (partial types) olarak adlandırılacaklar. Ayrıca parçalı tipler geniş projelerde daha kolay program tasarımı ve kod yazımı imkanı sağlayacaktır.

C# dilindeki öngörülen yenilikler ile hem büyük projeler için geliştirilecek platformların tasarımcıları hem de yazılım mimarları (software architects) için favori dil olmaya devam edecektir. Ayrıca modern söz dizimi ve bileşen yönelimli özellikleri (component-oriented) ile koda odaklanmış RAD aracı olarak karşımıza çıkacaktır.

Visual J#

J# Whidbey ile planlanmış bir çok yenilik gelecektir. Bunların amacı programcıların sahip oldukları Java deneyimlerini daha iyi bir biçimde .NET ortamında kullanmaları yönündedir. Yeniliklerin başında Borwser Controls ve J# dilinin geliştirilmesini sayabiliriz.

J#'ın 2002'de .NET'e katılması ile Java programcıları önceden yazdıkları Java Appletlerini .NET koduna çevirebilmek ve .NET ortamında da Applet türü yazılımlar geliştirebilmeyi talep ettiler. Programcıların bu isteklerine cevap olarak Microsoft J# Browser Controls adlandırılan teknolojiyi geliştirdi. Şu anda beta aşamasında olan bu teknoloji sayesinde var olan applet kaynak kodlarını açıp tekrar J# ile (çok çok az kod değişikliği ile) derlemek yeterli olacaktır. Bu teknolojinin tam olarak kullanılmaya başlandığı günlerde programcılar kendi J# Browser Control'larını tıpkı Java appletini bir web sayfasına gömer gibi gömebilecekler. Ek olarak, tabiki, J# Browser Control'ları .NET Framework'unun tüm olanaklarına erişim hakları olacak ve XML web servislerinin kullanımı mümkün olacak.

J#'a eklenecek yenilikler ile .NET dilleri arası uyumluluğu artacak ve Windows işletim sisteminin özelliklerine erişim daha rahat olacaktır. İlk olarak yeni J#'ta Enum sabitleri ve değer tipleri kavramları ile J# CLS'ye daha uyumlu olacak. İkincisi ise **volatile** ve **assert** anahtar kelimelerinin eklenmesi ile daha esnek ve daha optimize olarak çalışan program kodlarına sahip olacağız. Son olarak generic'lerin J# içinden çağrılabilmesi ile diğer .NET dilleri ile daha da uyumlu olacaktır.

Java programcıları için hem alışık bir söz dizimi hem de nesne yönelimli özellikleri ile .NET ortamında kolayca yazılım geliştirebilecekleri dil olarak J# öne çıkacaktır. Whidbey J#'ta gün yüzüne çıkacak harika özellikler sadece Java ve J++ programcılarını değil bilgisayar bilimlerinde eğitim gören öğrenciler ve onların hocalarını çok mutlu edecektir.

C# ile Rastgele Kod Üretimi

Bu uygulama birçok, yerde işimize yarayabilecek bir "Rastgele Kod Üretici" dir. Rastgele üretilmiş bir koda birçok yerde ihtiyaç duyabiliriz. Örneğin; web sitenizin üye kayıtlarında üye adaylarının gerçek email adreslerini girmelerini garantilemek isteyebilirsiniz. Bunu sağlamanın en basit yolu, kişinin verdiği email adresine rastgele ürettiğiniz bir kodu göndermektir. Böylece üye adayından, üyelik işlemlerinin tamamlanarak hesabın aktive olabilmesi için, email adresine gönderdiğiniz aktivasyon kodunu "üyelik aktivasyon" sayfanızda girmesini isteyebilirsiniz. Eğer email adresi doğru değilse aktivasyon kodunu edinemeyeceğinde üyeliği de geçerli olmaz.

Rastgele kod üretebilmek için kullanacağımız en önemli sınıf "System" isim alanı (namespace) içerisinde bulunan "Random" sınıfıdır (class). Bu sınıfı kullanarak kod içerisinde görünmesini istediğimiz karakterler dizisinin boyutu kadar rastgele tamsayı üreteceğiz.

Kullanacağımız diğer bir sınıf ise System.Text isim alanı içerisinde bulunan StringBuilder sınıfıdır. Yapacağımız işlem bir metin birleştirme döngüsü içermekte ve metin birleştirme işlemlerinde StringBuilder sınıfı, string tipine oranla daha fazla performans sağlamaktadır.

Uygulamayı bir fonksiyon olarak hazırlayacağız.

Fonksiyon üreteceği "rastgele kod" un kaç karakter uzunlukta olması istendiğini "codeLength" parametresiyle alacak. Ürettiği "codeLength" kadar karakter uzunluğundaki "Rastgele Kod"u da string veri tipinde, fonksiyonun çağırıldığı yere döndürecek.

```
private string GenerateCode(int codeLength)
{
}
}
```

Fonksiyonda ilk olarak "sb" değişken adıyla, "rastgele kod"u yapılandıracağımız StringBuilder nesnesini ve ikinci olarak da "objRandom" adıyla, rastgele sayı üretecek olan Random nesnesini yapılandıracağız.

```
System.Text.StringBuilder sb=new System.Text.StringBuilder();
```

```
System.Random objRandom=new System.Random();
```

Sıra "Rastgele Kod"umuz içinde yer almasını istediğimiz karakterleri bir metin dizisi olarak tanılamaya geldi. Ben bu örnekte "A-Z", "a-z" ve "0-9" arası karakterleri kullandım. Siz isterseniz uygulamayı zenginleştirmek için farklı karakterler de kullanabilirsiniz.

```
string[] strChars = { "A", "B", "C", ...
"1", "2", "3", ...
"a", "b", "c", ... };
```

Şimdi işlemlere başlayabiliriz. Önce rastgele üreteceğimiz sayı aralığını bulalım. Yukarıdaki karakterler "strChars" adında bir metin dizisinde tutulmaktalar. Diziler 0 indeksle başladıklarından rastgele üretilecek olan minimum rakam 0 olmalıdır. Üretilecek maksimum rakam ise dizinin en son elemanının indeksi olmalıdır. Dizinin en büyük indeksli elemanının indeks bilgisini

```
int maxRand=strChars.GetUpperBound(0);
```

koduyla aynı anda hem bu değeri tutacak olan "maxRand" adında bir değişken tanımlayarak dizinin "GetUpperBound(0)" metoduyla alırız.

"Rastgele Kod"un üretilmesi, istenilen kod uzunluğu için her bir basamağın rastgele oluşturulmasıyla sağlanır. Bunun için, 0 ile "Rastgele Kod" için kullanılacak karakter dizisinin en büyük indeksi arasında rastgele bir sayı objRandom.Next(maxRand) metoduyla üretilir ve bu değer "rndNumber" değişkenine atanır.

```
int rndNumber=objRandom.Next(maxRand);
```

Karakter dizisindeki rastgele bir eleman, edinilen "rndNumber" sayısını indeks olarak kullanarak "strChars[rndNumber]" ifadesiyle elde edilir ve bu karakter sb.Append metoduyla "sb" nesnesine eklenir.

```
sb.Append(strChars[rndNumber]);
```

Eğer 10 karakter uzunluğunda bir "rastgele kod" istenirse, önce birinci basamak için rastgele bir karakter üretilir, daha sonra ikinci basamak için ve bu böylece 10'a kadar devam eder. Bu üretilen karakterler "sb" değişkeni içerisinde ard arda eklenir.

```
for(int i=0;i<codeLength;i++)
{
    int rndNumber=objRandom.Next(maxRand);

    sb.Append(strChars[rndNumber]);
}
```

En son olarak da StringBuilder nesnesinin içerisinde yapılandırılmış olan ve rastgele karakterlerden meydana gelen sonuç, string veri tipine sb.ToString() metoduyla dönüştürülerek fonksiyon sonlandırılır ve değer fonksiyonun çağırıldığı yere döndürülür.

```
return sb.ToString();
```

Kodun tamamlanmış şekli aşağıdaki gibidir.

```
private string GenerateCode(int codeLength)
{
    System.Text.StringBuilder sb=new System.Text.StringBuilder();

    System.Random objRandom=new System.Random();

    string[] strChars = { "A","B","C","D","E","F","G","H","I",
                          "J","K","L","M","N","O","P","Q","R",
                          "S","T","U","V","W","X","Y","Z",
                          "1","2","3","4","5","6","7","8","9","0",
                          "a","b","c","d","e","f","g","h","i","j","k",
                          "l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"};

    int maxRand=strChars.GetUpperBound(0);

    for(int i=0;i<codeLength;i++)
    {
        int rndNumber=objRandom.Next(maxRand);

        sb.Append(strChars[rndNumber]);
    }

    return sb.ToString();
}
```

Metin(String) İşlemlerinde Performansı Artırmak

Programlama yaparken sık sık metinsel işlemler yapmak durumunda kalırız.

Bir metni başka bir metinle birleştiririz...

Büyükçe bir metnin içerisinden bir bölümünü alırız...

İki metni birbiriyle karşılaştırırız...

....

Bu liste böylece uzar gider. Metinsel işlemler denilince akla gelen ilk şey metin birleştirme işlemidir. "Bir metnin bir diğerine eklenmesi." İş metinle ilgili olduğunda bu işi gerçekleştirmek için C# ta ilk akla gelen veri tipi "`string`" dir. "`System.String`" sınıfını (class) temsil eden `string` bir referans türüdür. Ancak biraz farklı bir referans türüdür. Yapı olarak "immutable", yani değişmezdir/sabittir ve bu yüzden bir kez yapılandırıldıktan sonra içeriği değiştirilemez. İşte bu yapıdan dolayıdır ki bir `string` değışkene ancak bir kez değeri atayabilirsiniz.

Peki ya aşağıdaki kod?

```
string metin="Merhaba";  
  
metin=metin + " Dünya!"; // metin="Merhaba dünya!"
```

`String` bir referans türü olduğu için bellekteki öbek (heap) bölgesinde yaratılır ve içeriğine "Merhaba" yüklenir. Öbekteki bu nesneye kodun erişebilmesi için gerekli adres bilgisi ise belleğin yığın (stack) bölgesinde yaratılan "metin" değişkenine yazılır. Yani referans türlerde değişken nesneyi değil, nesnenin adresini tutar.

Şimdi ikinci satıra bakalım. Birinci satırda, metin değişkeninin temsil ettiği string nesnesine "birinci string nesnesi" diyelim. İkinci satırdaki kod işlendiğinde "birinci string nesnesi" üzerine yeni değer ekelenemeyeceğinden, öbekte ikinci bir string nesnesi türetiliyor ve işlem sonucu, yani "Merhaba Dünya!" bu ikinci string nesnesine yükleniyor.

Ve kilit nokta!

"metin" değişkeni ikinci satırdaki kodun sonunda artık "birinci string nesnesi"nin değil, "ikinci string nesnesi"nin adresini taşımaya başlıyor. Yani artık "birinci string nesnesi" tamamen erişilemez oluyor ve "Çöp Toplayıcı" (Garbage Collector) devreye girip belleği temizleyene kadar bellekte yer tutmaya devam ediyor.

İşte "`string`" veri tipinin bu yapısı nedeniyle metinsel işlemlerde daha iyi performans sağlamak amacıyla `System.Text` isim alanı içine bulunan "`StringBuilder`" sınıfı kullanılmaktadır.

Algoritmaya bağlı olarak değişmekle birlikte, `StringBuilder` sınıfının çok çok daha hızlı olduğu söylenebilir.

'string' veri tipi kullanılarak metin birleştirme.....

Baslangıç : 24.11.2003 20:31:06

Bitis : 24.11.2003 20:31:08

Toplam süre : 2193,1536 milisaniye

'StringBuilder' sinifi kullanılarak metin birleştirme.....

Baslangıç : 24.11.2003 20:31:08

Bitis : 24.11.2003 20:31:08

Toplam süre : 20,0288 milisaniye

Uygulamaya sonlandirmek için [Enter] tusuna basın.

Aşağıdaki örnek kod ve yukarıdaki ekran görüntüsü, metin birleştirme işlemlerinde gerçekten çok iyi performans gösteren `StringBuilder` sınıfının, `string` ile karşılaştırıldığı bir Console uygulamasıdır.

```
static void Main(string[] args)
{
    // Degisken tanimlari

    string str="";

    System.Text.StringBuilder sb=new System.Text.StringBuilder();

    DateTime start; // islemin baslangic ani

    DateTime end; // islemin bitis ani

    TimeSpan sure; // islemin toplam süresi

    // string veri tipiyle metin birlestirme islemi.....

    Console.WriteLine("'string' veri tipi kullanilarak metin birlestirme.....");

    start=DateTime.Now; // islemin baslangic zamani

    Console.WriteLine("Baslangic :\\t" + start.ToString());

    // Metin birlestirme islemi.....

    for(int i=0;i<10000;i++)

    {

        str +=i.ToString();

    }

    end=DateTime.Now; // islemin bitis zamani

    Console.WriteLine("Bitis :\\t\\t" + end.ToString());

    sure=end.Subtract(start); // gezen süre hesaplaniyor..

    Console.WriteLine("Toplam süre :\\t" + sure.TotalMilliseconds + " milisaniye");

    //-----

    Console.WriteLine();

    // StringBuilder sinifiyla metin birlestirme islemi.....

    Console.WriteLine("'StringBuilder' sinifi kullanilarak metin birlestirme.....");

    start=DateTime.Now; // islemin baslangic zamani

    Console.WriteLine("Baslangic :\\t" + start.ToString());
```

```
// Metin birleştirme işlemi.....

for(int i=0;i<10000;i++)

{

    sb.Append(i.ToString());

}

end=DateTime.Now; // islemin bitis zamanı

Console.WriteLine("Bitis :\\t\\t" + end.ToString());

sure=end.Subtract(start);// geçen süre hesaplanıyor..

Console.WriteLine("Toplam süre :\\t" + sure.TotalMilliseconds + " milisaniye");

//-----

Console.WriteLine();

Console.WriteLine("Uygulamaya sonlandırmak için [Enter] tusuna basin.");

Console.ReadLine();

}
```

"Builder" Tasarım Desenin C# ile Gerçekleştirilmesi

Bu yazıda "Creational" desenler grubunda yer alan "Builder" tasarım deseninden ve bu deseni C# ile nasıl gerçekleştirebileceğimizden bahsedeceğim.

Bu yazıyı okumadan önce "[Singleton](#)" ve "[Abstract Factory](#)" tasarım desenlerini konu alan yazılarımı okumanızı tavsiye ederim.

"Builder" deseni adından da anlaşılacağı üzere bir nesnenin oluşturulması ile ilgilidir. Bu desende kullanılan yapılar hemen hemen "Abstract Factory" deseninde kullanılan yapılar ile aynıdır. Bu iki desen arasında çok küçük farklılıklar vardır. Bu farkları ilerleyen kısımlarda açıklayacağım.

"Builder" deseni birden fazla parçadan oluşan kompleks yapıdaki bir nesnenin oluşturulmasını ve bu kompleks nesnenin oluşturulma safhalarını istemci modülünden tamamen gizlemek için kullanılır. Kompleks nesnenin yaratılması istemci modülünden tamamen yalıtıldığı için nesnenin yaratılması ile ilgili işlemler farklı versiyonlarda tamamen değiştirilebilir, bu istemci programın çalışmasını hiç bir şekilde etkilemeyecektir. Burda dikkat edilmesi gereken nokta ise şudur : bu desen kompleks nesneyi oluşturan yapıların gerçek nesneden tamamen bağımsız bir yapıda olduğu durumlarda kullanıldığı zaman esneklik getirecektir. Dolayısıyla her bir farklı parçanın kompleks nesnede kullanılması, kompleks nesnenin işlevini değiştirmeyeceği gibi

sadece görünümünü yada tipini değiştirecektir.

Builder deseninin Abstract Factory deseninden farkına gelince; "Abstract Factory" deseninde soyut fabrika olarak bilinen yapının metotları fabrikaya ait nesnelerin yaratılmasından bizzat sorumludur. Builder deseninde ise aynı mekanizma biraz daha farklı işlemektedir. Builder deseninde istemci modülü, nesnelerin ne şekilde oluşturulacağına soyut fabrika yapısına benzer bir yapı olan Builder yapısının metotları ile karar verir ve istemci oluşturulan bu nesneyi Builder sınıfından tekrar talep eder.

Yukarıdaki soyut açıklamalardan sonra daha gerçekçi bir örnekle bu deseni açıklamamın sizin ruh haliniz açısından faydalı olacağını düşünüyorum. Farklı donanım ürünlerinin biraraya getirilerek bilgisayar sistemlerinin oluşturulduğu bir teknik servisi göz önünde bulunduralım. Bir müşteri bu servise gelerek çeşitli özellikleri içinde barındıran bir bilgisayar talep eder. İsteği alan servis temsilcisi bu isteği teknik birimlere iletir ardından teknik eleman istenilen özelliklerde bir bilgisayarın oluşması için gerekli donanımları raflarından alır ve birleştirir. Sonuçta müşterinin isteği yerine gelir ve karşılıklı bir mutluluk havası içine girilir. Tabi bizi ilgilendirin müşterinin ve satıcının mutluluğu değil elbette. Bizi ilgilendiren nokta müşterinin bilgisayarı oluşturan parçaların birleştirilmesinden tamamen soyutlandığıdır. Dikkat ederseniz verilen özelliklerde bilgisayarın oluşmasında müşterinin hiç bir etkisi olmamıştır. Eğer müşteri son anda vazgeçip farklı özelliklerde bir bilgisayar istemiş olsaydı yine bir etkisi olmayacaktı. Bu durumda eski bilgisayarın ilgili parçaları değiştirilip yeni isteğe uygun parçalar takılacaktı. Burda da kompleks bir yapı olan bilgisayar sisteminin kendisini oluşturan parçalardan (donanım) tamamen bağımsız bir yapıda olduğunu görüyoruz. Herhangi bir diskin yada monitörün değişmesi bilgisayarı bilgisayar yapmaktan çıkarmayacak sadece bilgisayarın tipini ve görüntüsünü değiştirecektir. Bu durum bizim yukarıda açıkladığımız builder deseninin amacına tam olarak uymaktadır. Zira birazdan bir teknik servisteki bu işleri otomatize eden temel desenimizi C# ile nasıl gerçekleştirebileceğimizi göreceğiz.

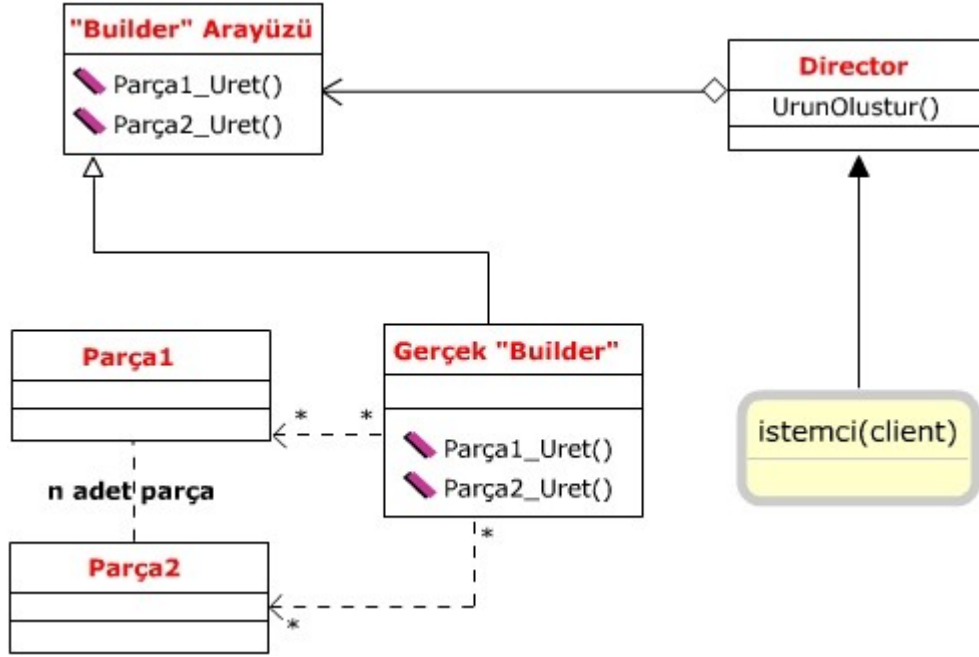
Builder desenin de temel olarak 4 yapı bulunmaktadır. Bu yapılar sırasıyla şöyledir :

- **"Builder" Arayüzü** : Bu yapı içinde gerçek/somut(concrete) Builder sınıflarının uygulaması gereken özellikleri ve metotları barındırır. "Builder" abstract bir sınıf olabileceği gibi bir arayüz de olabilir. En çok tercih edilen yapı arayüz olmasına rağmen uygulamanızın ihtiyacına göre abstract sınıfları da kullanabilirsiniz. Gerçek örneğimizde bu yapı bir bilgisayar sisteminin minimum olarak uygulaması gereken özellikleri temsil eder. Örneğin kullanıcılara satılacak her bilgisayar sistemi mutlaka HDD, RAM gibi yapıları içerisinde bulundurmalıdır. Dolayısıyla birbirinden farklı bilgisayar sistemlerinin bu tür yapıları mutlak olarak içerisinde barındırabilmesi için zorlayıcı bir etkene diğer bir deyişle abstract üye elemanlara yada arayüzlere ihtiyacımız vardır.
- **"Concrete(Somut) Builder" Yapısı** : Gerçek bir bilgisayar sistemini temsil eden ana yapıdır. Farklı farklı donanımların bir araya getirilmesi ile çok sayıda sistem oluşturulabilir. Oluşturulabilecek her sistem gerçek bir Builder yapısına denk düşmektedir.
- **Product(Ürün)** : Gerçek bir bilgisayar sistemini ve bu sistemde hangi özelliklerin bulunduğunu somut olarak yapısında barındıran bir modeldir. Bu noktada bir önceki Concrete Builder yapısının parçaları oluşturduğunu, bu parçaların birleşmesinden oluşan kompleks nesnenin Product olduğunu belirtmekte yarar var.
- **Director(Yönetici)** : Genellikle istemci ile yani örneğimizdeki müşteri ile Builder(parçaları birleştirip ürün yapan birimler) yapıları arasında köprü

görevinde bulunan yapı olarak bilinir. Kısacası bilgisayar sisteminin oluşması için raflardan donanımları alıp onları birleştiren teknik eleman Director görevindedir. Tabi Director yapısının iş yaparken Builder arayüzünün kısıtlarına göre çalışması gerektiğinin de belirtmek gerekir.

UML Modeli

Aşağıdaki şekil "Builder" tasarım deseninin UML sınıf diagramını göstermektedir. Şemadaki her bir şekil desendeki ilgili yapıyı modellemektedir. Ayrıca desendeki sınıflar arasındaki ilişkilerde detaylı bir şekilde gösterilmiştir.



Yukarıdaki UML diyagramını kısaca özetleyecek olursak : Builder arayüzünde bulunan metotlar yardımıyla gerçek builder sınıflarında kompleks nesnenin parçaları birleştirilerek istenilen nesne oluşturulur. Kompleks sistemi oluşturan parçalar farklı sayılarda olabildiği halde minimum gereklilikleri de sağlamak zorundadır. Bu minimum gereksinimler Builder arayüzünde bildirilmiştir. Director sınıfındaki bir metot, her sistem için standart olan Builder arayüzündeki metotları kullanarak hangi sistemin oluşturulduğundan bağımsız olarak kompleks nesnenin oluşturulması sağlanır. Yani Director hangi tür bir bilgisayar sisteminin oluşturulacağı ile ilgilenmez. Director sınıfı sadece kendisine gönderilen parametre sayesinde Builder arayüzünün metotlarından haberdardır. Son olarak istemci Director sınıfının ilgili metotlarını kullanarak gerçek ürünün oluşmasını sağlar. Bunun için elbette Director sınıfının ilgili metoduna hangi ürünün oluşturulacağını bildirmesi gerekir.

Bu kısa açıklamadan sonra yukarıdaki diyagramı baz alarak modern bir programlama dili olan C# ile bu deseni nasıl gerçekleştirebileceğimizi inceleyelim.

Builder Deseninin C# ile Gerçekleştirilmesi

! Aşağıdaki kodları kısaca inceleyip açıklamaları okumaya devam edin.

```
using System;

namespace BuilderPattern
{
```



```
public interface IBilgisayarToplayicisi
{
    Bilgisayar Bilgisayar{get;}

    void CDROM_Olustur();
    void HDD_Olustur();
    void Monitor_Olustur();
    void RAM_Olustur();
}

public class GoldPC : IBilgisayarToplayicisi
{
    private Bilgisayar mBilgisayar;

    public Bilgisayar Bilgisayar
    {
        get{return mBilgisayar;}
    }

    public GoldPC()
    {
        mBilgisayar = new Bilgisayar("Gold-PC");
    }

    public void CDROM_Olustur()
    {
        mBilgisayar["cdrom"] = "52X GoldStar";
    }

    public void HDD_Olustur()
    {
        mBilgisayar["hdd"] = "60 GB Seagate";
    }

    public void Monitor_Olustur()
    {
        mBilgisayar["monitor"] = "17' Hyundai";
    }

    public void RAM_Olustur()
    {
        mBilgisayar["ram"] = "512 MB DDR Kingston";
    }
}

public class SilverPC : IBilgisayarToplayicisi
{
    private Bilgisayar mBilgisayar;

    public Bilgisayar Bilgisayar
    {
        get{return mBilgisayar;}
    }
}
```

```

public SilverPC()
{
    mBilgisayar = new Bilgisayar("Silver-PC");
}

public void CDROM_Olustur()
{
    mBilgisayar["cdrom"] = "48X Creative";
}

public void HDD_Olustur()
{
    mBilgisayar["hdd"] = "30 GB Maxtor";
}

public void Monitor_Olustur()
{
    mBilgisayar["monitor"] = "15' Vestel";
}

public void RAM_Olustur()
{
    mBilgisayar["ram"] = "256 MB SD Kingston";
}
}

public class Bilgisayar
{
    private string mBilgisayarTipi;
    private System.Collections.Hashtable mParcalar = new
System.Collections.Hashtable();

    public Bilgisayar(string BilgisayarTipi)
    {
        mBilgisayarTipi = BilgisayarTipi;
    }

    public object this[string key]
    {
        get
        {
            return mParcalar[key];
        }
        set
        {
            mParcalar[key] = value;
        }
    }

    public void BilgisayariGoster()
    {
        Console.WriteLine("Bilgisayar Tipi : " + mBilgisayarTipi);
        Console.WriteLine("---> CD-ROM Model : " + mParcalar["cdrom"]);
        Console.WriteLine("---> Hard Disk Model : " + mParcalar["hdd"]);
        Console.WriteLine("---> Monitör Model : " + mParcalar["monitor"]);
    }
}

```

```

        Console.WriteLine("---> RAM Model : " + mParcalar["ram"]);
    }
}

public class TeknikServis
{
    public void BilgisayarTopla(IBilgisayarToplayicisi bilgisayarToplayicisi)
    {
        bilgisayarToplayicisi.CDROM_Olustur();
        bilgisayarToplayicisi.HDD_Olustur();
        bilgisayarToplayicisi.Monitor_Olustur();
        bilgisayarToplayicisi.RAM_Olustur();
    }
}
}

```

Kaynak koddan da görüleceği üzere en temel yapımız IBilgisayarToplayicisi arayüzüdür. Bu arayüzde bir Bilgisayar ürününü temsil etmek için bir özellik ve bilgisayarı oluşturan parçaları oluşturmak için gereken metotlar bulunmaktadır. Bu metotların bu arayüzden türeyen bütün sınıflar tarafından uygulanması gerekmektedir. Örneğimizde bir GoldPC ve SilverPC bilgisayar modelleri için gereken bileşenleri oluşturmak için 4 adet metot bulunmaktadır. Burada en önemli nokta her Bilgisayar tipinde bir özelliğinde bulunması. Bu özellik istendiği zaman oluşturulan Bilgisayar ürününü istemciye vermektedir. Zaten dikkat ederseniz her bir sisteme ilişkin özellikler Bilgisayar sınıfındaki Hashtable nesnesinde saklanmıştır. Ayrıca her Bilgisayar nesnesinin ayrıca bir tip bilgisi saklanmaktadır. Son olarak TeknikServis isimli sınıfımızı ele alalım. Bu sınıf Builder desenindeki Director yapısına denk düşmektedir. Bu sınıftaki BilgisayarTopla metodu kendisine gönderilen bilgisayarToplayicisi arayüzü referansına ait metotları kullanarak Bilgisayar nesnesinin parçalarını kendisi oluşturmaktadır. Bu örnekte bilgisayarı oluşturan her parça kafa karıştırmamak için ayrı bir sınıf olarak tasarlanmamıştır. Bunun yerine string türünden değerlere sahip olan bir hashtable nesnesi kullanılmıştır.

Not : Yukarıdaki örnek gerçek otomasyonda birebir kullanılamayabilir. Örneğin her bir bilgisayar modeli için ayrı ayrı sınıf tasarlamak hoş olmayabilir. Bu sorunu daha dinamik bir şekilde çözmek gerekir. Buradaki örnek sadece Builder desenin daha rahat bir şekilde kavrayabilmeniz için verilmiştir.

Son olarak yukarıdaki yapıları kullanan bir istemci programı yazıp desenimizi test edelim.

```

using System;

namespace BuilderPattern
{
    class Class1
    {
        static void Main(string[] args)
        {
            TeknikServis teknikservis = new TeknikServis();

            IBilgisayarToplayicisi BT1 = new GoldPC();
            IBilgisayarToplayicisi BT2 = new SilverPC();

            tekniksevis.BilgisayarTopla(BT1);
        }
    }
}

```

```
teknikservis.BilgisayarTopla(BT2);

BT1.Bilgisayar.BilgisayariGoster();
Console.WriteLine("-----");
BT2.Bilgisayar.BilgisayariGoster();
    }
}
}
```

Programı çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.

```
Bilgisayar Tipi : Gold-PC
---> CD-ROM Model : 52X GoldStar
---> Hard Disk Model : 60 GB Seagate
---> Monitör Model : 17' Hyundai
---> RAM Model : 512 MB DDR Kingston
-----
Bilgisayar Tipi : Silver-PC
---> CD-ROM Model : 48X Creative
---> Hard Disk Model : 30 GB Maxtor
---> Monitör Model : 15' Vestel
---> RAM Model : 256 MB SD Kingston
```

Desenlerle ilgili bir sonraki yazımda Creational desenlerden olan "Prototype" desenini ele alacağım. Herkese iyi çalışmalar.

Kod Optimizasyonu ve "volatile" Anahtar Sözcüğü

Bu yazıda C#'ın önemli ama tam olarak neden kullanıldığı bazı profesyonel programcılar tarafından bile pek fazla bilinmeyen bir anahtar sözcük olan volatile üzerinde duracağız.

Bir çok popüler derleyici sizin isteğiniz dışında kodunuzun işleyiş mantığına müdahale edebilmektedir. Bu müdahalenin en bilinen sebeplerinden birisi uygulamanızın kod boyutunu küçültmek yada uygulamanızın çalışma zamanını düşürmektir. Aslında biz bu işlemlerin tamamına birden optimizasyon da diyebiliriz. Zira hemen hemen bütün derleyicilerin optimizasyon işleminin yapılıp yapılmayacağını belirten bir parametresi vardır. C# derleyicisi için bu parametre /optimize yada kısaca /o şeklindedir.

Peki optimizastondan neyi anlamalıyız? Genel olarak iki farklı şekilde optimizasyondan bahsetmek mümkündür. Birincisi daha henüz derleme aşamasındayken programcının gözünden kaçan bazı gereksiz bildirimlerin veya tanımlamaların derleyici tarafından derlenecek koda dahil edilmemesi ile olabilir. Örneğin hiç kullanmadığınız bir değişken için bellekte bir alan tahsis edilmesinin hiç bir gereği yoktur. Bu yüzden hiç bir yerde

kullanılmayan değişkenlerin derleyiciniz tarafından derleme modülüne iletilmemesi bir optimizasyon olarak görülmektedir. Bu tip optimizasyon kapsamı içinde ele alınabilecek diğer bir örnek ise aşağıdaki kod parçası ile gösterilmiştir.

Not : Aşağıdaki kodun bilinçsiz yada dalgın bir programcı tarafından yazıldığını varsayıyoruz.

```
int a = 0;

while(a != 0)
{
    a = 2 ;
    a = 0 ;
}
```

Yukarıdaki kodda akış hiç bir zaman while bloğunun içine gelmeyecektir. Ve üstelik eğer a değişkeni farklı bir iş parçacığı(thread) tarafından while bloğuna girmeden değiştirilip akış while bloğuna girse bile a değişkeni while bloğu içinde tekrar eski sabit değerine atanıyor. Dolayısıyla while bloğunda bulunan kodların çalıştırılabilir uygulama dosyasının boyutunu büyütmekten başka bir işe yaramayacağı açıktır. O halde burda insan üstü bir mekanizmanın devreye girip kodu optimize etmesi gerekir. Bu mekanizma elbetteki derleyicinin optimizasyon işine yarayan parametresidir. Optimizasyon işleminde derleyiciden derleyiciye fark olmasına rağmen yukarıdaki kod parçasının geebileceği en optimum biçim aşağıda gösterildiği gibidir.

```
int a = 0;

while(a != 0)
{
}

}
```

Diğer bir optimizasyon biçimi ise derleyicinin değişkenlerin elde edilmesinde yada tekrar yazılmasında belleğin yada işlemcinin tercih edilmesi ile ilgilidir. Bu noktada mikro işlemciler ile ilgili kısa bir bilgi vermekte fayda var : Kaynak kodumuz çalıştırılabilir durumdayken aslında makine koduna çevrilmiştir. Bu komutlar daha önceden mikro işlemcilerde elektronik düzeyde programlandıkları için bu komutlar tek tek mikro işlemcide kolaylıkla icra edilir. Mikro işlemciler aynı anda tek bir iş yapabileceği için yapacağı işlemler içinde kullandığı değişkenleri her defasında bellekten almak yerine daha hızlı olması açısından mikro işlemcideki data register dediğimiz kayıtçılarda tutar. Bu register dediğimiz bölümlerin sınırlı sayıda bulunduğunu belirtmek gerekir. Dolayısıyla bellekte bulunan uygulamamızın ihtiyacına göre daha doğrusu icra edilecek bir sonraki makine kodunun çeşidine göre işlemci bellekten ilgili değişkenleri register'larına yükler ve ilgili komutunu çalıştırır. Doğaldır ki bir değerın mikro işlemci tarafından belleğe(ram) yazılması ile mikro işlemcideki register bölgesine yazılması arasında dağlar kadar fark vardır. Bu fark elbette hız faktörüdür. İşte tam bu noktada ikinci tip optimizasyon kuralını tanımlayabiliriz. Derleyici öyle bloklara rastlayabilir ki, bu bloklar içinde bulunan bir değişkenin değerini her defasında bellekten okuyacağına bu değişkenin değerini bir defaya mahsus olmak üzere mikro işlemcinin ilgili register bölgesine bölgesine kaydeder ve sonraki okumalarda işlemci bellek yerine bu register bölgesini kullanır. Böylece kodunuzun çalışma süresinde önemli sayılabilecek bir azalma görülür. Elbetteki bu optimizasyon işleminin yüzlerce kez tekrarlandığını varsayarak bu sonuca varıyoruz.

Yukarıda değişken okuma ile ilgili söylediklerimin aynısı bir değişkenin değerini değiştirmek için de geçerli olduğunu belirtmeliyim. Yani siz programınızın 10. satırında bir

değişkenin değerini bir değerden başka bir değer çektiğiniz halde derleyici bu işlemi 15.satırda yapabilir. Bu durumda 15. satıra kadar o değişkenin kullanılmadığı yorumunu yapabiliriz. Bu şekilde mikroişlemcinin belleğe yazma işlemi geciktirilerek belli ölçüde optimizasyon sağlanır. Tabi bu optimizasyonun ölçüsü tamamen mikroişlemcinin o anki durumuna bağlıdır.

Buraya kadar herşey normal. Bir de madolyonun öteki yüzüne bakalım. Bildiğiniz üzere uygulamalar genellikle çoklu iş parçacıklarından(multi thread) ve proseslerden oluşur. Her bir proses diğer bir proses teki değişkene işletim sisteminin izin verdiği ölçüde erişip üzerinde işlemler yapabilir. Aynı şekilde bir iş parçacığıda diğer bir iş parçacığında bulunan değişkene erişip üzerinde çeşitli işlemler yapabilir. Peki bunun bizim optimizasyon kurllarımızla bağlantısı ne? Şöyle ki : derleyici bir değişkenin değerinin farklı bir iş parçacağı tarafından yada farklı bir proses tarafından işleneceği üzerinde durmaz. Bu tamamen işletim sisteminin yönetimindedir. Hal böyleyken bizim yukarıda bahsettiğimiz ikinci optimizasyon tipi bazı durumlarda yarar getireceğiniz zarar getirebilir. Zira optimizasyon adına bir değişkenin değerini her defasında bellekten okuma yerine mikroişlemcideki ilgili register dan okurken o anda farklı bir iş parçacağı yada farklı bir proses hatta ve hatta işletim sistemi sizin erişmeye çalıştığınız değişkenin değerini sizin uygulamanızın mantığına göre değiştirebilir. Bu durumda siz o değişkenin son halini kullanmamış olursunuz. **Dolayısıyla programınızda farklı thread lar yada prosesler arasında paylaşılan veya işletim sistemi tarafından değiştirilmesi muhtemel olan değişkenlerinizi optimizasyon kuralına tabi tutmamanız gerekir.** Peki bunu nasıl başaracağız?

volatile anahtar sözcüğü burada imdadımıza yetişiyor. Bir değişkeni volatile anahtar sözcüğü ile bildirdiğiniz takdirde derleyicinizin optimizasyon ile ilgili parametresini açık tutsanız bile ilgili değişken yukarıda bahsi geçen tehlikeli optimizasyon kurallarına tabi tutulmayacaktır. Yani volatile ile bildirilen değişkenlere programın akışı sırasında her ihtiyaç duyulduğunda değişkenin gerçek yeri olan belleğe başvurulur. Aynı şekilde bir değişkene yeni bir değer yazılacağı zaman bu yazma işlemi hiç geciktirilmeden bellekteki yerine yazılır. Böylece volatile ile bildirilen değişkenler farklı iş parçacıkları yada prosesler tarafından ortak kullanılıyor olsada programın akışı içerisinde her zaman son versiyonu elde edilecektir. Çünkü bu değişkenlerin değeri her defasında bellekten çekilir. Her ne kadar optimizasyondan taviz verme zorunda kalsak ta böylece uygulamalarımızda çıkabilecek olası bugların(böcek) önüne geçmiş oluruz.

volatile, C# dilindeki anahtar sözcüklerden biridir. Üye değişken bildirimi ile birlikte kullanılır. volatile anahtar sözcüğü yalnızca aşağıdaki değişken tipleri ile birlikte kullanılabilir.

- Herhangi bir referans tipindeki değişken ile
- byte, sbyte, short, ushort, int, uint, char, float yada bool. türünden olan değişkenler ile
- byte, sbyte, short, ushort, int, yada uint türünden semboller içeren numaralandırmalar(enums) ile
- unsafe modda iken herhangi bir gösterici türü ile

volatile anahtar sözcüğünün kullanımına bazı örnekler verelim :

```
public static volatile int a;
```

```
public volatile bool a;
```

```
public volatile int* a;
....
```

Microsoft'un resmi dökümanlarında(MSDN) verilen bir örneği buraya taşıyarak ne gibi durumlarda volatile anahtar sözcüğüne ihtiyaç duyabileceğimizi görelim.

```
using System;
using System.Threading;

class Test
{
    public static int result;
    public static volatile bool finished;

    static void Thread2()
    {
        result = 143;
        finished = true;
    }

    static void Main()
    {
        finished = false;
        new Thread(new ThreadStart(Thread2)).Start();

        for (;;)
        {
            if (finished)
            {
                Console.WriteLine("result = {0}", result);
                return;
            }
        }
    }
}
```

Yukarıdaki örnek programdaki püf nokta finished isimli değişkenin ana thread ve ana thread içinde başlatılan yeni thread tarafından ortak kullanılan bir değişken olmasıdır. Eğer finished değişkeni volatile olarak bildirilmemiş olsaydı, akış thread2 metoduna gelmiş olmasına rağmen Main metodu içindeki if bloğu çalıştırılmayabilirdi. Çünkü derleyici ana thread içinden finished değişkenininine tıpolanmış bir bölgeden(register) erişebilir. Bu durumda finished değişkeninin gerçek değeri true olmasına rağmen ana thread de finished değişkeni halen false olarak ele alınır. Bu yüzden finished değişkeninin her durumda son versiyonunu elde etmek için bu değişken volatile anahtar sözcüğü ile bildirilmiştir.

volatile anahtar sözcüğünün kullanımına bir örnek daha verelim. Belli bir anda bir sınıftan sadece bir nesnenin oluşmasını sağlayan **Singleton** desenini daha önceki bir makalemde ele almıştım. Bu konu ile ilgili bilgi eksikliğiniz varsa ilgili makaleyi okumanızı tavsiye ederim. Bahsi geçen makalede verilen desenin bir uygulaması aşağıda ki gibi yeniden yazılmıştır.

```
public class SingletonDeseni
{
    private static volatile SingletonDeseni nesne;
```

```
private static Object kanalKontrol = new Object;

private SingletonDeseni()
{
}

public static Singleton Nesne()
{
    if(nesne == null)
    {
        lock(kanalKontrol)
        {
            if(nesne == null)
            {
                nesne = new SingletonDeseni();
            }
        }
    }

    return nesne;
}
}
```

Bu örnekte SingletonDeseni nesnesinin belli bir anda tekil olarak bulunduğunu çok kanallı uygulamalar içinde geçerli kılmak için bu nesne volatile olarak bildirilmiştir. Üstelik bu örnekte farklı bir prosesin müdahalesi olsa bile bu nesneden ancak ve ancak bir adet yaratılacaktır.

Son olarak volatile kelimesinin sözlük anlamı üzerinde durmak istiyorum. İki yıl önce İngilizce'den Türkçe'ye çevrilmiş bir Visual C++ kitabını okuduğumda volatile ile bildirilmiş değişkenlerden oynak(!) değişkenler diye bahsedildiğine şahit oldum. İlk başta bu ilginç kullanım bana birşey ifade etmedi ama hislerimin yardımıyla aslında yazarın volatile dan bahsettiğine karar verdim. Sizde takdir edersiniz ki yukarıda anlattıklarımız ile "oynak" kelimesi arasında pek bir bağ bulunmamaktadır. Kitabın çevirisini yapan yazar muhtemelen bir programcı değil bir çevirmendi. Çünkü eğer iyi bir programcı olsaydı oynak kelimesi yerine daha uygun bir kelime seçilebilirdi. volatile'ın sözlük anlamı "uçucu olan", "buhar olan" anlamına gelmektedir. Ancak ben henüz volatile sözcüğüne kendi mesleğimizle ilgili uygun bir karşılık bulamadım. Bu konuda daha önce **cdili** ve **cderne** isimli iki yahoo grubunda çeşitli tartışmalar olmuştur. Bu gruplara üye olarak ilgili tartışmalarda geçen konuşmaları okumanızı öneririm. Eğer sizin de bu konuda önerileriniz varsa bizimle paylaşırsanız seviniriz.

Umarım faydalı bir yazı olmuştur. Herkese iyi çalışmalar...

Overloaded(Aşırı Yüklenmiş) Metotların Gücü

Bu makalemde sizlere overload kavramından bahsetmek istiyorum. Konunun daha iyi anlaşılabilmesi açısından, ilerleyen kısımlarda basit bir örnek üzerinde de çalışacağız.

Öncelikle Overload ne demek bundan bahsedelim. Overload kelime anlamı olarak Aşırı Yükleme anlamına gelmektedir. C# programlama dilinde overload dendiğinde, aynı isme sahip birden fazla metod akla gelir. Bu metodlar aynı isimde olmalarına rağmen, farklı imzalara sahiptirler. Bu metodların imzalarını belirleyen unsurlar, parametre sayıları ve parametre tipleridir. Overload edilmiş metodları kullandığımız sınıflarda, bu sınıflara ait nesne örnekleri için aynı isme sahip fakat farklı görevleri yerine getirebilen (veya aynı görevi farklı sayı veya tipte parametre ile yerine getirebilen) fonksiyonellikler kazanmış oluruz.

Örneğin;

```
public string MetodA(int a)
```

```
public int MetodA(int a,int c)
```

```
public void MetodA(string d)
```

Şekil 1 : Overload metodlar.

Şekil 1 de MetodA isminde 3 adet metod tanımı görüyoruz. Bu metodlar aynı isime sahip olmasına rağmen imzaları nedeni ile birbirlerinden tamamıyla farklı metodlar olarak algılanırlar. Bize sağladığı avantaj ise, bu metodları barındıran bir sınıf nesnesi yarattığımızda aynı isme sahip metodları farklı parametreler ile çağırabilmemizdir. Bu bir anlamda her metoda farklı isim vermek gibi bir karışıklığında bir nebze önüne geçer. Peki imza dediğimiz olay nedir? Bir metodun imzası şu unsurlardan oluşur.

Metod İmzasın Kabul Edilen Unsurlar	Metod İmzası Kabul Edilmeyen Unsurlar
<ul style="list-style-type: none">Parametre sayısıParametrelerin tipleri	<ul style="list-style-type: none">Metodun geri dönüş tipi

Tablo 1. Kullanım Kuralları

Yukarıdaki unsurlara dikkat ettiğimiz sürece dilediğimiz sayıda aşırı yüklenmiş (overload edilmiş) metod yazabiliriz.

Şimdi dilerseniz küçük bir Console uygulaması ile , overload metod oluşumuna engel teşkil eden duruma bir göz atalım. Öncelikle metodun geri dönüş tipinin metodun imzası olarak kabul edilemeyeceğinden bahsediyoruz. Aşağıdaki örneğimizi inceleyelim.

```
using System;

namespace Overloading1
{
    class Class1
    {
        public int Islem(int a)
        {
            return a*a;
        }

        public string Islem(int a)
        {
            string b=System.Convert.ToString(a);
            return "Yasim:"+b;
        }

        [STAThread]
        static void Main(string[] args)
        {
        }
    }
}
```

```
}  
}
```

Örneğin yukarıdaki uygulamada, Islem isimli iki metod tanımlanmıştır. Aynı parametre tipi ve sayısına sahip olan bu metodların geri dönüş değerlerinin farklı olması neden ile derleyici tarafından farklı metodlar olarak algılanmış olması gerektiği düşünülebilir. Ancak böyle olmamaktadır. Uygulamayı derleme çalıştığımızda aşağıdaki hata mesajı ile karşılaşırız:

Overloading1.Class1' already defines a member called 'Islem' with the same parameter types

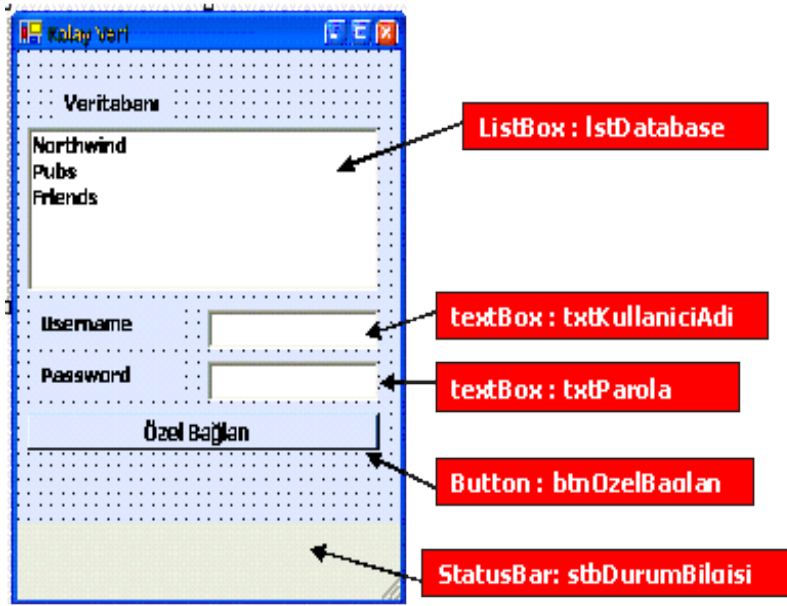
Yapıcı metodları da overload edebiliriz. Bu nokta oldukça önemlidir; ki metodları overload etmeyi .NET ile yazılım geliştirirken sıkça kullanırız. Örneğin, SqlConnection sınıfından bir nesne örneği oluşturmak istediğimizde bunu yapabileceğimiz 2 tane overload edilmiş yapıcı metod olduğunu görürüz. Bunlardan birisi aşağıda görülmektedir.

```
SqlConnection con=new SqlConnection(  
    1 of 2 SqlConnection (string connectionString)  
    connectionString:  
    The connection used to open the SQL Server database.  
Thread]
```

Şekil 2. Örnek bir Overload Constructor(Aşırı Yüklenmiş Yapıcı) metod.

Dolayısıyla bizde yazdığımız sınıflara ait constructorları overload edebiliriz. Şimdi dilerseniz overload ile ilgili olarak kısa bir uygulama geliştirelim. Bu uygulamada yazdığımız bir sınıfa ait constructor metodları overload ederek değişik tipte fonksiyonellikler edinmeye çalışacağız.

Bu uygulamada KolayVeri isminde bir sınıfımız olacak. Bu sınıfın üç adet yapıcısı olacak. Yani iki adet overload constructor yazacağız. İki tane diyorum çünkü C# zaten default constructoru biz yazmasak bile uygulamaya ekliyor. Bu default constructorlar parametre almayan constructorlardır. Overload ettiğimiz constructor metodlardan birisi ile, seçtiğimiz bir veritabanına bağlanıyoruz. Diğer overload metod ise, parametre olarak veritabanı adından başka, veritabanına bağlanmak için kullanıcı adı ve parola parametrelerininide alıyor. Nitekim çoğu zaman veritabanlarımızda yer alan bazı tablolara erişim yetkisi sınırlamaları ile karşılaşabiliriz. Bu durumda bu tablolara bağlantı açabilmek için yetkili kullanıcı adı ve parolayı kullanmamız gerekir. Böyle bir olayı canlandırmaya çalıştım. Elbetteki asıl amacımız overload constructor metodların nasıl yazıldığını, nasıl kullanıldığını göstermek. Örnek gelişmeye çok, hemde çok açık. Şimdi uygulamamızın bu ilk kısmına bir gözatalım. Aşağıdakine benzer bir form tasarım yapalım.



Şimdi sıra geldi kodlarımızı yazmaya. Öncelikle uygulamamıza KolayVeri adında bir class ekliyoruz. Bu class'ın kodları aşağıdaki gibidir. Aslında uygulamaya bu aşamada baktığımızda SqlConnection nesnemizin bir bağlantı oluşturmasını özelleştirmiş gibi oluyoruz. Gerçekten de aynı işlemleri zaten SqlConnection nesnesini overload constructor'ları ile yapabiliyoruz. Ancak temel amacımız aşırı yüklemeyi anlamak olduğu için programın çalışma amacının çok önemli olmadığı düşüncesindeyim. Umuyorum ki sizlere aşırı yükleme hakkında bilgi verebiliyor ve vizyonunuzu geliştirebiliyorumdur.

```
using System;
using System.Data.SqlClient;

namespace Overloading
{
    public class KolayVeri
    {
        /* Connection'in durumunu tutacak ve sadece bu class içinde
        geçerli olan bir string değişken tanımladık. private anahtar kelimesi
        değişkenin sadece bu class içerisinde yaşayabilceğini belirtir.
        Yazmayabiliriz de, nitekim C# default olarak değişkenleri private kabul
        eder.*/
        private string baglantiDurumu;

        /* Yukarıda belirttiğimiz baglantiDurumu isimli değişkenin sahip olduğu
        değeri, bu class'a ait nesne örneklerini kullandığımız yerde görebilmek
        için sadece okunabilir olan (readonly), bu sebeplede sadece Get bloğuna
        sahip olan bir özellik tanımlıyoruz.*/
        public string BaglantiDurumu
        {
            get
            {
                /* Bu özelliğe eriştiğimizde baglantiDurumu değişkeninin
                o anki değeri geri döndürülecek. Yani özelliğin
                çağırıldığı yere döndürülecek.*/
                return baglantiDurumu;
            }
        }
    }
}
```

```

    }
}

/* Iste C# derleyicisinin otomatik olarak eklediği parametresiz yapıcı
metod. Biz bu yapıcıya tek satırlık bir kod ekliyoruz. Eğer nesne örneği
parametresiz bir Constructor ile yapılırsa bu durumda bağlantının kapalı
olduğunu belirtmek için baglantiDurumu değişkenine bir değer atıyoruz. Bu
durumda uygulamamızda bu nesne örneğinin BaglantiDurumu özelliğine
eristiğimizde BAĞLANAMADIK değerini elde edeceğiz.*/
public KolayVeri()
{
    baglantiDurumu="BAĞLANAMADIK";
}

/* Bizim yazdığımızı aşırı yüklenmiş ilk yapıcı metoda gelince. Burada
yapıcımız, parametre olarak bir string alıyor. Bu string veritabanının
adını barındırıcak ve SqlConnection nesnemiz için gerekli bağlantı
stringine bu veritabanının adını geçirecek.*/
public KolayVeri(string veritabaniAdi)
{
    string connectionString="initial
catalog="+veritabaniAdi+";data source=localhost;integrated
security=sspi";
    /* SqlConnection bağlantımız yaratılıyor.*/
    SqlConnection con=new SqlConnection(connectionString);

/* Bağlantı işlemini bir try bloğunda yapıyoruz ki, herhangi bir nedenle
Sql sunucusuna bağlantı sağlanamazsa (örneğin hatalı veritabanı adı
nedeni ile) catch bloğunda baglantiDurumu değişkenine BAĞLANAMADIK
değerini atıyoruz. Bu durumda program içinde KolayVeri sınıfından örnek
nesnenin BaglantiDurumu özelliğinin değerine baktığımızda BAĞLANAMADIK
değerini alıyoruz böylece bağlantının sağlanamadığına kanaat getiriyoruz.
Kanaat dedikte aklıma Üsküdar'da ki Kanaat lokantası geldi :) Yemekleri
çok güzeldir. Sanırım karnımız acıktı... Neyse kaldığımız yerden devam
edelim.*/
    try
    {
        con.Open(); // Bağlantımız açılıyor.
        /* BaglantiDurumu özelliğimiz (Property), baglantiDurumu
değişkeni sayesinde BAĞLANDIK değerini alıyor.*/
        baglantiDurumu="BAGLANDIK";
    }
    /* Eğer bir hata olursa baglantiDurumu değişkenine
BAĞLANAMADIK değerini atıyoruz.*/
    catch(Exception hata)
    {
        baglantiDurumu="BAĞLANAMADIK";
    }
}

/* Sıra geldi ikinci overload constructor metoda. Bu metod ekstradan iki
parametre daha alıyor. Bir tanesi user id'ye tekabül edecek olan
kullaniciAdi, diğeri ise bu kullanıcı için password'e tekabül edecek olan
parola. Bunları SqlConnection'in connection stringine alarak ,
veritabanına belirtilen kullanıcı ile giriş yapmış oluyoruz. Kodların
işleyişi bir önceki metodumuz ile aynı.*/
public KolayVeri(string veritabaniAdi,string kullaniciAdi,string
parola)

```

```

        {
            string connectionString="initial
catalog="+veritabaniAdi+";data source=localhost;user
id="+kullaniciAdi+";password="+parola;
SqlConnection con=new SqlConnection(connectionString);
try
{
    con.Open();
    baglantiDurumu="BAGLANDIK";
}
catch(Exception hata)
{
    baglantiDurumu="BAGLANAMADIK";
}
}
}
}

```

Şimdi sıra geldi, formumuz üzerindeki kodları yazmaya.

```
string veritabaniAdi;
```

```
private void lstDatabase_SelectedIndexChanged(object sender,
System.EventArgs e)
{
```

```
    veritabaniAdi=lstDatabase.SelectedItem.ToString();
```

```
    /* Burada kv adında bir KolayVeri sınıfından nesne örneği (object
instance) yaratılıyor. Dikkat edecek olursanız burada yazdığımı
ikinci overload constructor'u kullandık.*/
```

```
    KolayVeri kv=new KolayVeri(veritabaniAdi);
```

```
    /* Burada KolayVeri( dediğimizde .NET bize kullanabileceğimiz aşırı
yüklenmiş constructorları aşağıdaki şekilde olduğu gibi
hatırlatacaktır. IntelliSense'in gözünü seveyim.*/
```

```
private void lstDatabase_SelectedIndexChanged(object sender, System.EventArgs e)
{
    string veritabaniAdi=lstDatabase.SelectedItem.ToString();
    KolayVeri kv=new KolayVeri(|
    stbDurumBilgi[2 of 3] KolayVeri.KolayVeri(string veritabaniAdi)String() +" "+kv.BaglantiDurumu;
    KolayVeri kvOzel=new KolayVeri(
}

```

Sekil 4. 2nci yapıcı

```
    stbDurumBilgisi.Text=lstDatabase.SelectedItem.ToString()+"
"+kv.BaglantiDurumu;
}

```

```
private void btnOzelBaglan_Click(object sender, System.EventArgs e)
{
    string kullanıcı,sifre;
```

```

kullanici=txtKullaniciAdi.Text;
sifre=txtParola.Text;
veritabaniAdi=lstDatabase.SelectedItem.ToString();

KolayVeri kvOzel=new KolayVeri(veritabaniAdi,kullanici,sifre);
/* Burada ise diğer aşırı yüklenmiş yapıcımızı kullanarak bir
KolayVeri nesne örneği oluşturuyoruz.*/

private void lstDatabase_SelectedIndexChanged(object sender, System.EventArgs e)
{
    string veritabaniAdi=lstDatabase.SelectedItem.ToString();
    KolayVeri kv=new KolayVeri(veritabaniAdi);
    stbDurumBilgisi.Text=lstDatabase.SelectedItem.ToString()+" "+kv.BaglantiDurumu;

    KolayVeri kvOzel=new KolayVeri(|
}

```

3 of 3 KolayVeri.KolayVeri (string veritabaniAdi, string kullaniciAdi, string parola)

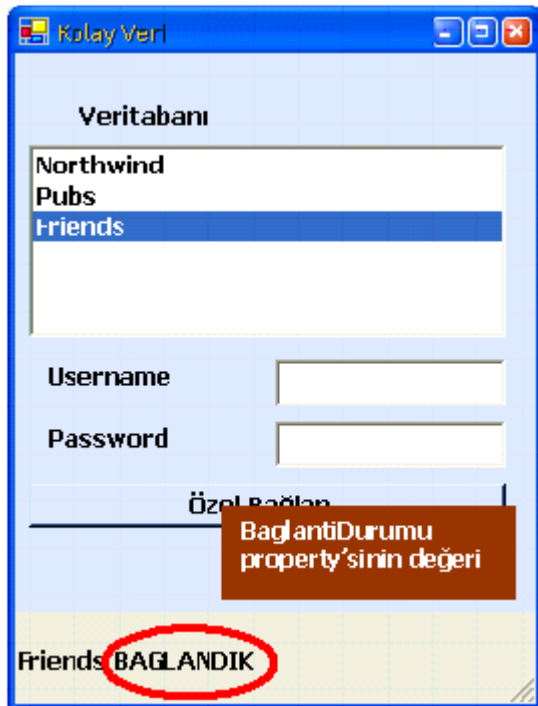
Şekil 5. 3ncü yapıcı

```

stbDurumBilgisi.Text=lstDatabase.SelectedItem.ToString()+"
"+kvOzel.BaglantiDurumu+" User:"+kullanici;
}

```

Evet şimdide programın nasıl çalıştığına bir bakalım. Listbox nesnesi üzerinde bir veritabanı adına bastığımızda bu veritabanına bir bağlantı açılır.



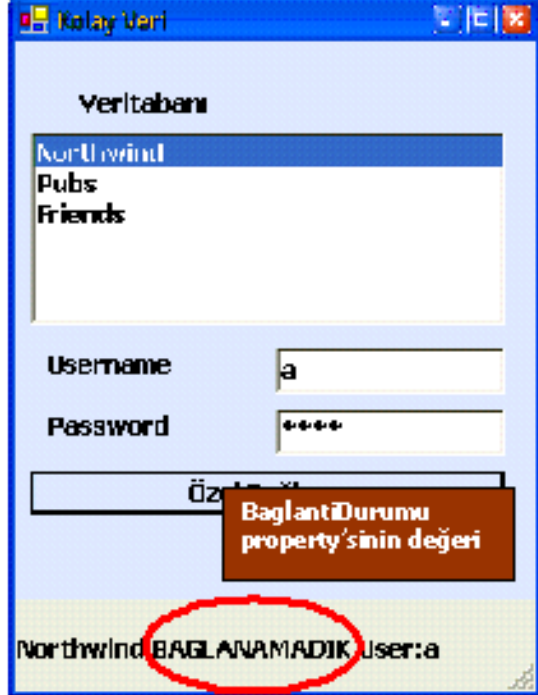
Şekil 6. Listboxta tıklanan veritabanına bağlandıktan sonra.

Ve birde kullanıcı adı ile parola verilerek nasıl bağlanacağımızı görelim.



Şekil 7. Kullanıcı adı ve parola ile bağlantı

Peki ya yanlış kullanıcı adı veya parola girersek?



Şekil 8. Yanlış kullanıcı adı veya parolası sonrası.

Evet değerli okuyucular bu seferlikte bu kadar. Bir sonraki makalemizde görüşmek dileğiyle hepinize mutlu günler, yarınlar dilerim.

C#'ta Numaralandırıcılar(Enumerators)

Bu makalemizde, kendi değer türlerimizi oluşturmanın yollarından birisi olan Enumerator'ları inceleyeceğiz. C# dilinde veri depolamak için kullanabileceğimiz temel veri türleri yanında kendi tanımlayabileceğimiz türlerde vardır. Bunlar Structs(Yapılar), Arrays(Diziler) ve Enumerators(Numaralandırıcılar)'dır.

Numaralandırıcılar, sınırlı sayıda değer içeren değişkenler yaratmamıza olanak sağlarlar. Burada bahsi geçen değişken değerleri bir grup oluştururlar ve sembolik bir adla temsil edilirler. Numaralandırıcıları kullanma nedenlerimizden birisi **verilere anlamlar yükleyerek, program içerisinde kolay okunabilmelerini ve anlaşılabilmelerini sağlamaktır.**

Örneklerimizde bu konuyu çok daha iyi anlayacaksınız. Bir Numaralandırıcı tanımlamak için aşağıdaki sözdizimi (syntax) kullanılır.

```
<Kapsam belirteçleri> enum <numaralandırıcıAdi>
{
    <birinciUye>,
    <ikinciUye>,
    <ucuncuUye>,
}
```

Kapsam belirteçleri protected, public, private, internal yada new değerini alır ve numaralandırıcının geçerli olduğu kapsamı belirtir. Dikkat edilecek olursa, elemanlara herhangi bir değer ataması yapılmamıştır. Nitekim bu numaralandırıcıların özelliğidir. İlk eleman 0 değerine sahip olmak üzere diğer elemanlar 1 ve 2 değerlerini sahip olacak şekilde belirlenirler. Dolayısıyla programın herhangi bir yerinde bu numaralandırıcıya ait elemana ulaştığımızda, bu elemanın index değerine erişmiş oluruz. Gördüğünüz gibi numaralandırıcı kullanmak okunurluğu arttırmaktadır.

Dilersek numaralandırıcı elemanlarının 0 indexinden değil de her hangibir değere bağlamasını sağlayabilir ve hatta diğer elemanlarada farklı index değerleri atayabiliriz. Basit bir numaralandırıcı örneği ile konuyu daha iyi anlamaya çalışalım.

```
using System;

namespace enumSample1
{
    class Class1
    {

        /* Haftanın günlerini temsil edicek bir numaralandırıcı tipi oluşturuyoruz.
        Pazartesi 0 index değerine sahip iken Pazar 6 index değerine sahip
        olacaktır.*/

        enum Gunler
        {
            Pazartesi,
            Salı,
            Carsamba,
            Persembe,
            Cuma,
            Cumartesi,
        }
    }
}
```

```

        Pazar
    }

    static void Main(string[] args)
    {
        Console.WriteLine("Pazartesi gününün degeri={0}",
(int)Gunler.Pazartesi);

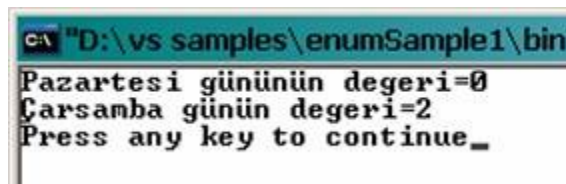
        Console.WriteLine("Çarsamba günün degeri={0}",
(int)Gunler.Carsamba);
    }
}
}

```

Burada Gunler yazdıktan sonra VS.NET 'in intellisense özelliği sayesinde, numaralandırıcının sahip olduğu değerlere kolayca ulaşabiliriz.



Programı çalıştıracak olursak aşağıdaki ekran görüntüsünü elde ederiz:



Şimdi başka bir örnek geliştirelim. Bu kez numaralandırıcının değerleri farklı olsun.

```

using System;

namespace enumSample
{
    class Class1
    {

        enum Artis
        {
            Memur = 15,
            Isci = 10,
            Muhendis = 8,

```

```

        Doktor = 17,
        Asker = 12,
    }

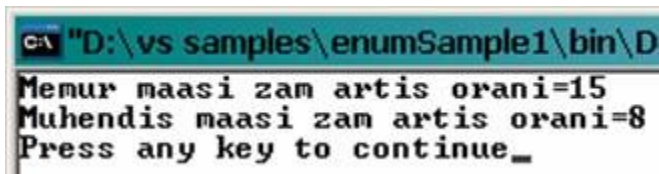
    static void Main(string[] args)
    {
        Console.WriteLine("Memur maasi zam artis orani={0}",
(int)Artis.Memur);

        Console.WriteLine("Muhendis maasi zam artis orani= {0}",
(int)Artis.Muhendis);
    }

}

}

```



Dikkat edicek olursak, numaralandırıcıları program içinde kullanırken, açık olarak(explicit) bir dönüşüm yapmaktayız. Şu ana kadar numaralandırıcı elemanlarına integer değerler atadık. Ama dilersek Long tipinden değer de atayabiliriz. Fakat bu durumda enum'in değer türünde belirtmemiz gerekmektedir.Örneğin:

```

using System;

namespace enumSample
{
    class Class1
    {

        enum Sinirlar: Long
        {
            EnBuyuk = 458796452135L,
            EnKucuk = 255,
        }

        static void Main(string[] args)
        {
            Console.WriteLine("En üst sinir={0}", (long)Sinirlar.EnBuyuk);

            Console.WriteLine("Muhendis maasi zam artis orani={0}",
(long)Sinirlar.EnKucuk);
        }

    }


}

```

Görüldüğü gibi Sınırlarlar isimli numaralandırıcı long tipinde belirtilmiştir. Bu sayede numaralandırıcı elemanlarına long veri tipinde değerler atanabilmektedir. Dikkat edilecek bir diğer nokta ise, bu elemanlara ait değerleri kullanırken, long tipine dönüştürme yapılmasıdır.

Bir numaralandırıcı varsayılan olarak integer tiptedir. Bu nedenle integer değerleri olan bir numaralandırıcı tanımlanırken int olarak belirtilmesine gerek yoktur.

Şimdi daha çok ise yarar bir örnek geliştirmeye çalışalım. Uygulamamız son derece basit bir forma sahip ve bir kaç satır koddan oluşuyor. Amacımız numaralandırıcı kullanmanın programcı açısından işleri daha da kolaylaştırıyor olması. Uygulamamız bir Windows Application. Form tasarımı aşağıdaki gibi olacak.



Form yüklenirken Şehir Kodlarının yer aldığı comboBox kontrolümüz otomatik olarak numaralandırıcının yardımıyla doldurulacak. İşte program kodları:

```
using System;

namespace enumSample
{
    class Class1
    {
        enum AlanKodu: Long
        {
            Anadolu=216,
            Avrupa=212,
            Ankara=312,
            Izmir=412
        }

        private void Form1_Load(object sender, System.EventArgs e)
        {
            comboBox1.Items.Add(AlanKodu.Anadolu);

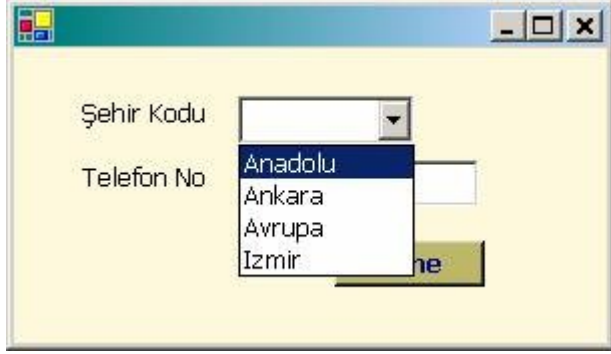
            comboBox1.Items.Add(AlanKodu.Ankara);

            comboBox1.Items.Add(AlanKodu.Avrupa);

            comboBox1.Items.Add(AlanKodu.Izmir);
        }
    }
}
```

```
}  
  
}
```

İşte sonuç:



Aslında bu comboBox kontrolünü başka şekillerde de alan kodları ile yükleyebiliriz. Bunu yapmanın sayısız yolu var. Burada asıl dikkat etmemiz gereken nokta numaralandırıcı sayesinde bu sayısal kodlarla kafamızı karıştırmak yerine daha bilinen isimler ile aynı sonuca ulaşmamızdır.

Geldik bir makalemizin daha sonuna. İlerliyen makalelerimizde bu kez yine kendi değer tiplerimizi nasıl yaratabileceğimize struct kavramı ile devam edeceğiz. Hepinize mutlu günler dilerim.

Microsoft .NET Programlama Dilleri

Bu yazıda Microsoft dotNET ile birlikte programcılara sunulun geliştirme dilleri üzerinde durulmuştur. Bu yazı Temmuz 2003'te MSDN de yayınlanan [Microsoft Programming Languages](#) yazısından Türkçe'ye çevrilmiştir.

Microsoft .NET'in Yararları

Microsoft . NET Framework ,XML Web Servisi ve uygulamalarının derlenip çalıştırılması için gerekli olan Microsoft Windows® bileşenlerini içerir. Bu; bize geliştirdiğimiz uygulamalarda

- yüksek bir verim ,
- standart bir alt yapı,
- daha basit uygulama geliştirme ,
- çoklu dillerin(multilanguage) bulunduğu bir ortam ,
- var olan programlama bilgilerinden yararlanabilme,
- var olan uygulamalar ile kolay entegre olabilme,
- internet uygulamalarında kullanabilme ve çalıştırmanın rahatlığını getirir.

.NET Framework, iki ana bölümden oluşur : CLR(Common Language Runtime) ve web tabanlı programlamada devrimsel bir gelişme yaratan birleştirilmiş sınıf kütüphanesi(ASP.NET), akıllı istemci uygulamalarını gerçekleştirmek için Windows formları, ortam ve temel veri girişleri için ADO.NET alt sistemi. Programcılar iki farklı dille uygulama geliştirirken rahatlıkla .NET Framework kullanabilirler. Bu dillerin hepsi (MSIL)'e derlenir ve daha sonra native(ana) koda dönüştürülür ve CLR ortamında çalıştırılır. Bu sayede herhangi bir dille yazılan herhangi bir uygulama başka bir uygulama ile kolaylıkla entegre olabilir. Bu ortamın programcılara sağladığı yarar şudur; işlerini yaparken kullanmaları için geniş bir dil seçeneği vardır ve dolayısıyla programcılar en iyi bildikleri dili seçebilirler.

Geniş Bir Dil Seçeneği

Sanatçılar çalışırken bulundukları ortam ve araçlar, sanatçıların tecrübelerini ve kişiliklerini yansıtır. Aynı yazılım uzmanları gibi onlarda bildikleri dili ve eğitimlerini göz önüne alarak çalışırlar. Tüm yazılım uzmanlarını memnun edecek bir dil henüz yoktur. Programcılar, doğuştan farklı bir kimliklerdir: kısmen bilim adamı, kısmen sanatçı, her zaman dik kafalı, ve hep daha iyisini araştıran insanlardır. Buna rağmen modern programlama dillerinin eksikliklerini kabul ederler.

"Biz her zaman, yarım milyon veya 50 milyon VB(Visual Basic) bilen programcıya sahip olacağız. Bizim, .NET 'de VB miz var. Ve bizim şimdi, . NET'de Java dilimiz ve hatta,

COBOL dilimiz bile var! Bunun ne demek olduğunu tahmin et? " -Tim Huckaby, başkan ve CEO, Interknowlogy

Programlama dilleri personel alımında önemli bir faktördür. Bir yazılım uzmanının bildiği dilden başka bir dili kullanmaya geçmesi zordur. .NET yapısının gücü ile bir kaç dili içinde barındıran bir platform sağlar. Programcılar bu platform da-C++, Objective C, Fortran, COBOL, Visual Basic®, Perl -'in her biri ile güçlü yazılım geliştirilebilirler.

Bu bölümde Microsoft'un dört farklı programlama dili sunduğunu ve bunları tek bir ortamda nasıl birleştirdiğini göreceğiz.

- Visual Basic.NET , dünyanın en popüler geliştirme aracının ve dilinin en son uyarlamasıdır. Visual Basic.NET, vazgeçilemez verimliliği teslim eden, ve task(görev)-oriented(yönelimli) programlama için eşsiz özellikler sunan bir dildir.
- Visual C++ dili, power(güç)-oriented(yönelimli) güçlü uygulamalar geliştirilebilen , farklı teknolojilerle köprü kurabilen, hem windows doğal diline hemde(assembly) .NET ara diline(IL) derlenebilen maksimum performans karakteristiklerine ve yüksek fonksiyonalliteye sahip bir dil olarak karşımıza çıkar.
- Visual C #®.NET , modern ve yenilik getiren programlama dili ve geliştirme aracıdır. Microsoft, C# ile bizi 2001'de tanıştırdı, C++ ve Java programcılarının bildiği bir sentaks(sözdizimi) ile sunulmuştu, bu yüzden C++ ve Java geliştiricilerinin ilgisini çekmiş olan C#, .NET Framework ile beraber kod odaklı uygulamaları daha düzenli bir dil yapısı ile sunar.
- Visual J#®.NET , Microsoft .NET için Java-dili geliştirme aracıdır. Visual J#.NET, Visual J++ ve Java geliştiricilerine kendi dil ve söz dizimlerinden uzaklaşmadan .NET'in olanaklarından tam olarak yararlanabilmeyi ve endüstrinin en gelişmiş XML web servisleri platformundan faydalanabilmelerini sağlamıştır.

Burada Microsoft, programlama dillerindeki geniş dil seçeneğinin geliştiricilere uygunluğu ile dikkat çeker. Microsoft.NET, programcılar bu yeni platformda birleştirmek için eğitim ve çözüme daha hızlı ulaşmayı sağlamada onlara yardım etme imkanı da sunar.

Visual Basic .NET

Visual Basic 1.0, Windows'un gelişmesi ve daha geniş bir kullanıcı sayısına ulaşması ve günümüzdeki verimliliğine kavuşması için bir devrim yarattı. Böyle zengin bir tarihe sahip olan VB , okunabilir bir syntax, sezgisel bir arayüzün ve aletlerin olduğu , task-oriented(görev yönelimli) yapısı ile hızlı build edilebilmesi ile ve .Net ile yeni bir yapılandırmaya kavuşmuştur. Visual Basic.NET dilinde diğer popüler diller gibi her tür Windows uygulaması, WEB,ve mobile aletler için uygulama yapabilme yetenekleri eklendi.

Task-Oriented(görev yönelimli) Gelişme

Deadline(Son teslim) tarihleri, yazılım sanayisine yeni olan bir şey değildir. Programcılar büyük bir grubu için, deadline tarihleri, günlük hayatın bir gerçeğidir. Programcılar çoğunlukla, plan yaparken ,işin gereksinimlerini karşılayacak hızlı bir yolun çözümünü ararlar. Bazi çözümler , bu ortamlar yaratılırken dikkatlice test edilecek, daha sonra uygulama bu tarz yapılarda hemen kullanılacaktır. Uygulama geliştirme uzmanının problemlerde çözüme odaklanması için bir dahaki iş verilene kadar serbest bırakılması gerekir. Task-Oriented bir geliştirme ortamı, ortam farklılıklarından programcıyı kurtarmak için kabul edilebilir bir yöntemdir.

Hangi Programcılar Neden VB.NET'i Seçmelidir?

Gelecek kuşak uygulamaları ve servisleri birleştirerek .NET ortamında araştırma yapmak isteyen aşağıdaki tipteki programcılar için Visual Basic.NET ideal bir dildir.

- **.NET Framework ortamında hızlı ve üretken bir araçla uygulama geliştirmek isteyen programcılar** : .Net ile birlikte hızlı ve rahat bir geliştirme aracı sunulurken Visual Basic.NET uygulamalarında kolay syntax ve sezgiyle elde edilen geliştirme ortamını sunar. Ayrıca VB.NET programcıları ilgili kaynaklara çok hızlı bir şekilde erişebime imkanına sahiptir.

- **VB ile uygulama geliştirme tecrübesi olan programcılar** : Visual Basic bilen programcılar için Visual Basic.Net yapısına geçmek zor olmayacaktır. Visual Basic .NET anahtar sözcükleri, sentaks ve derleme yapısı ile farklılıklar oluştursa da, geleneksel VB programcısına tanıdık gelecektir. VB.NET te ayrıca case-insensitivity, anlaşılır kodlar ve sentaksı vardır. Visual Basic'in ilk versiyonlarını kullananlarda .NET kullanmaya yönelebilirler. Mevcut bulunan ActiveX kontrollerini de Visual Basic .NET te kullanmaya devam edebilirler.

- **Halihazırdaki ortamlara benzer tasarım zamanı ve kod editörü paradigmaları arayan programcılar** : Geliştiriciler bildik bir arayüz ve editor ararken, VB.NET ile uygulamaların tasarımı drag and drop ile yapılabilir. Ayrıca otomatik, kolay biçimlendirilmiş bir kodlama sunar.

- **Sezgisel ve erişebilirlik özelliği yüksek olan bir dille geliştirme yapmak isteyen programcılar** : Visual Basic.NET, geliştiricilerin büyük bir bölümüne ulaşabilmek için tasarlanmıştır. Bu nedenle hem uzmanlara, hem de yeni başlayanlara önerilir. Yeni başlayanlar, Visual Basic ortamının ve Visual Basic dilinin birçok benzersiz özelliğini faydalı bulacaktır.

VB.NET Diline Has Özellikler

VB.NET, uygulama üretkenliğini hızlandıracak diğer .NET dillerinde olmayan bir takım özellikleri içinde barındırır.

- **Değişkenlere varsayılan ilk değer verme** :VB.NET'te değişkenleri kullanılmadan önce onlara ilk değer verilme zorunluluğu yoktur. Bu yüzden yeni başlayan bir çok programcının diğer dillerde olduğu gibi kafası karışmayacaktır.

- **Implicit typing(dolaylı tür belirtme) ve geç bağlama(late binding)** :Visual Basic.Net te bir değişken kullanılmadan önce onun tipini belirtmek zorunda değiliz. Bu da programcıya daha az eforla daha kullanışlı kod yazmasında yardım eder.

- **Numaralandırıcıların Davranışı** :Visual Basic. NET,Enumeration tipleri kullanmak gerektiğinde .NET ortami sezgiyle bunu programcıya getirir.

- **Varsayılan public erişimi** :Visual Basic.NET sınıfının üye elemanları varsayılan olarak public olduğu için programcılara sezgisel gücü fazla olan bir özellik sunar.

- **Shared üye elemanlarını kullanma** :Shared(C# taki static) üye elemanlarına hem sınıfın ismi üzerinden hem de ilgili sınıf nesnelerinden erişilebilir. Bu da programcıya daha esnek bir yapı sunar. Örneğin

Dim x as new MyClass

x.SharedMethod() ' ile birlikte

MyClass.SharedMethod()

ifadesi de geçerlidir.

- **Opsiyonel parametreler** : Visual Basic.NET programcıları nesne yönelimli programlama tekniğinin bütün nüanslarını bilmeye gerek kalmadan esnek sınıf yapıları tasarlayabilir. Örneğin sınıf tasarımcıları opsiyonel parametreleri kullanarak daha esnek sınıflar tasarlayabilirler.
- **Filtrelenmiş catch blokları** :VB.NET istisnai durumları ele almada esnek bir yapı sunar. Filtrelenmiş catch blokları sayesinde programcılar hataları, hata numarasına göre, istisnalarının türüne göre yada herhangi bir koşullu ifadeye göre filtreleyebilirler.
- **Parametrelili Özellikler** : Visual Basic.NET te özellikler C# taki karşılığından farklı olarak parametrelili olabilir. Böylece özellikler daha esnek bir yapı sunmuş olur.
- **Declarative event handlers** : Visual Basic.NET'te olaylara ilişkin metotları bildirirken Handles anahtar sözcüğü kullanılır.
- **Arayüz üye elemanlarının yeniden bildirilmesi** : VB.NET'te implemente edilen bir arayüzün üye elemanının ismi arayüzü uygulayan sınıf tarafından değiştirilebilir.

VB.NET Geliştirme Ortamına Has Özellikler

Visual Basic.NET programcılara daha çok yarar sağlayacak bir tasarıma, uygulama ve servis yazmakta kolay bir ortam sağlar. Bu sadece Visual Basic.NET'i değil tüm .NET platformunu kapsar.

Arka planda kodun derlenmesi : Geliştirme ortamı siz çalışırken arka planda kodunuzu derler ve eğer kodda hata varsa bunu size listeler.

Visual Basic . NET otomatik olarak yazdığınız kodu düzenler ve kaydeder. Otomatik olarak düzenlerken kodun durumu , anahtar sözcüklerin durumunu ve değişkenleri hizalayabilir. Bu da çok fazla ifadenin kullanıldığı durumlarda yanlış ifadelerin yada formatsız ifadelerin düzgün görülmesini sağlar.

Performans

Son önemli nokta performanstır. Visual Basic . NET derleyicisinin ürettiği ara kod, C# derleyicisinin ürettiği IL kodu ile aynı performansa sahiptir.

Visual C++

Çoğunlukla yazdığı programlardan güç beklentisi olan programcıların bu platformun tüm özelliklerinden faydalanması mümkündür. CLR ve . NET altyapısının pek çok faydalarına rağmen, bazı programcılar hala uygulamaları geliştirirken, Windows işletim sisteminin tüm genişlik ve derinliklerine güvenerek uygulamalarını oluştururlar. Geleneksel olarak programcılar, sistem verimliliğini en iyi kullanan kodu yazmak sistem tarafından sağlanan kaynaklara(disk,hafıza) en etkili erişimi sağlamak amacıyla Visual C++ ortamını seçmişlerdir. Visual C++.NET bu geleneksel yöntemlerin devamını sağlamayı hedeflemiştir. Tabi bunu yaparken Win32 API den sıklıkla faydalanır. C++.NET aynı zamanda .NET Framework ve yönetilebilir CLR nin bir çok imkanına erişmeyi de sağlar.

Güç Yönelimli(Power-Oriented) Geliştirme

Bir çok durumda geliştiriciler, işletim sisteminin sağladığı bütün imkanlara erişmek isterler. Microsoft, bu imkanlardan soyutlanmış yada tamamen bu imkanlar üzerine kurulmuş değişik araçlar tasarlamıştır. Bugün itibariyel .NET framework sağlam uygulama geliştirmek için bu imkanların birçoğunu sunarken yinede işletim sisteminin bütün yeteneklerini içinde barındırmaz. Güç yönelimli(Power-Oriented) geliştirme araçları, programcılara bu dilin tüm özelliklerinin yanında, uygulamanın gerektirdiği çözümlerin de kolayca çözüme kavuşabileceği kütüphaneleri sağlar.

Hangi Programcılar Neden VB.NET'i Seçmelidir?

Visual C++, aşağıdaki tipteki programcılar için ideal bir dildir.

- **Win32 tabanlı bileşen ve uygulama geliştirmek isteyen programcılar :**

Günümüzde programcılarının bir grubu, native windows tabanlı uygulamalar yapmaya ihtiyacı vardır. Bu programcılar bu tr uygulamalar için Win32 API yi ve native(doğal) C++ kütüphanelerini kullanırlar. Visual C++.NET 2003, bu tür uygulamaların performansını daha iyi yönde etkileyebilircek bir takım derleme parametrelerine sahiptir.

- **Win32 tabanlı uygulamalar ve .NET ile geliştirilmiş uygulamalar arasındaki boşluğu doldurmak isteyen programcılar :**

Halihazırda yazılmış olan bir çok uygulama, karmaşık kodlar yüzünden, zaman, ücret veya bir çok nedenden dolayı .NET Framework kullanılarak yeniden yazılamayabilir. Visual C++ ile programcılar, var olan uygulamalarının genişleterek devamını .NET Framework çatısı altında geliştirebilirler. Üstelik daha karmaşık bir WinAPI altyapısını da kullanma imkanına kavuşurlar. Microsoft C# ve ya Visual Basic Windows API lerine erişimi sağlarken, C++'a karşı bir rakip olarak tasarlanmadı.

- **Uygulamaların ana olarak performansı ile ilgilenen programcılar :** Uygulama tasarımında ve çalıştırılmasında C++ geliştiriciye geniş bir kontrol imkanı sunar. İleri düzey geliştiriciler C++ kullanarak diğer dillerde geliştirebilecekleri uygulamalardan(native Windows yada .NET tabanlı) daha hızlı ve etkili çalışan uygulama tasarlayıp implemente edebilirler.

- **Farklı platformlar arasında çalışabilecek program geliştirmek isteyen programcılar :** Yalnızca C++ dili ISO standartlarını içerir ve gerçekten taşınabilir sentaksı, her sistemde çalışabilecek bir yapı içerir. Visual C++.NET 2003'ün genişletilmiş standart uyumluluğunu sağlarken aynı zamanda programcılara ileri seviye dil özelliklerini ve bir çok işletim sisteminde bulunan popüler sınıf kütüphanelerini kullanma imkanı sunar.

C++ .NET Diline Has Özellikler

Visual C++. NET, ileri düzey yazılımcılar tarafından büyük bir taleple karşılan kendine has bir takım özelliklere sahiptir. Bu özelliklerin hepsi C++.NET'i .NET dilleri arasında en güçlü kılmaya yetmektedir.

- **Şablonlar(Templates) :** Büyük ölçüde C++ diline has olan şablonlar yeniden kullanılabilirliği ve performans artışı gibi bir çok önemli özelliği sağlamaktadır.

- **Göstericiler(Pointers) :** Göstericiler, C + + geliştiricilerine makinenin yerel hafızasına doğrudan erişebilmesini sağlar ve böylece en yüksek seviyede performans elde edilir.

- **Çoklu Türetme(Multiple Inheritance) :** C++, hemen hemen tüm OOP desenlerini implemente etmeyi sağlayan ve bütün OOP özelliklerini uygulamaya geçirecek özelliklere sahiptir. Çoklu türetme de bu özelliklerden biridir.

- **Intrinsics** : Intrinsics'ler, geleneksel programlama tekniklerinde olmayan bir takım yeni özelliklere erişmeye imkan sunar. Örneğin MMX ve AMD 3D Now! registerları ve komutları.

C++ .NET Geliştirme Ortamına Has Özellikler

Visual C++ .NET 2003 geliştirme ortamında daha esnek ve daha gelişmiş uygulamalar geliştirmek için bir takım özellikler sağlar.

- **Derleyiciyi optimize etmek** : Visual C++ derleyicisi bir çok geliştirme seneryosu için derleme işlemini optimize edebilir. Bu seneryolardan bir kaç : MSIL üretimi, kodun çalışacağı sisteme özgün optimizasyon, floating sayı hesaplamalarının yoğun olduğu derlemeler.

- **Çalışma zamanı kod güvenliği kontrolü** : Programcılar kötü niyetli ataklara karşı derleyicinin ileri seviye özelliklerini kullanarak daha güvenli Windows tabanlı uygulamalar geliştirebilirler.

- **32 bit ve 64 bit desteği** : Visual C++ .NET derleyicisi 32 ve 64 bitlik Intel ve AMD mikroşlemcilerine ve yönelik gömülü mikrpişlemcilere yönelik ölçeklenebilir kod üretebilmektedir.

- **İleri düzey hata raporlama** : Uygulamalar daima programcıların hatalarından etkilenirler. Minidump teknolojisiyle Visual C++ geliştirme ortamı, uygulama geliştirme uzmanlarına hataların kolayca belirlenmesine yardımcı olur. Üstelik derlenmiş kodda bile bu hatalar kolaylıkla rapor edilebilir.

- **Gelişmiş hata ayıklama(debug)** : Visual Studio hata ayıklayıcısı aynı anda hem native hemde yönetilebilir kodda hata ayıklama desteğini benzer bir şekilde sunar.

Gelecekte, Visual C + + . NET Programcılara yardım etmek için daha güçlü özellikleri içerecek.

- **Generics** : Parametrelili kod algoritmalarını yeniden kullanılabilirliğini sağlamayı hedefleyen çalışma zamanı(run-time) teknolojisi.

- **Yönetilen(Manage) tiplerde şablonlar** : Derleme zamanı C++ şablon sentaksının yönetilen(manage) tipler içinde kullanılabilmesi.

Visual C#

Geleneksel Olarak Visual Basic ve Visual C + + farklı spektrumdaki programcılar hedeflemiştir. Visual Basic özellikle üretkenliği ön plana çıkarması ile programcılara daha kolay ve sezgisel bir geliştirme modeli sunuyordu. Öte yandan Visual C++, üretkenliği azaltıyor gibi görünmesine rağmen Windows işletim sisteminin bütün özelliklerini etkili bir şekilde kullanma imkanı tanıyordu.

Bu iki dilin sunduğu imkanlar arasındaki boşluğu doldurmak için Microsoft kod odaklı uygulama geliştirmeyi modern ve yenilikçi bir tarzda ele alan C# dilini geliştirdi. C#, C++ sözdizimine benzer bir şekilde temiz ve güzel bir programlama dili sunarken aynı zamanda Visual Basic dilinin üretkenliğinde korur.

Kod Odaklı Geliştirme

Programcılarının hepsi projelerinde mutlaka belli özelliklerde kod yazarlar. Fakat programcılarının çoğu zamanlarının önemli bir kısmını sihirbaz(wizard),kontrol ve tasarım araçları kullanarak harcarlar ve böylece önemli ölçüde bir üretkenlik sağlarlar. Bu özelliğin programcılar için tek kötü yanı sihirbaz tarafından üretilen kodun anlaşılabilirliğinin az olmasıdır.Fakat programcılar kodlarında anlaşılabilirlik ve verimlilik arasındaki tercihlerinde güveni tercih ettiler.

Ayrıca kod odaklı geliştirme yapan programcılar başkaları tarafından doğru tasarlanmış kodu yeniden yazmaya yönelir ve bu daha az bilgili programcıların pratik olarak iyi kod geliştirebilmesindeki karmaşıklığı düzeltir.

Hangi Programcılar Neden VB.NET'i Seçmelidir?

Gelecek kuşak uygulamaları ve servisleri birleştirerek .NET ortamında araştırma yapmak isteyen aşağıdaki tipteki programcılar için C# ideal bir dildir.

- **Üretkenlik arayan C/C++ ailesindeki diller ile geliştirme yapan programcılar** :C# dili, C++ dili gibi işleç(operator) aşırı yüklerken buna ek olarak numaralandırmalar, küçük-büyük harf duyarlılığı ve component-oriented(bilesen-yönelimli) özellikler olan property,delegate, events ve daha fazlasını içerir. C#, .NET framework ile beraber yüksek verimlilik ,yönetilebilir , daha güvenli ve anlaşılabilir sözdizimi ile yeni özellikler isteyen C++ programcılarına sunulmuş bir dil olarak da düşünülebilir.
 - **Framework tasarımcıları ve yazılım mimarları** : Framework tarafından iyi desteklenen , işleç yükleme özellikleri içeren, güvenli olmayan kodlara ve önceden yazılmış yazılımlara erişimiyle C#, yazılım mimarlarına geniş ve esnek kütüphaneler ve iş parçacıkları tasarlama imkanı sağlar.
 - **Java tabanlı yazılımlar geliştirmiş programcılar** : Java Language Conversion Asistant (JLCA) ile Java programcıları uygulamalarını C# ve .NET Framework'e rahatlıkla taşıyabilirler. JLCA kaynak kod seviyesinde bir analiz yapar ve Java kodunu C# koduna dönüştürür. Dönüştürme işlemi bittiğinde geliştiricinin dikkat etmesi gereken noktalar belirtilir. Böylece taşıma işlemi an az hasarla gerçekleştirilmiş olur.
- ### **C#'ın C ++ Diline Benzer olan Özellikleri**
- C# dili, geleneksel C++ özelliklerinin bir çoğunu desteklemektedir. Bu geleneksel özelliklerin bir çoğu Visual C++ 'taki üretkenliği artırmak için de kullanılmıştır.
- **Tüm CLR türleri için destek** : C# dili tüm CLR veri tiplerini destekler , programcılar çözüm sunarken .NET ortamının yönetilen çalışma ortamının tüm özelliklerinden yararlanırlar.
 - **Referens yolu ile parametre aktarma ve out parametresi** : C# programcıları, parametreleri fonsiyonlara referans yolu ile aktarabilirler ayrıca out parametresi ile değişkenlere ilk değer vermeden onları fonsiyonlara parametre olarak geçirebilirler.
 - **Overloading (Operatör aşırı yükleme)** : Sınıf kütüphanesi tasarımcıları operatörleri aşırı yükleyerek daha sağlam sınıflar tasarlayabilirler.
 - **Using ifadesi** : Programcılar, uygulamalarında bulunan kaynakları daha kontrollü bir şekilde yönetebilmek için using anahtar sözcüğünü kullanırlar.

- **Güvensiz kod(Unsafe code)** : C#, programcılara gösterici kullanma imkanı tanıyarak hafızaya direkt erişimi sağlar. Her ne kadar güvensiz kodda CLR yönetiminde olsa da ileri düzey programcılar uygulamalarının hafıza yönetimi üzerinde söz sahibi olmaları için güvensiz kod yazabilirler. Buna rağmen hafıza üzerinde daha etkili bir kontrol için Visual C++ kullanılması daha çok tavsiye edilir.

- **XML dökümantasyonu** : Programcılar, kodlarına açıklayıcı notlar eklemek için XML formatındaki bildirimleri kullanabilirler.

Öte yandan C# dil tasarımının sınırlarını genişletecek şekilde hızla büyümektedir. C# dilinin tasarımcıları yakın bir gelecekte dile eklemeyi planladıkları bir kaç önemli özellikten bahsetmektedir. Daha modern ve yenilikçi bir yaklaşım sunan bu özellikler şunlardır :

- **Generics** : Varolan kodların yeniden kullanılabilirliğini kolayca sağlayan C++ şablonlarına benzer bir yapı.

- **Erişiciler (Iterators)** : Koleksiyon tabanlı sınıfların elemanları arasında daha hızlı ve kolay bir şekilde dolaşmamızı sağlayacak yapı.

- **Anonim(Anonymous) Metotlar** : Basit görevleri temsilcilerle daha rahat bir şekilde ele alacak yöntem

- **Kısmi(partial) Türler** : Bir kodu farklı dosyalara bölebilecek türler.

Visual J

Microsoft, Visual J# dili ile JAVA dilini .NET ortamına sokmuş oldu. Microsoft Java diline .NET ortamının pratikliğini getirdi ve okullarda müfredatları olan programcılara, öğrencilere, ve profesörlere Java yapısını muhafaza ederek onların .NET e hızlı bir şekilde girmelerini sağladı. Ayrıca J# dili windows tabanlı uygulama geliştiren Visual J++ 6.0 kullanıcılarına kolayca Visual J# .Net ortamına geçebilmelerinde kolaylıklar sağladı.

Java Geliştirme Ortamı

C++ geliştiricilerin sıklıkla karşılaştıkları problemler etkili ve kolay bir sentaks yapısı aynı zamanda benzer OOP fonksiyoneliyeti ile JAVA ile giderilmiştir. Java ile uygulama geliştirenlerin .NET ortamında uygulama geliştirebilmeleri için en uygun dil J# olarak görülmektedir. Java programcılarını dil değiştirmek zorunda kalmadan .NET framework teki bütün olanaklardan hızlı bir şekilde faydalanma imkanına kavuşmuştur.

Hangi Programcılar Neden VB.NET'i Seçmelidir?

Visual J#, aşağıdaki tipteki programcılar için ideal bir dildir.

- **Java-dili geliştiricileri** : Daha önceden Java dilini kullanan bir programcı .NET e geçerken başka bir dili öğrenmek istemeyebilir. Visual J#, .NET platformunun getirdiği özellikler ile java programcılarının bilgilerini kullanabildikleri rahat ve hızlı bir platform sunar.

- **Visual J++ ile kod geliştiren programcılar** : Visual J# ortamı varolan Visual J++ uygulamalarını .NET ortamına sorunsuzca taşıyabilir ve böylece Visual J#.NET kullanmaya başlayan programcılar projelerinde .NET alt yapısının getirdiği pratikliği ve rahatlığı hemen farkedebilirler.

- **Öğrenciler, öğretmenler, ve profösörler** : Öğrenciler ve öğretmenler Computer Science derslerinde Java dilinin basitliğinden faydalanmak için Visual J#.NET dilini kullanabilirler. Visual J#.NET, ileri bilgisayar bilimin bütün gerekliliklerini karşılar.

J#.NET Diline Has Özellikler

Bir çok dilde bulunan özelliklerin çoğunu yapısında içeren J#, daha rahat ve bildik yapısı ile deneyimli Java geliştiricileri için .Net Framework'e yönelik uygulamalar geliştirmek için ideal bir dildir.

- **Java dilinin söz dizimi** : Java geliştiricileri bildik bir dil yapısı ile karşılamak ve aynı zaman .NET in tüm imkanlarından faydalandıklarını görecekler.

- **Sınıf kütüphanesi desteği** : Bağımsız olarak geliştirilen ve bir çok özelliği sunan Java 1.1.4 JDK versiyonundaki kütüphane ile JDK 1.2 java.util de bulunan hemen hemen bütün sınıfları içerir.

- **Özellikler, temsilciler(delegates) ve olaylar(events)** : .NET geleneksel JAVA söz dizimi ile .NET'in güçlü özelliklerinden olan event ,delegate, ve property yapılarını destekler.

- **Javadoc Yorumları** : J#, Javadoc kollarındaki yorumlama stilini destekler. Visual J# .NET, kullanıcıların HTML API belgesini yaratabilmesine olanak kılar.

Visual J#.NET Geliştirme Ortamına Has Özellikler

Visual J#.NET direkt olarak Visual Studio.NET geliştirme ortamına entegre bir şekilde çalışır. Dolayısıyla tasarlama araçları, editörler ve hata ayıklayıcılar Visual J# geliştirme ortamında rahatlıkla kullanılabilir. Ayrıca hazlihazırdaki JAVA programcılarının .NET'e geçişini kolaylaştıracak bir takım araçlar da vardır.

- **Visual J++ Upgrade Wizard** : Visual J++ geliştiricileri projelerini Visual J# ortamı için upgrade edebilirler. Bu sihirbaz proje dosyalarını çevirir ve olası potansiyel sorunlar için kullanıcıyı bilgilendirir.

- **İkili dönüştürücü** : Bu araç, Java byte kodunu, .NET uygulamalarında kullanmak üzere MS.NET assembly lerine dönüştürür.

Özet

Programlama dilleri fakli çözümler için kullanılabilir. Her dil kendi özelliklerini ve belirli bir uygulamanın ihtiyaçlarını karşılayabilecek en uygun ortamı içerir. Microsoft geniş bir dil seçeneğini sunduğu gelişmiş .NET yapısı ile yazılım uygulamalarında daha sağlam ve fonksiyonallite sağlamış bulunmakta.

C#'ta Params ile Değişken Sayıda Parametre ile Çalışma

Bu makalemizde, C# metodlarında önemli bir yere sahip olduğunu düşündüğüm params anahtar kelimesinin nasıl kullanıldığını incelemeye çalışacağız. Bildiğiniz gibi metodlara verileri parametre olarak aktarabiliyor ve bunları metod içersinde işleyebiliyoruz. Ancak parametre olarak geçirilen veriler belli sayıda oluyor. Diyelimki sayısını bilmediğimiz bir eleman kümesini parametre olarak geçirmek istiyoruz. Bunu nasıl başarabiliriz? İşte params anahtar sözcüğü bu noktada devreye girmektedir. Hemen çok basit bir örnek ile konuya hızlı bir giriş yapalım.

```
using System;

namespace ParamsSample1
{
    class Class1
    {
        /* burada Carpim isimli metodumuza, integer tipinde değerler geçirilmesini sağlıyoruz. params anahtarı bu metoda istediğimiz sayıda integer değer geçirebileceğimizi ifade ediyor*/
        public int Carpim(params int[] deger)
        {
            int sonuc=1;

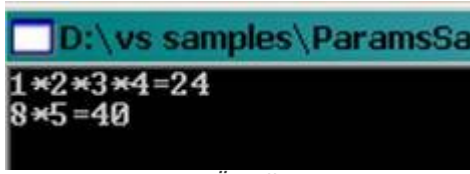
            for(int i=0;i<deger.Length;++i) /*Metoda gönderilen elemanlar doğal olarak bir dizi oluştururlar. Bu dizideki elemanlara bir for döngüsü ile kolayca erişebiliriz. Dizinin eleman sayısını ise Length özelliği ile öğreniyoruz.*/
            {
                sonuc*=deger[i]; /* Burada metoda geçirilen integer değerlerin birbirleri ile çarpılmasını sağlıyoruz*/
            }

            return sonuc;
        }

        static void Main(string[] args)
        {
            Class1 cl=new Class1();
            Console.WriteLine("1*2*3*4={0}",cl.Carpim(1,2,3,4)); /* Burada Carpim isimli metoda 4 integer değer gönderdik. Aşağıdaki kodda ise 2 adet integer değer gönderiyoruz.*/

            Console.WriteLine("8*5={0}",cl.Carpim(8,5));
            Console.ReadLine();
        }
    }
}
```

Bu örneği çalıştıracak olursak, aşağıdaki sonucu elde ederiz.



Şekil 1. İlk Params Örneğinin Sonucu

Peki derleyici bu işlemi nasıl yapıyor birazda ondan bahsedelim. Carpim isimli metoda değişik sayılarda parametre gönderdiğimizde, derleyici gönderilen parametreye sayısı kadar boyuta sahip bir integer dizi oluşturur ve bu dizinin elemanlarına sırası ile (0 indexinden başlayacak şekilde) gönderilen elemanları atar. Daha sonra aynı metodu bu eleman sayısı belli olan diziyi aktararak çağırır. cl.Carpim(8,5) satırını düşünelim; derleyici,

İlk adımda,

```
int[] dizi=new int[2] ile 2 elemanlı 1 dizi yaratır.
```

İkinci adımda,

```
dizi[0]=8
```

```
dizi[1]=5 şeklinde bu dizinin elemanlarını belirler.
```

Son adımda ise metodu tekrar çağırır.

```
cl.Carpim(dizi);
```

Bazı durumlarda parametre olarak geçireceğimiz değerler farklı veri tiplerine sahip olabilirler. Bu durumda params anahtar sözcüğünü, object tipinde bir dizi ile kullanırız. Hemen bir örnek ile görelim. Aynı örneğimize Goster isimli değer döndürmeyen bir metod ekliyoruz. Bu metod kendisine aktarılan değerleri console penceresine yazdırıyor.

```
public void Goster(params object[] deger)
{
    for(int i=0;i<deger.Length;++i)
    {
        Console.WriteLine("{0}. değerimiz={1}",i,deger[i].ToString());
    }

    Console.ReadLine();
}

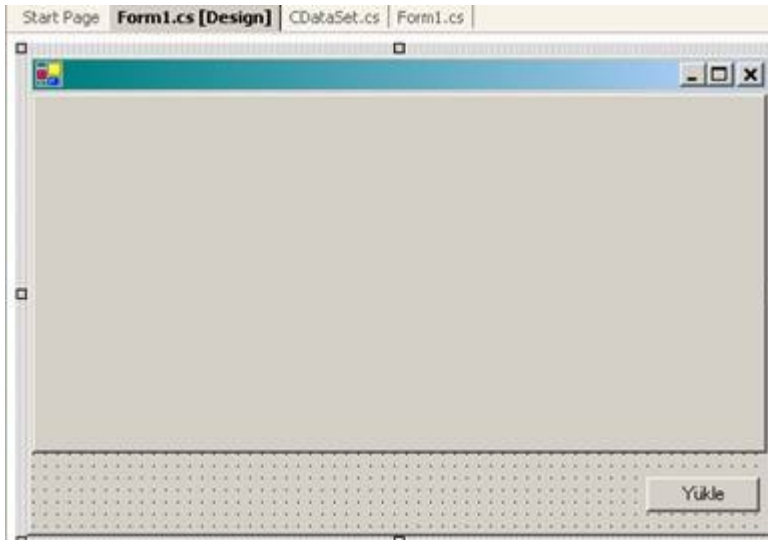
static void Main(string[] args)
{
    cl.Goster(1,"Ahmet",12.3F,0.007D,true,599696969,"C");
}
```

Görüldüğü gibi Goster isimli metodumuza değişik tiplerde(int,Float,Decimal,bool, String) parametreler gönderiyoruz. İşte sonuç;


```
C:\D:\vs samples\ParamsS
1*2*3*4=24
8*5=40
0. degerimiz=1
1. degeriniz=Ahmet
2. degeriniz=12,3
3. degeriniz=0,007
4. degeriniz=True
5. degerimiz=599696969
6. degeriniz=C
```

Şekil 2. params object[] kullanımı.

Şimdi dilerseniz daha işe yarar bir örnek üzerinde konuyu pekiştirmeye çalışalım. Örneğin değişik sayıda tabloyu bir dataset nesnesine yüklemek istiyoruz. Bunu yapıcak bir metod yazalım ve kullanalım. Programımız, bir sql sunucusu üzerinde yer alan her hangibir database'e bağlanıp istenilen sayıdaki tabloyu ekranda programatik olarak oluşturulan dataGrid nesnelere yükleyecek. Kodları inceledikçe örneğimizi daha iyi anlıyacaksınız.



Şekil 3. Form Tasarımımız

Uygulamamız bir Windows Application. Bir adet tabControl ve bir adet Button nesnesi içeriyor. Ayrıca params anahtar sözcüğünü kullanan CreateDataSet isimli metodumuzu içeren CDataSet isimli bir class'ımızda var. Bu class'a ait kodları yazarak işimize başlayalım.

```
using System;
using System.Data;
using System.Data.SqlClient;
```

```
namespace CreateDataSet
{
    public class CDataSet
    {
```

```
        /* CreateDataSet isimli metod gönderilen baglantiAdi stringinin değerine göre bir
        SqlConnection nesnesi oluşturur. tabloAdi ile dataset nesnesine eklemek istediğimizi tablo
        adlarını bu metoda göndermekteyiz. params anahtarı kullanıldığı için istediğimiz sayıda
        tablo adı gönderebiliriz. Elbette, geçerli bir Database ve geçerli tablo adları
        göndermeliyiz.*/
```

```
        public DataSet CreateDataSet(string baglantiAdi, params string[] tabloAdi)
        {
```

```

        string sqlSelect,conString;
        conString="data source=localhost;initial catalog="+baglantiAdi+";integrated
security=sspi"; /* Burada SqlConnection nesnesinin kullanacağı connectionString'i
belirliyoruz.*/
        DataSet ds=new DataSet();/* Tablolarimizi taşıyacak dataset nesnesini
oluşturuyoruz*/
        SqlConnection con=new SqlConnection(conString); /*SqlConnection nesnemizi
oluşturuyoruz*/
        SqlDataAdapter da;/* Bir SqlDataAdapter nesnesi belirtiyoruz ama henüz
oluşturmuyoruz*/

        /*Bu döngü gönderdiğimiz tabloadlarını alarak bir Select sorgusu oluşturur ve
SqlDataAdapter yardımıyla select sorgusu sonucu dönen tablo verilerini oluşturulan bir
DataTable nesnesine yükler. Daha sonra ise bu DataTable nesnesi DataSet nesnemizin
Tables koleksiyonuna eklenir. Bu işlem metoda gönderilen her tablo için yapılacaktır.
Böylece döngü sona erdiğinde, DataSet nesnemiz göndermiş olduğumuz tablo adlarına
sahip DataTable nesnelerini içermiş olacaktır. */

        for(int i=0;i<tabloAdi.Length;++i)
        {
            sqlSelect="SELECT * FROM "+tabloAdi[i];
            da=new SqlDataAdapter(sqlSelect,con);
            DataTable dt=new DataTable(tabloAdi[i]);

            da.Fill(dt);
            ds.Tables.Add(dt);
        }

        return ds; /* Son olarak metod çağırıldığı yere DataSet nesnesini
göndermektedir.*/
    }

    public CDataSet()
    {
    }
}

```

Şimdi ise btnYukle isimli butonumuzun kodlarını yazalım.

```

private void btnYukle_Click(object sender, System.EventArgs e)
{
    CDataSet c=new CDataSet();
    DataSet ds=new DataSet();
    ds=c.CreateDataSet("northwind","Products","Orders");

    for(int i=0;i<ds.Tables.Count;++i)
    {
        /* tabControl'umuza yeni bir tab page ekliyoruz.*/
        tabControl1.TabPages.Add(new
System.Windows.Forms.TabPage(ds.Tables[i].TableName.ToString()));
    }
}

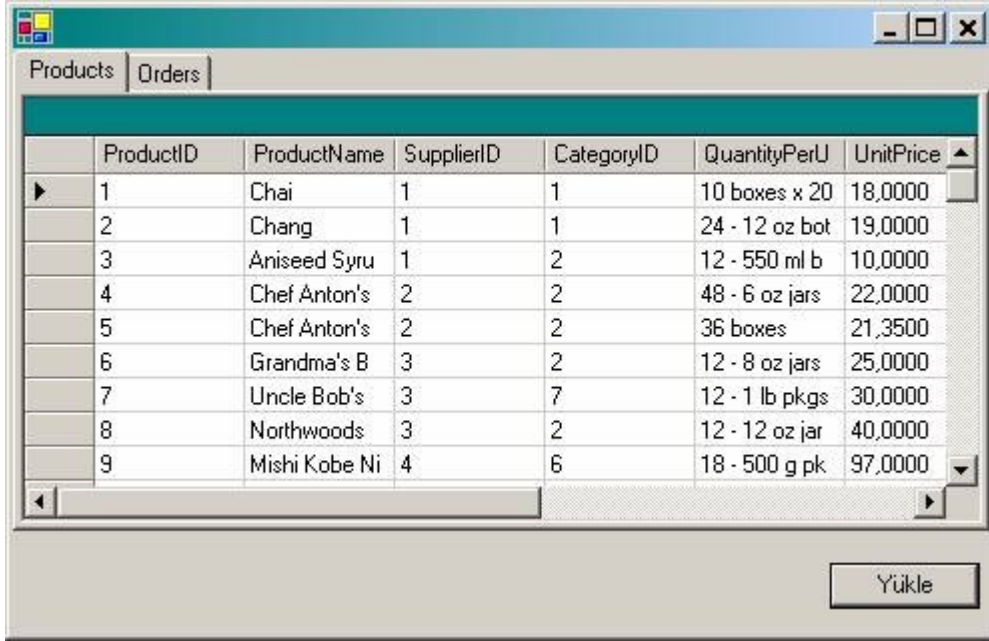
```

```

/* Oluşturulan bu tab page'e eklenmek üzere yeni bir datagrid oluşturuyoruz.*/
DataGridView dg=new DataGridView();
dg.Dock=DockStyle.Fill; /*datagrid tabpage'in tamamını kaplıyacak*/
dg.DataSource=ds.Tables[i]; /* DataSource özelliği ile DataSet te i indexli tabloyu
bağlıyoruz.*/
tabControl1.TabPages[i].Controls.Add(dg); /* Oluşturduğumuz dataGridView nesnesini tabPage
üstünde göstermek için Controls koleksiyonunun Add metodunu
kullanıyoruz.*/
}
}

```

Şimdi programımızı çalıştıralım. İşte sonuç;



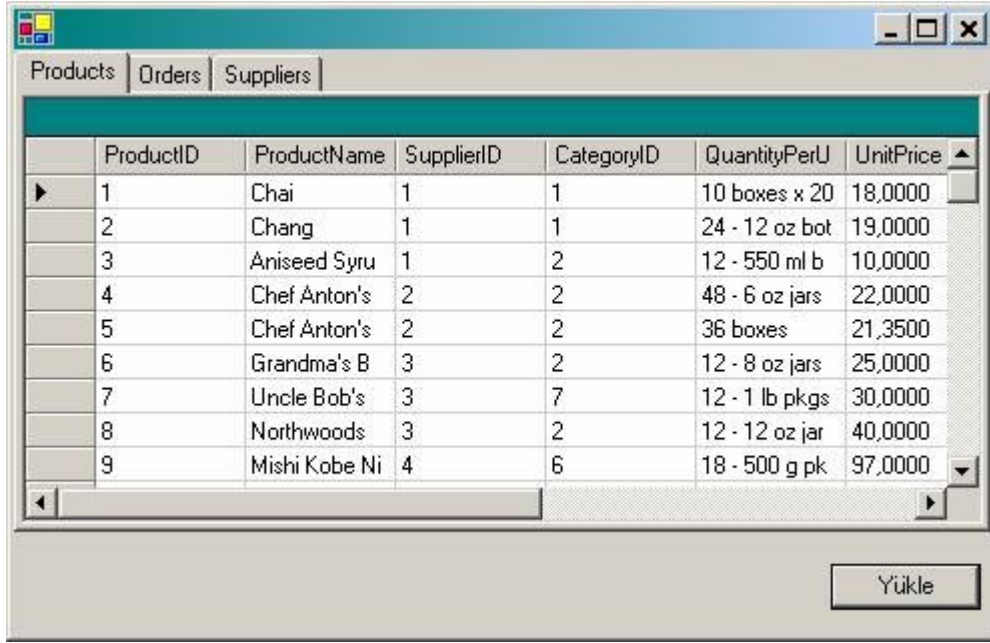
	ProductID	ProductName	SupplierID	CategoryID	QuantityPerU	UnitPrice
▶	1	Chai	1	1	10 boxes x 20	18,0000
	2	Chang	1	1	24 - 12 oz bot	19,0000
	3	Aniseed Syru	1	2	12 - 550 ml b	10,0000
	4	Chef Anton's	2	2	48 - 6 oz jars	22,0000
	5	Chef Anton's	2	2	36 boxes	21,3500
	6	Grandma's B	3	2	12 - 8 oz jars	25,0000
	7	Uncle Bob's	3	7	12 - 1 lb pkgs	30,0000
	8	Northwoods	3	2	12 - 12 oz jar	40,0000
	9	Mishi Kobe Ni	4	6	18 - 500 g pk	97,0000

Şekil 4. Tabloların yüklenmesi.

Görüldüğü gibi iki tablomuzda yüklenmiştir. Burada tablo sayısını arttırabilir veya azaltabiliriz. Bunu params anahtar kelimesi mümkün kılmaktadır. Örneğin metodumuzu bu kez 3 tablo ile çağıralım;

```
ds=c.CreateDataSet("northwind","Products","Orders","Suppliers");
```

Bu durumda ekran görüntümüz Şekil 5 teki gibi olur.



	ProductID	ProductName	SupplierID	CategoryID	QuantityPerU	UnitPrice
►	1	Chai	1	1	10 boxes x 20	18,0000
	2	Chang	1	1	24 - 12 oz bot	19,0000
	3	Aniseed Syru	1	2	12 - 550 ml b	10,0000
	4	Chef Anton's	2	2	48 - 6 oz jars	22,0000
	5	Chef Anton's	2	2	36 boxes	21,3500
	6	Grandma's B	3	2	12 - 8 oz jars	25,0000
	7	Uncle Bob's	3	7	12 - 1 lb pkgs	30,0000
	8	Northwoods	3	2	12 - 12 oz jar	40,0000
	9	Mishi Kobe Ni	4	6	18 - 500 g pk	97,0000

Yükle

Şekil 5. Bu kez 3 tablo gönderdik.

Umuyorumki params anahtar sözcüğü ile ilgili yeterince bilgi sahibi olmuşsunuzdur. Bir sonraki makalemizde görüşmek dileğiyle hepinize mutlu günler dilerim.

C#'ta Özyenilemeli Algoritmalar (Recursion)

Özyenilemeli algoritmalar tüm dünyada bilgisayar bilimleriyle ilgili bölümlerde veri yapıları ve algoritmalar derslerinde detaylı olarak incelenir. Bu bağlamda biz de makalemizde özyenilemeli algoritmaları geliştirmeyi ve C# ile kodlamayı öğreneceğiz. Önce konunun teorik temelleri üzerinde duracağız. Daha sonra daha iyi anlaşılabilmesi için konu ile ilgili örnekler yapacağız. Makaleyi bitirmeden önce ise klasik döngüler ve özyenilemeli algoritmaları karşılaştıracacağız.

Bir algoritma geliştirirken genelde döngüleri ve karar mekanizmalarını metodların içinde kullanırız. Fakat bazı durumlarda döngüler yerine özyenilemeli algoritmalar kullanmak daha kolay ve anlaşılır olabilir. Özyenilemeli (recursive) metodların püf noktası **bu tür metodların bir şekilde tekrar tekrar kendilerini çağırmasıdır**.

Özyenilemeli algoritmalarda problemin en basit hali için çözüm bulunur. Bu **en basit duruma temel durum (base case) denir**. Eğer metod temel durum için çağırılsa sonuç dönderilir. Daha karmaşık durumlar için metod, temel durumdan yararlanılarak, problemi çözmeye çalışır yani kendini çağırır. Karmaşık durumlar için yapılan her çağrı **recursion step** olarak adlandırılır.

İsterseniz konunun kafanızda daha iyi canlanması için klasik faktoriyel örneğiyle devam edelim. Sıfırdan büyük herhangi bir n tamsayısının faktoriyelinin formülü şudur:

$$n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$$

Ayrıca $0!$ ve $1!$ 'in değerleri bir olarak tanımlanmıştır. Mesela $5! = 4*3*2*1 = 120$ 'dir.

Bir sayının, mesela n , faktoriyelini özyenilemeli değilde döngü kullanarak bulmak istersek aşağıdakine benzer bir kod kullanabiliriz:

```
int faktoriyel = 1;

for( int i = n; i >= 1; i-- )
    faktoriyel *= i;
```

Kod 1: Döngü ile Faktoriyel hesabı

Eğer bu problemi özyenilemeli algoritma yardımıyla çözmek istersek şu noktaya dikkat etmemiz gerekiyor:

$$n! = n * (n-1) !$$

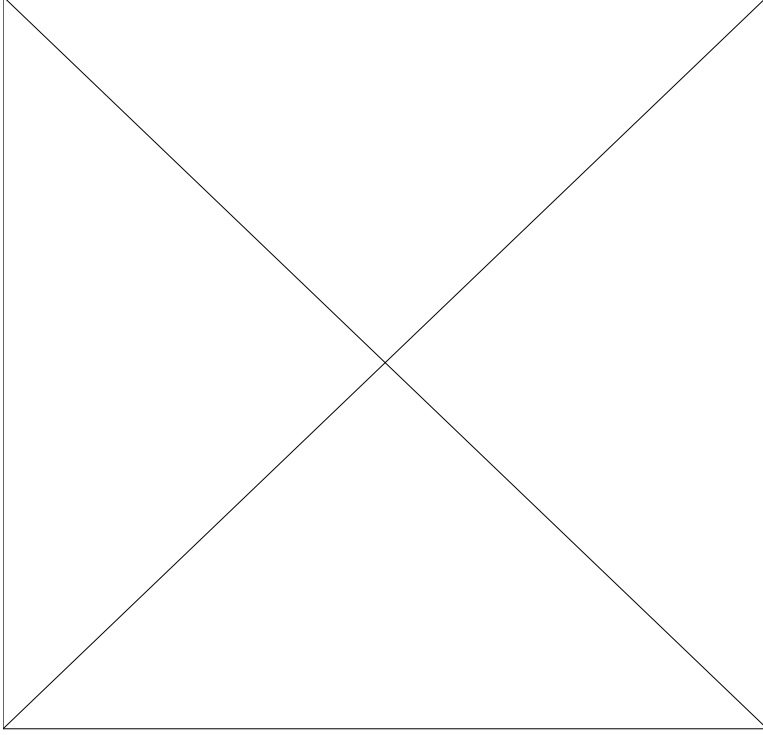
Daha açık bir yazım ile;

$$5! = 5 * 4 * 3 * 2 * 1$$

$$5! = 5 * (4 * 3 * 2 * 1)$$

$$5! = 5 * (4!)$$

Aşağıda şekilde 5!in özyenilemeli bir algorithmada nasıl hesaplanacağını görüyoruz. Şeklin solunda 5!'den 1!'le kadar her özyenilemeli çağrıda neyin çağrılacağı sağda ise sonuca ulaşılan kadar her çağrıda dönen değerler yeralıyor.



C# diliyle özyenilemeli biçimde Faktoriyel hesabı yapan bir metodu aşağıdaki gibi yazabiliriz. Bu fonksiyona int tipinde **sayi** isimli bir değişken geçiriyoruz. Eğer **sayi** 1'den küçük veya eşit ise, ki bu temel durumdur, fonksiyon 1 değerini dönderiyor. Diğer durumlarda ise fonksiyonumuz

```
sayi * Faktoriyel(sayi-1)
```

değerini dönderiyor.

```
private static long Faktoriyel(int sayi)
{
    if( sayi <= 1 ) // Eğer temel durumsa 1 dönder
        return 1;
    else
        // Temel durum değilse n * (n -1)! bul.
        return sayi * Faktoriyel( sayi-1 );
}
```

Kod 2: Özyenileme ile Faktoriyel hesabı

Sıra örneğimizi bir Windows uygulaması olacak biçimde programlayalım. Bunun için öncelikle aşağıda gördüğümüz Form'u tasarlayalım. Metin kutusuna **txtSayi** ve düğmeye **btnHesapla** isimleri vermeyi unutmayalım.



Formdaki **btnHesapla** isimli düğmeye çift tıklayalım düğmenin Click olayına cevap veren metodu yazalım.

```
private void btnHesapla_Click(object sender, System.EventArgs e)
{
    string sonucMetin="";

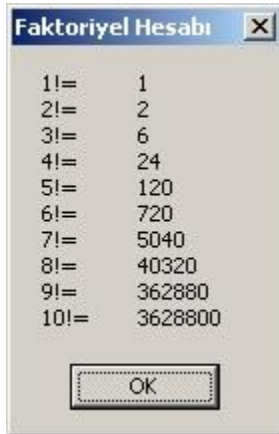
    // metin kutusundan değeri al ve int tipine çevir:
    int sayi = Convert.ToInt32(txtSayi.Text);

    for(int i=1; i<= sayi; i++)
        sonucMetin += i + "!= \t" + Faktoriyel(i).ToString() + "\n";

    MessageBox.Show(sonucMetin.ToString(),"Faktoriyel Hesabı");
}
```

Kod 3: Örnek programda `btnHesapla_Click()` metodu

Yukarıdaki metod içinde metin kutusuna girilen **sayi** değerine kadar olan tüm faktoriyeller hesaplanıp ekrana mesaj kutusu içinde yazdırılıyor. Programımızı çalıştırmadan önce programımıza Kod 2'de yeralan metodu da eklemeyi unutmayınız. Örneğimizi 10 değeri için çalıştırsak aşağıdaki sonucu elde ederiz:



Özyenilemeli Algoritma Örneği: Fibonacci Serisi

Lise ve üniversitelerde matematik derslerinde seriler konusunda gösterilen Fibonacci serilerini programlamak için döngüler yerine özyenilemeli algoritma kullanmak daha kolay ve anlaşılır oluyor. Bu serinin tanımı ise herhangi bir n sayısı için $Fibonacci(n)$ 'nin değeri $Fibonacci(n-1)$ ve $Fibonacci(n-2)$ 'nin şeklindedir. Tabiki $Fibonacci(0)$ ve $Fibonacci(1)$ 'in değeri 1 olarak alınıyor.

Bu serideki sayılar doğada çok sayıda bulunur ve spiral şeklini oluştururlar. Ayrıca art arda gelen iki Fibonacci değerinin oranı 1.68.. şeklindedir. Bu değer altın oran olarak

adlandırılır. Özellikle kart postallar, tablolar vb nesnelerin boy ve enlerinin oranları da 1.68.. gibidir. Çünkü insan gözüne çok hoş görünürler. Neyse konumuza devam edelim isterseniz...

Yukarıdaki tanımlardan yola çıkarak Fibonacci serisini hesaplayan metod'ta iki temel durum olacaktır. Bunlar Fibonacci(0) = 1 ve Fibonacci(1) = 1. Ayrıca diğer durumlar için dönen değer, herhangi bir n için, Fibonacci(n-1) ve Fibonacci(n-2)'nin toplamıdır. O zaman metodumuz aşağıdaki gibi olacaktır:

```
private static long Fibonacci(int sayi)
{
    if( sayi == 0 || sayi == 1) // Eğer temel durumlardan biriye 1 dönder
        return 1;
    else
        // Temel durum değilse (n-1) + (n -2) bul.
        return Fibonacci( sayi-1 ) + Fibonacci( sayi-2 );
}
```

Kod 4: Özyenileme ile Fibonacci serisi hesabı

Fibonacci serisi ile ilgili aşağıdaki küçük formu tasarlayalım. Sonra gerekli kodları yazıp örneğimizi deneyelim. Formdaki metin kutusunun ismi yine **txtSayi** olsun. Ayrıca Hesapla etiketine sahip düğmenin ismi **btnHesapla** olsun. Son olarak arka planı koyu kırmızı olan etiketin ismi de **label2** olacak.

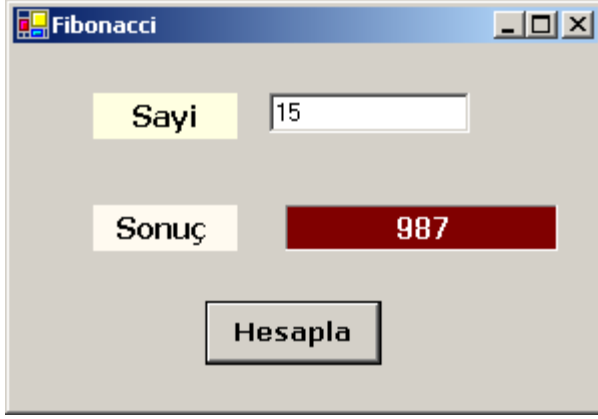
Programı tamamlamak için **Hesapla** düğmesini tıkladığında gerekli işleri yapacak kodu yazmaya geldi. Ayrıca programın kodunun içine **Kod 4** yeralan fonksiyonu da ekleyiniz.

```
private void btnHesapla_Click(object sender, System.EventArgs e)
{
    // Kullanıcını girdiği değeri al ve int tipine çevir.
    int sayi = Convert.ToInt32(txtSayi.Text);

    // Fibonacci Hesabı yapan fonksiyonu girilen sayi değeri ile çağır.
    // Sonucu label2'ye yazdır.
    label2.Text =Fibonacci(sayi).ToString();
}
```

Kod 5: Fibonacci örneğinde btnHesapla_Click() metodu

Programı test etmek için Fibonacci(15) değerini bulmak istersek aşağıdaki sonucu elde ederiz.



Özyenilemeli Algoritma Örneği: Fibonacci Serisi

Makalemizi bitirmeden önce özyenilemeli algoritmalar ile döngülerden oluşan algoritmaların aralarındaki farklara ve benzerlikte gözetmekte yarar olduğu kanısındayım. İki algoritma türü de akış kontrol mekanizmalarını kullanır. Döngülerde **for**, **while** ve **do while** yapıları kullanılırken özyenilemeli algoritmalarda **if**, **if/else** ve **switch** yapıları yer alır. Aslında hem döngüler de hem de özyenilemeli metodlarda iterasyonlar bulunur. Döngüler doğaları gereği açık bir biçimde iteratiftirler. Fakat özyenilemeli algoritmalarda iterasyon aynı metodun tekrar tekrar kendi içinden çağırılması ile gerçekleşir. Döngülü ve özyenilemeli algoritmalarda göze çarpan diğer bir benzerlikte sonlandırıcı testlerin bulunmasıdır. Döngülerde sayacın(counter) belli bir değere ulaşip ulaşmadığı kontrol edilir ve gerekirse döngü sonlanır. Özyenilemeli algoritmalarda o andaki durumun temel durum olup olmamasına göre işlemler devam edebilir veya sonlanabilir. Son olarak hem döngülerle hem de özyenilemeli algoritmalar ile bilgisayarı sonsuz döngüye(infinite loop) sokabiliriz. Birincisi döngünün sonlanacağı sayaca ulaşmanın imkansız olmasından ikincisi ise temel duruma ulaşamamaktan kaynaklanır.

Aslında özyenilemeli algoritmaları kullanırsak hem daha yavaş hem de hafızada daha çok yer kaplayan programlar yazmış oluruz. Fakat çoğu zaman aynı problemin çözümünü özyenilemeli olarak bulmak daha kolaydır. Ya da döngülerle aynı sonuca varacak algoritmayı düşünmek zor olur. Ayrıca özyenilemeli algoritmaları inceleyince anlamak ve hata ayıklamak daha kolaydır. Bu durumda seçim programcıya kalmıştır. Yalnız işlemcilerin giderek hızlanması, hafıza fiyatlarındaki düşüşü ve programın daha kolay yönetilebilmesinin getirdiği avantajları göz önüne almanızı tavsiye ederim.

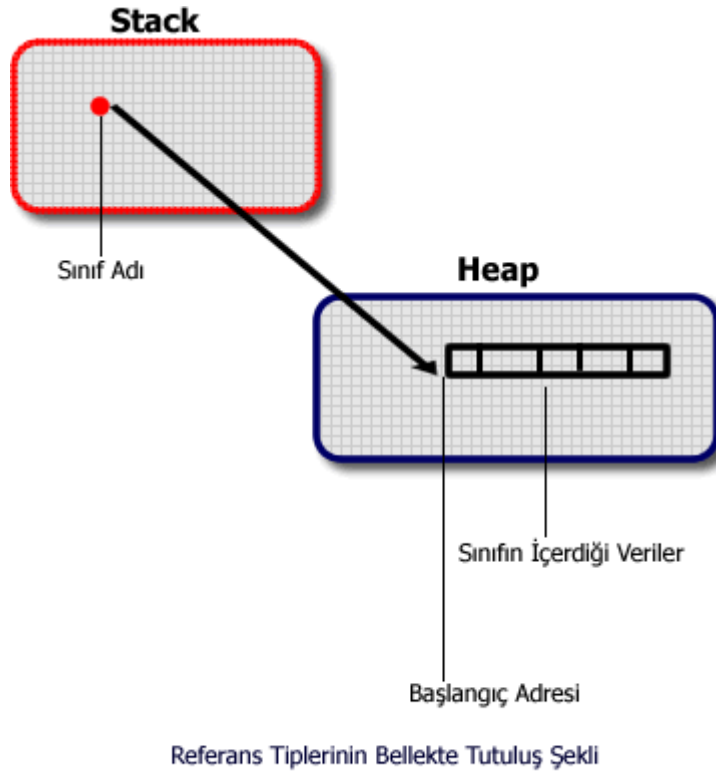
Bu makalede özyenilemeli algoritmaları detayları ile inceledik. Konu ile ilgili sorularınızı rahatlıkla sorabileceğinizi belirterek makalemize son verelim.

Struct(Yapı) Kavramı ve Class(Sınıf) ile Struct(Yapı) Arasındaki Farklar

Bu makalemizde struct kavramını incelemeye çalışacağız. Hatırlayacağınız gibi, kendi tanımladığımız veri türlerinden birisi olan Numaralandırıcıları (Enumerators) görmüştük. Benzer şekilde diğer bir veri tipide struct (yapı)lardır.

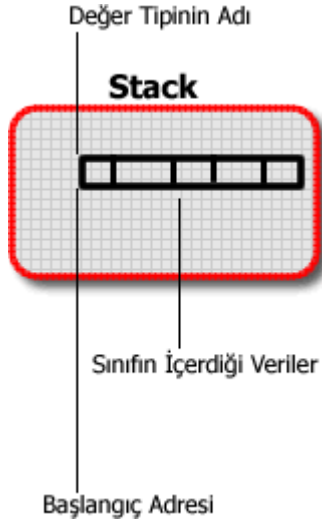
Yapılar, sınıflar ile büyük benzerlik gösterirler. Sınıf gibi tanımlanırlar. Hatta sınıflar gibi, özellikler, metodlar, veriler, yapıcılar vb... içerebilirler. Buna karşın sınıflar ile yapılar arasında çok önemli farklılıklar vardır.

Herşeyden önce en önemli fark, yapıların değer türü olması ve sınıfların referans türü olmasıdır. Sınıflar referans türünden oldukları için, bellekte tutuluş biçimleri değer türlerine göre daha farklıdır. Referans tiplerinin sahip olduğu veriler belleğin öbek(heap) adı verilen tarafında tutulurken, referansın adı stack(yığın) da tutulur ve öbekteki verilerin bulunduğu adresi işaret eder. Ancak değer türleri belleğin stack denilen kısmında tutulurlar. Aşağıdaki şekil ile konuyu daha net canlandırabiliriz.



Şekil 1. Referans Tipleri

Aşağıdaki şekilde ise değer tiplerinin bellekte nasıl tutulduğunu görüyorsunuz.



Şekil 2. Değer Tipleri

İşte sınıflar ile yapılar arasındaki en büyük fark budur. Peki bu farkın bize sağladığı getiriler nelerdir? Ne zaman yapı ne zaman sınıf kullanmalıyız? Özellikle metodlara veriler aktarırken bu verileri sınıf içerisinde tanımladığımızda, tüm veriler metoda aktarılacağını sadece bu verilerin öbekteki başlangıç adresi aktarılır ve ilgili parametrenin de bu adresteki verilere işaret etmesi sağlanmış olur. Böylece büyük boyutlu verileri stack'ta kopyalayarak gereksiz miktarda bellek harcanmasının önüne geçilmiş olunur. Ancak küçük boyutlarda veriler ile çalışırken bu verileri sınıflar içerisinde kullandığımızda bu kezde gereksiz yere bellek kullanıldığı öbek şişer ve performans düşer. Bu konudaki uzman görüş 16 byte'tan küçük veriler için yapıların kullanılması, 16 byte'tan büyük veriler için ise sınıfların kullanılmasıdır.

Diğer taraftan yapılar ile sınıflar arasında başka farklılıklarda vardır. Örneğin bir yapı için varsayılan yapıcı metod (default constructor) yazamayız. Derleyici hatası alırız. Ancak bu değişik sayıda parametreler alan yapıcılar yazmamızı engellemez. Oysaki sınıflarda istersek sınıfın varsayılan yapıcı metodunu kendimiz yazabilmekteyiz.

Bir yapı içerisinde yer alan constructor metod(lar) içinde tanımlamış olduğumuz alanlara başlangıç değerlerini atamak zorundayız. Oysaki bir sınıftaki constructor(lar) içinde kullanılan alanlara başlangıç değerlerini atamaz isek, derleyici bizim yerimize sayısal değerlere 0, boolean değerlere false vb... gibi başlangıç değerlerini kendisi otomatik olarak yapar. Ancak derleyici aynı işi yapılar da yapmaz. Bu nedenle bir yapı içinde kullandığımız constructor(lar)daki tanımlamış olduğumuz alanlara mutlaka ilk değerlerini vermemiz gerekir. Ancak yine de dikkat edilmesi gereken bir nokta vardır. Eğer yapı örneğini varsayılan yapılandırıcı ile oluşturursak bu durumda derleyici yapı içinde kullanılan alanlara ilk değerleri atanmamış ise kendisi ilk değerleri atar. Unutmayın, parametrelili constructorlarda her bir alan için başlangıç değerlerini bizim vermemiz gerekmektedir. Örneğin, aşağıdaki Console uygulamasını inceleyelim.

```
using System;

namespace StructSample1
{
    struct Zaman
    {
        private int saat,dakika,saniye;
    }
}
```

```
private string kosucuAdi;

public string Kosucu
{
    get
    {
        return kosucuAdi;
    }
    set
    {
        kosucuAdi =value;
    }
}

public int Saat
{
    get
    {
        return saat;
    }
    set
    {
        saat =value;
    }
}

public int Dakika
{
    get
    {
        return dakika;
    }
    set
    {
        dakika =value;
    }
}

public int Saniye
{
    get
    {
        return saniye;
    }
    set
    {
        saniye =value;
    }
}
}

class Class1
{
    static void Main (string[] args)
    {
        Zaman z;
        Console.WriteLine ("Koşucu:"+z.Kosucu);
        Console.WriteLine ("Saat:"+z.Saat.ToString());
    }
}
```

```
        Console.WriteLine ("Dakika:"+z.Dakika.ToString());  
        Console.WriteLine ("Saniye:"+z.Saniye.ToString());  
    }  
}
```

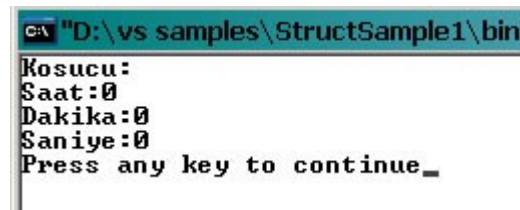
Yukarıdaki kod derlenmeyecektir. Nitekim derleyici **"Use of unassigned local variable 'z'"** hatası ile z yapısı için ilk değerlerin atanmadığını bize söyleyecektir. Ancak z isimli Zaman yapısı türünü new anahtarı ile tanımlarsak durum değişir.

Zaman z;

Satırı yerine

Zaman z=new Zaman ();

yazalım . Bu durumda kod derlenir. Uygulama çalıştığında aşağıdaki ekran görüntüsü ile karşılaşırız. Görüldüğü gibi z isimli yapı örneğini new yapılandırıcısı ile tanımladığımızda, derleyici bu yapı içindeki özelliklere ilk değerleri kendi atamıştır. Kosucu isimli özellik için null, diğer integer özellikler için ise 0.



```
C:\> "D:\vs samples\StructSample1\bin  
Kosucu:  
Saat:0  
Dakika:0  
Saniye:0  
Press any key to continue_
```

Şekil 3. New yapılandırıcısı ile ilk değer ataması.

Yine önemli bir farkta yapılarda türetme yapamayacağımızdır. Bilindiği gibi bir sınıf oluşturduğumuzda bunu başka bir temel sınıftan kalıtım yolu ile türetebilmekteyiz ki inheritance olarak geçen bu kavramı ilerleyen makalelerimizde işleyeceğiz. Ancak bir yapıyı başka bir yapıyı temel alarak türetmeyiz. Şimdi yukarıda verdiğimiz örnekteki yapıdan başka bir yapı türetmeye çalışalım.

struct yeni:Zaman

```
{  
  
}
```

satırlarını kodumuza ekleyelim. Bu durumda uygulamayı derlemeye çalıştığımızda aşağıdaki hata mesajını alırız.

'Zaman' : type in interface list is not an interface

Bu belirgin farklılıklarıda inceledikten sonra dilerseniz örneklerimiz ile konuyu pekiştirmeye çalışalım.

```
using System;

namespace StructSample1
{
    struct Zaman
    {
        private int saat,dakika,saniye;
        private string kosucuAdi;

        /* Yapı için parametrelili bir constructor metod tanımladık. Yapı içinde yer alan kosucuAdi,saat,dakika,saniye alanlarına ilk değerlerin atandığına dikkat edelim. Bunları atamassak derleyici hatası alırız. */

        public Zaman(string k,int s,int d,int sn)
        {
            kosucuAdi =k;
            saat =s;
            dakika =d;
            saniye =sn;
        }

        /* Bir dizi özellik tanımlayarak private olarak tanımladığımız asıl alanların kullanımını kolaylaştırıyoruz. */
        public string Kosucu
        {
            get
            {
                return kosucuAdi;
            }
            set
            {
                kosucuAdi =value;
            }
        }

        public string Saat
        {
            get
            {
                return saat;
            }
            set
            {
                saat =value;
            }
        }

        public string Dakika
        {
            get
            {
                return dakika;
            }
            set
            {
                dakika =value;
            }
        }

        public string Saniye
        {
            get
            {
```

```

        return saniye;
    }
    set
    {
        saniye =value;
    }
}

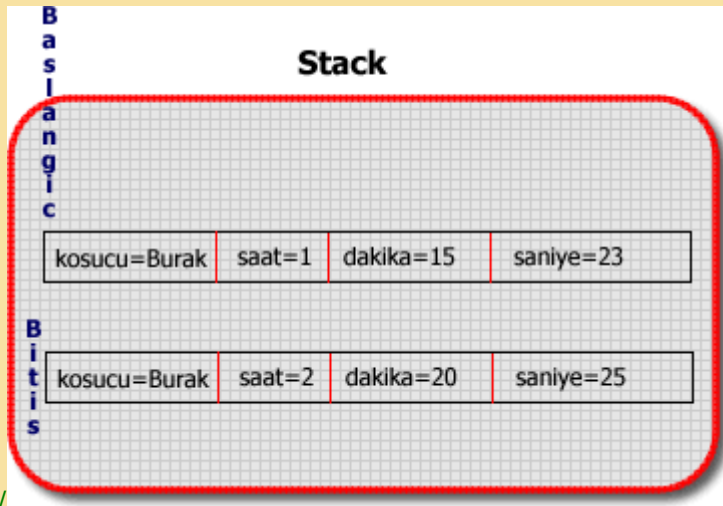
```

```

class Class1
{
    static void Main (string[] args)
    {

```

/* Zaman yapısı içinde kendi yazdığımız parametrelili constructorlar ile Zaman yapısı örnekleri oluşturuyoruz. Yaptığımız bu tanımlamaların ardından belleğin stack bölgesinde derhal 4 adet değişken oluşturulur ve değerleri atanır. Yani kosucuAdi,saat,dakika,saniye isimli private olarak tanımladığımız alanlar bellekte stack bölgesinde oluşturulur ve atadığımız değerleri alırlar. Bu oluşan veri dizisinin adıda Zaman yapısı tipinde olan Baslangic ve Bitis değişkenleridir. */



```

//

```

```

Zaman Baslangic=new Zaman ("Burak",1,15,23);

```

```

Zaman Bitis=new Zaman ("Burak",2,20,25);

```

/* Zaman yapısı içinde tanımladığımız özelliklere erişip işlem yapıyoruz. Burada elbette zamanları birbirinden bu şekilde çıkarmak matematiksel olarak bir cinayet. Ancak amacımız yapıların kullanımını anlamak. Bu satırlarda yapı içindeki özelliklerimizin değerlerine erişiyor ve bunların değerleri ile sembolik işlemler yapıyoruz */

```

int saatFark=Bitis.Saat-Baslangic.Saat;

```

```

int dakikaFark=Bitis.Dakika-Baslangic.Dakika;

```

```

int saniyeFark=Bitis.Saniye-Baslangic.Saniye;

```

```

Console.WriteLine ("Fark {0} saat, {1} dakika, {2} saniye",saatFark,dakikaFark,saniyeFark);

```

```

    }
}

```

Bir sonraki makalemizde görüşmek dileğiyle. Hepinize mutlu günler dilerim

Windows 98'de C# Kodu Derleyin

Windows 9x işletim sistemlerine dotNET Framework kurularak, dotNET platformunda yazılmış programlar çalıştırılabilir, fakat siz 9x işletim sistemlerinde dotNET programı yazıp derleyemiyorsunuz, çünkü .NET programlarını yazabilmek için Microsoft 2 araç sunuyor.

1) .NET Framework SDK: Yükleyebilmek için en az windows NT 4.0 SP6 kurulu 32 MB RAM'li bir sistem gerekmektedir.

2) Visual Studio .NET: Yükleyebilmek için en az Windows 2000 Professional SP3 ve 96 MB RAM içeren bir sistem gerekmektedir.

Diyelim ki elimizde bir Windows 98 işletim sistemi yüklü sistem var. Bu sistemin belleği de sadece 32 MB olsun. Bu özelliklere sahip bir sistem üzerinde .NET programları yazmak isteyelim. Bunu yapabilir miyiz?

Eğer .NET programlarını yazmanın tek yolu yukarıdaki araçları kullanmak olsaydı bu sorunun yanıtı "hayır" olacaktı. Fakat .NET programlarını yazmanın birkaç yolu daha var. Bunlardan bir tanesi Mono projesi dahilinde geliştirilen C# derleyicisidir. <http://www.gomono.com/> adresinden projeyle ilgili bilgilere ve gerekli tüm programlara ücretsiz ulaşabilir bilgisayarınıza indirebilirsiniz. mono'nun bugünkü tarih itibarıyla Windows için 0.28 sürümü mevcut. Mono'nun CLR altyapısı ile basit bir uygulamayı "**mint den1.exe**" şeklinde çalıştırmak istediğimde benim bilgisayarımda "bellek yetersiz" gibi bir hata verdi. Bu yüzden Microsoft .NET Framework kurmanızı da tavsiye ederim. Microsoft'un sitesinden son .net framework kurulum dosyasını indirip kurduktan sonra Mono'nun "mcs" derleyicisi ile derlediğiniz programları normal Windows uygulaması çalıştırıyormuş gibi çift tıklayarak çalıştırabilirsiniz. Aşağıda basit bir örnek görülüyor.

```
class den1{
public static void Main(){
System.Console.WriteLine("denemedir.");
System.Console.ReadLine();
}
}
```

Bu örnek uygulamayı mcs derleyicisi ile aşağıdaki gibi derliyoruz.

```
D:\Program Files\Mono-0.28\bin\>mcs den1.cs
Compilation succeeded
```

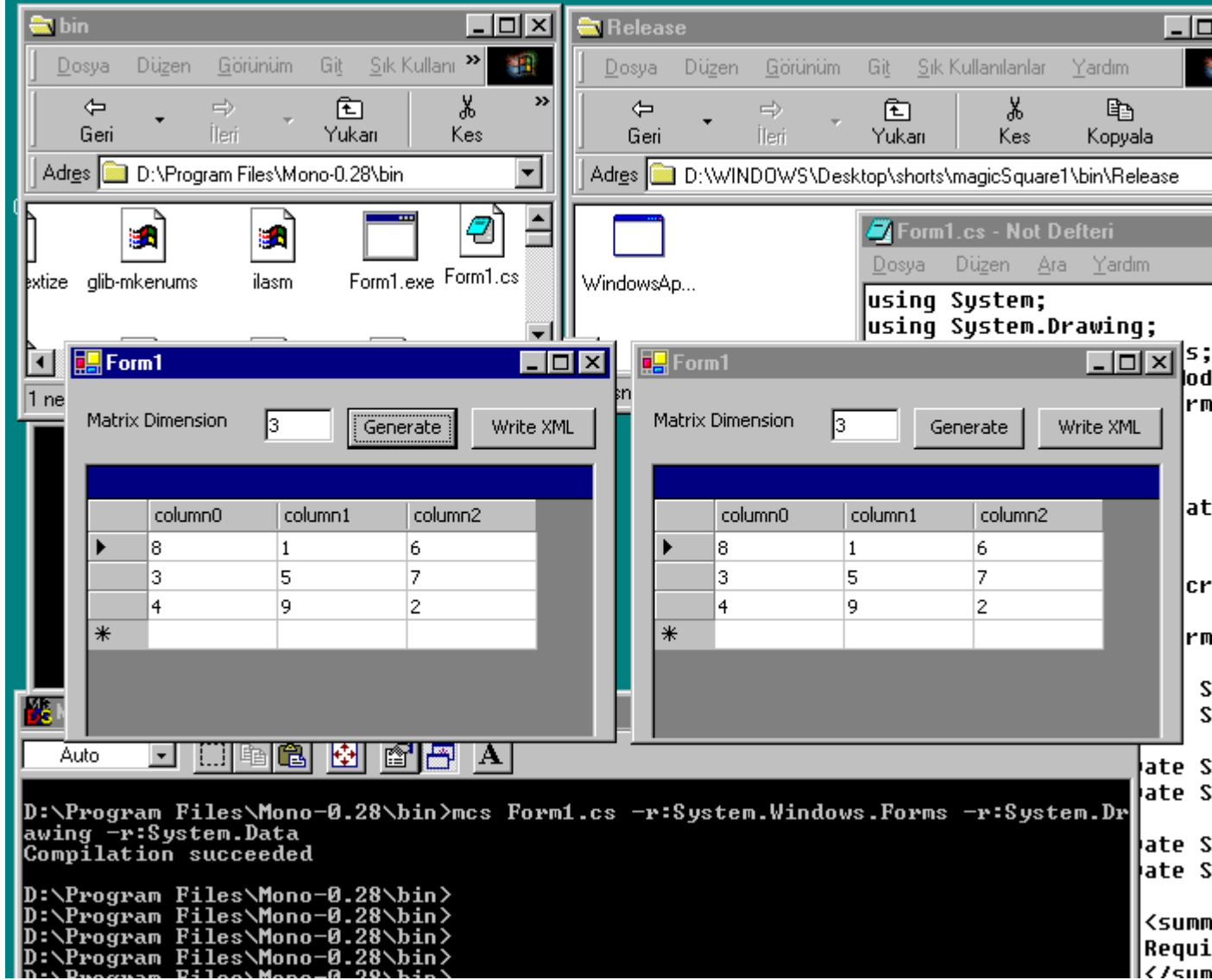
Oluşan dosyaya Çift tıklayarak ya da exe'nin adını yazarak uygulamayı çalıştırıyoruz.

```
D:\Program Files\Mono-0.28\bin\>den1.exe
Denemedir.
```

İsterseniz daha karmaşık bir uygulama ile mcs derleyicisinin yeteneklerini test edelim.

Matematik'te satır, sütun veya diyagonallerindeki sayıların toplamının hep aynı sayıya eşit olduğu karelere "sihirli kare" denir. Verilen bir tek sayılı boyut için sihirli kare oluşturan algoritma uygulaması C# ile verilmiştir. Program Visual Studio .NET 2003 ortamında

yazılmıştır ve derlenmiştir. VS.NET ile oluşturulan çalıştırılabilir dosyanın adı WindowsApplication6.exe'dir. Aynı kaynak kod (Form1.cs) hiçbir değişikliğe uğratılmadan Windows 98 üzerine kurulu Mono-0.28 ve .NET Framework 1.1 yüklü makinede Mono derleyicisiyle "**mcs Form1.cs -r:System.Windows.Forms -r:System.Drawing -r:System.Data**" komutuyla derlenmiştir. Derleme başarıyla sonuçlanmış ve Form1.exe adlı dosya oluşmuştur. Aşağıdaki masaüstü görüntüsünde sol tarafta çalıştırılan uygulama Mono ile derlenen, sağ tarafta çalıştırılan uygulama ise (aynı kaynak koddan derlenmiştir) VS.NET 2003'te derlenmiştir. İki dosya da çift tıklanarak çalıştırılmıştır.



Şekil 1: Mono ve VS.NET ile derlenen uygulamalar.

[VS.NET Uygulamasını indirmek için tıklayın.](#)

[Mono ile derlenen uygulamayı indirmek için tıklayın.](#)

Sihirli Karelerin Oluşturulması - Basamak Yöntemi

"Sihirli Kare" oluşturmak için kullanılan yöntem La Loubere'in bulduğu "Basamak" adı verilen yöntemdir.

Aşağıda genel kuralı verilen "Basamak" yöntemi her tek boyutlu sihirli kareyi oluşturabilir. Aşağıdaki anlatım "Yaşayan Matematik" adlı kitabın 53. sayfasından alınmıştır, bu konu

hakkında daha detaylı bilgi ve buna benzer keyifli matematik eğlencelerini öğrenmek için bu kitaba başvurmanız tavsiye edilir. Bu yöntemin 3x3'lük bir sihirli kareye uygulandığı aşağıdaki şekilde gösterilmiştir.

La Loubere'un Basamak Yöntemi														
1.adım			2.adım			3.adım			4.adım			5.adım		
1			1			1			1			1		
						3			3			3	5	
				2			2		4		2	4		2
6.adım			7.adım			8.adım			9.adım					
1	6		1	6		8	1	6	8	1	6			
3	5		3	5	7	3	5	7	3	5	7			
4		2	4		2	4		2	4	9	2			

Şekil 2: Sihirli kare algoritması

"

- 1) 1 sayısını en üst satırın ortasındaki kareye yerleştirerek başlayalım.
- 2) Her koyduğumuz sayının sağ üst çaprazına bir sonraki sayıyı koyalım. Eğer burası sihirli karenin dışındaki hayali bir kareyse (a,b,...,g diye isimlendirdiklerimizden biriyse) sihirli karede bu yere denk gelen kutuyu bulup buraya sayımızı yerleştirelim.
- 3) Eğer sihirli karedeki sağ üst çapraz doluysa, o zaman sayıyı bir önceki sayının altındaki kutuya yerleştirelim (4 ve 7 sayılarında olduğu gibi).
- 4) 2. ve 3. basamakları uygulamayı sürdürerek sihirli karedeki diğer sayıların yerlerini bulalım."

["Yaşayan Matematik",s.53]

Tavsiyeler

[Mono'nun resmi sayfası](#)
[Mono semineri slaytları](#)

Referans

"Yaşayan Matematik", Theoni PAPPAS, Türkçe'ye çeviren: Yıldız SİLİER, Sarmal Yayımevi, Ekim 1993.

Windows XP Stillerinin Kontrollere Uyarlanması

Merhaba, bu makalede Windows XP stillerinin form kontrollerine herhangi bir Manifest dosyası olmadan uygulanmasını tartışacağız. Ben uygulama örneği olarak bir ProgressBar kontrolünü seçtim. Çalıştırılabilir örnek kod bu siteden indirilebilir.

Herşeyden önce bu isteğimizi gerçekleştirmek için isteğimizi uygulayacağımız bir sınıfa sahip olmamız gerekir. Ne yazık ki standart ProgressBar sınıfı "sealed" olduğu için ben sınıfımı Control sınıfından türettim:

```
public class XPProgressBar:Control
{
}
```

Daha sonra yapmamız gereken standart ProgressBar özelliklerini (Maximum, Minimum, Value, Step) özelliklerini sınıfımıza uygulamaktır. Bu özelliklerin hepsi integer'dir. Kolay olduğu için bu kısmı geçiyorum. Dikkat edilmesi gereken nokta Value özelliğine bir değer geçerken sınıf örneğimizin yeniden boyanmasını sağlamaktır. Bunu da Refresh() fonksiyonu ile yapabiliriz. Uygulamamızın amacı XP stillerinin uygulanması olduğu için Style isminde bir özellik tanıtırız. Bu özellik Normal ve System isimlerinde iki enum değeri tutar. Eğer özelliğimizin değeri System ise XP stili uygulanır.

```
public Styles Style
{
    get {<return FStyle;}
    set
    {
        FStyle=value;
        Refresh();
    }
}
```

Şimdi sınıfımızın boyanması için OnPaint metoduna geçebiliriz. İlkönce Normal değerini görelim.

```
protected override void OnPaint(PaintEventArgs e)
{
    Rectangle rect = ClientRectangle;

    rect.Width-=6;

    rect.Height-=6;

    if(Maximum!=Minimum)
    {
        rect.Width =(Value-Minimum)*100/(Maximum-Minimum)*rect.Width/100;
    }
}
```

```

if(Style==Styles.Normal)
{
    ControlPaint.DrawBorder3D(e.Graphics,ClientRectangle,Border3DStyle.Sunken);

    rect=new Rectangle(new Point(rect.X+3,rect.Y+3),rect.Size);

    e.Graphics.FillRectangle(Brushes.Blue,rect);
}
else
{
    //Burada Styles.System değerini uygulayacağız.
}
}

```

Yukarıdaki kodda yapılan Maximum ve Minimum değerlerine göre boyanacak alanın bulunup daha sonra boyanmasıdır. Dikkat edeceğimiz gibi segment şeklinde değil de düz boya şeklinde boyanmıştır. Eğer isterseniz bu aşamada bileşenimizi test edebilirsiniz.

Dikkat edeceğimiz gibi standart bir ProgressBar'ın yapacağı işlemleri gerçekleştirdik. Şimdi asıl amacımız XP sitillerinin uygulanması olduğuna göre bu aşamaya geçelim.

Öncelikle kullandığımız işletim sisteminin yapacağımız bu işlemi desteklemesi için Windows XP olması lazım. Bunu nasıl anlayacağımızı görelim:

```

bool IsThemedos()
{
    if(Environment.OSVersion.Platform != PlatformID.Win32NT

        || Environment.OSVersion.Version.Major < 5

            || Environment.OSVersion.Version.Minor < 1)

        return false;

    return true;
}

```

Yukarıdaki fonksiyonda yapılan kullanılan işletim sisteminin version numaralarına bakarak Windows XP olup olmadığını anlamaktır. Eğer değilse program XP sitillerinin kullanımına izin vermeyecektir.

Şimdi burada bir ara verip XP sitillerinin kullanımına olanak veren WinApi leri tanıtmak istiyorum. Öncelikle kullanacağımız dll "UxTheme.dll" dir. Bizim bu programda kullanacağımız 4 tane WinApi fonksiyonu var. Şimdi bunları tanıyalım:

HTHEME OpenThemeData(HWND *hwnd*,LPCWSTR *pszClassList*);

Bu fonksiyon bir window için ilgili sınıfın datasını açar. Dönüş değeri IntPtr dir. "pszClassList" parametresi ise kullanacağımız sınıfın string değeridir. Bizim örneğimizde bu değer "PROGRESS" dir.

BOOL IsAppThemed(VOID);

Bu Api bir kontrol fonksiyonudur. uygulamamızın visual sitilleri uygulayıp uygulamayacağını sorgular.

BOOL IsThemeActive(VOID);

Bu da başka bir kontrol fonksiyonudur. Visual sitillerin uygulamamız için aktif olup olmadığını denetler.

HRESULT DrawThemeBackground(

```
HTHEME hTheme,  
HDC hdc,  
int iPartId,  
int iStateId,  
const RECT *pRect,  
const RECT *pClipRect  
);
```

Uygulamamızın belki de en can alıcı Api si budur. Bu fonksiyonla visual sitillerin çizim işlemini gerçekleştiriyoruz. Dönüş değeri integer dir. Parametrelerine gelince:

hTheme: OpenThemeData fonksiyonu ile elde ettiğimiz IntPtr değerini kullanacağız.

hdc: Controlümüzün hdc değeri. CreateGraphics().GetHdc() ile elde edilir.

iPartId ve iStateId: Bu parametreler çizeceğimiz kontrolün bölümlerini ifade eder. Visual Studio nun yardım indeksi bölümüne "Parts and States" yazarsak konuyla ilgili dökümanı bulabiliriz. Bu durumda Controlümüzün arkaplanını çizeceksek iPartId 1 değerini, eğer öndeki ilerleme bölümünü çizeceksek 3 değerini almalıdır. StateId değeri ise ProgressBar için kullanılmaz. Eğer dökümanı iyice incelersek yapacağımız diğer uygulamalarda XP sitillerini nasıl uygulayacağımız kolaylıkla anlaşılır.

pRect ve pClipRect: Bu parametreler çizeceğimiz dörtgen bölümü ifade eder. ClipRect "null" değerini alabilir.

Şimdi programımıza geri dönüp anlattığımız Api lerin kullanımına geçebiliriz. Öncelikle sınıfımızın yapıcısında OpenThemeData apisini kullanarak XP sitilimizin değerini tutabiliriz. Daha sonra sınıfımızın OnPaint metodunda boş bıraktığımız "else" ifadesini dolduralım.

```
if(IsThemedos() && IsAppThemed() && IsThemeActive())  
{  
    DrawThemeBackground(e.Graphics,1,1,ClientRectangle);  
    DrawSystemSegments(e.Graphics,rect);  
}  
  
private void DrawSystemSegments(Graphics g, Rectangle rc)  
{  
    int segwidth = 8;  
  
    int Count = ((rc.Width)/(segwidth+2))+((rc.Width<=0)?0:1);  
  
    Rectangle rect = new Rectangle(3,3,Count*(segwidth+2),rc.Height);  
  
    if (rect.Width>(Width-2*3))  
    {  
  
        rect.Width = rc.Width;
```

```
}  
  
    DrawThemeBackground(g,3,1,rect);  
}
```

Öncelikle kontrolümüzün arka planını çizdik. Daha sonra DrawSystemSegments fonksiyonu ile segmentleri hesaplayıp sonra da bunu çizdirdik.

Bu konu hakkında anlatacaklarım bitti. Eğer örnek kod incelenirse anlaşılmayan bölümlerin anlaşılmasına yardımcı olacaktır. Hepinize iyi çalışmalar.

Not : XP stilindeki kontrolleri görmeniz için işletim sisteminizin temasını XP Stil olarak değiştirmeniz gerekmektedir.

Hashtable Koleksiyon Sınıfının Kullanımı

Bu makalemizde Hashtable koleksiyon sınıfını incelemeye çalışacağız. Bildiğiniz gibi Koleksiyonlar System.Collections namespace'inde yer almakta olup, birbirlerinin aynı veya birbirlerinden farklı veri tiplerinin bir arada tutulmasını sağlayan diziler oluşturmamıza imkan sağlamaktadırlar. Pek çok koleksiyon sınıfı vardır. Bugün bu koleksiyon sınıflarından birisi olan Hashtable koleksiyon sınıfını inceleyeceğiz.

Hashtable koleksiyon sınıfında veriler key-value dediğimiz anahtar-değer çiftleri şeklinde tutulmaktadırlar. Tüm koleksiyon sınıflarının ortak özelliği barındırdıkları verileri object tipinde olmalarıdır. Bu nedenle, Hashtable'lardada key ve value değerleri herhangi bir veri tipinde olabilirler. Temel olarak bunların her biri birer DictionaryEntry nesnesidir. Bahsetmiş olduğumuz key-value çiftleri hash tablosu adı verilen bir tabloda saklanırlar. Bu değer çiftlerine erişmek için kullanılan bir takım karmaşık kodlar vardır.

Key değerleri tektir ve değiştirilemezler. Yani bir key-value çiftini koleksiyonumuza eklediğimizde, bu değer çiftinin value değerini değiştirebilirken, key değerini değiştiremeyiz. Ayrıca key değerleri benzersiz olduklarında tam anlamıyla birer anahtar alan vazifesi görürler. Diğer yandan value değerline null değerler atayabilirken, anahtar alan niteliğindeki Key değerlerine null değerler atayamayız. Şayet uygulamamızda varolan bir Key değerini eklemek istersek ArgumentException istisnası ile karşılaşırız.

Hashtable koleksiyonu verilere hızlı bir biçimde ulaşmamızı sağlayan bir kodlama yapısına sahiptir. Bu nedenle özellikle arama maliyetlerini düşürdüğü için tercih edilmektedir. Şimdi konuyu daha iyi pekiştirebilmek amacıyla, hemen basit bir uygulama geliştirelim. Uygulamamızda, bir Hashtable koleksiyonuna key-value çiftleri ekliyecek, belirtilen key'in sahip olduğu değere bakılacak, tüm Hashtable'ın içerdiği key-value çiftleri listelenecek, eleman çiftlerini Hashtable'dan çıkartacak vb... işlemler gerçekleştireceğiz. Form tasarımını ben aşağıdaki şekildeki gibi yaptım. Temel olarak teknik terimlerin türkçe karşılığına dair minik bir sözüğü bir Hashtable olarak tasarlayacağız.

Şekil 1. Form Tasarımımız.

Şimdi kodlarımıza bir göz atalım.

```
System.Collections.Hashtable htTeknikSozluk; /* HashTable koleksiyon nesnemizi
tanımlıyoruz.*/

private void Form1_Load(object sender, System.EventArgs e)
{
    htTeknikSozluk=new System.Collections.Hashtable(); /* HashTable nesnemizi
    oluşturuyoruz.*/
    stbDurum.Text=htTeknikSozluk.Count.ToString(); /* HashTable'imizdaki eleman
    sayisini Count özelliği ile öğreniyoruz.*/
}

private void btnEkle_Click(object sender, System.EventArgs e)
{
    try
    {
        htTeknikSozluk.Add(txtKey.Text,txtValue.Text);/* HashTable'imiza key-value çifti
        ekleyebilmek için Add metodu kullanılıyor.*/
        lstAnahtar.Items.Add(txtKey.Text);
        stbDurum.Text=htTeknikSozluk.Count.ToString();
    }
    catch(System.ArgumentException) /* Eger var olan bir key'i tekrar eklemeye çalışırsak
    bu durumda ArgumentException istisnası fırlatılacaktır. Bu durumda, belirtilen key-
    value çifti HashTable koleksiyonuna eklenmez. Bu durumu kullanıcıya bildiriyoruz.*/
    {
        stbDurum.Text=txtKey.Text+" Zaten HashTable Koleksiyonunda Mevcut!";
    }
}

private void lstAnahtar_DoubleClick(object sender, System.EventArgs e)
{
    string deger;
```



```

        deger=htTeknikSozluk[lstAnahtar.SelectedItem.ToString()].ToString(); /*
        HashTable'daki bir degere ulasmak için, köseli parantezler arasında aranacak key degerini
        giriyoruz. Sonucu bir string degiskenine aktariyoruz.*/

        MessageBox.Show(deger,lstAnahtar.SelectedItem.ToString());
    }

    private void btnSil_Click(object sender, System.EventArgs e)
    {
        if(htTeknikSozluk.Count==0)
        {
            stbDurum.Text="Çıkartilabilecek hiç bir eleman yok";
        }
        else if(lstAnahtar.SelectedIndex== -1)
        {
            stbDurum.Text="Listeden bir eleman seçmelisiniz";
        }
        else
        {
            htTeknikSozluk.Remove(lstAnahtar.SelectedItem.ToString()); /* Bir HashTable'dan
            bir nesneyi çıkartmak için, Remove metodu kullanilir. Bu metod parametre olarak
            çıkartilmek istenen deger çiftinin key degerini alır.*/

            lstAnahtar.Items.Remove(lstAnahtar.SelectedItem);

            stbDurum.Text="Çıkartildi";

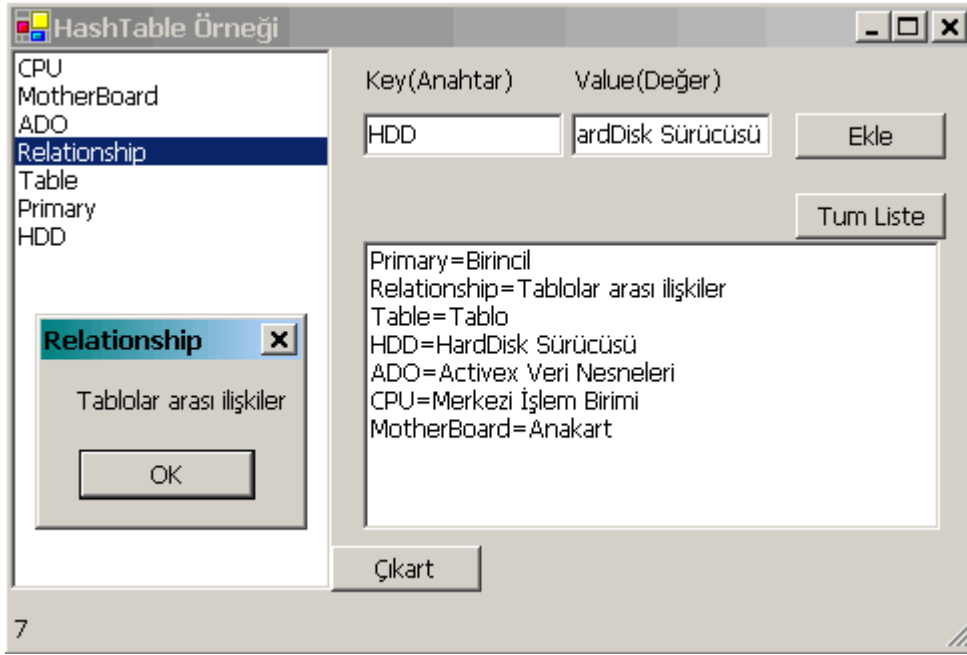
            stbDurum.Text=htTeknikSozluk.Count.ToString();
        }
    }

    private void btnTumu_Click(object sender, System.EventArgs e)
    {
        lstTumListe.Items.Clear(); /* Asagidaki satirlarda, bir HashTable koleksiyonu içinde yer
        alan tüm elemanlara nasıl erişildiğini görmekteyiz. Keys metodu ile HashTable
        koleksiyonumuzda yer alan tüm anahtar degerlerini (key'leri), ICollection
        arayüzü(interface) türünden bir nesneye atıyoruz. Foreach döngümüz ile bu nesne içindeki
        her bir anahtari, HashTable koleksiyonunda bulabiliyoruz.*/

        ICollection anahtar=htTeknikSozluk.Keys; foreach(string a in anahtar)
        {
            lstTumListe.Items.Add(a+"="+htTeknikSozluk[a].ToString());
        }
    }
}

```

Şimdi uygulamamızı çalıştıralım.



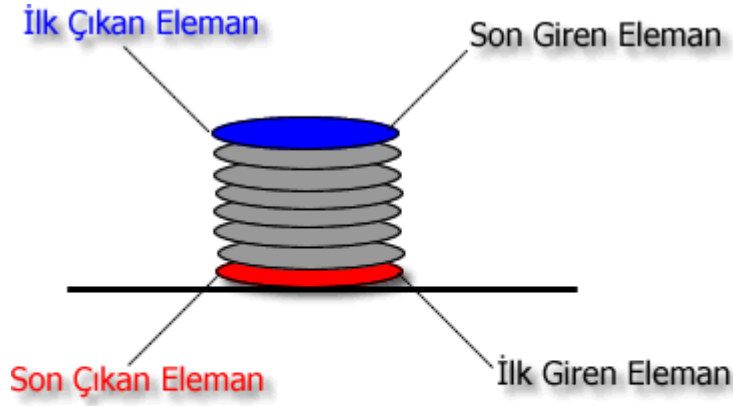
Şekil 2. Programın Çalışmasının sonucu.

Geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşmek dieğiyle hepinize mutlu günler dilerim.

Stack ve Queue Koleksiyon Sınıfları

Bu makalemizde Stack ve Queue koleksiyon sınıflarını incelemeye çalışacağız. Bir önceki makalemizde bildiğiniz gibi, HashTable koleksiyon sınıfını incelemiştik. Stack ve Queue koleksiyonlarında, System.Collections isim alanında yer alan ve ortak koleksiyon özelliklerine sahip sınıflardır.

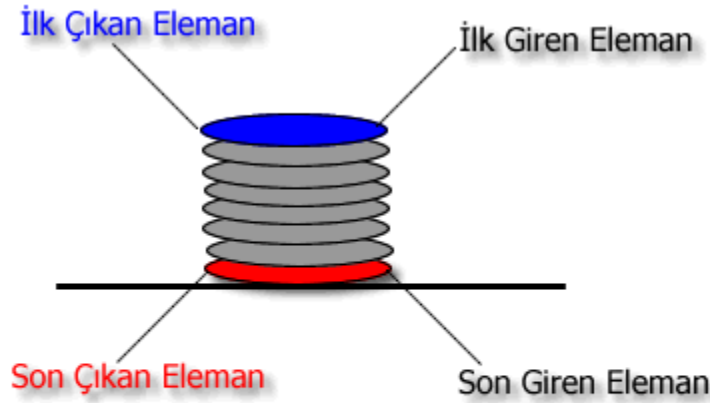
Stack ve Queue koleksiyonları, her koleksiyon sınıfında olduğu gibi, elemanlarını object tipinde tutmaktadırlar. Bu koleksiyonların özelliği giren-çıkan eleman prensipleri üzerine çalışmalarıdır. Stack koleksiyon sınıfı, LIFO adı verilen, Last In First Out(Son giren ilk çıkar) prensibine göre çalışırken, Queue koleksiyon sınıfı FIFO yani First In First Out(ilk giren ilk çıkar) prensibine göre çalışır. Konuyu daha iyi anlayabilmek için aşağıdaki şekilleri göz önüne alalım.



LIFO (Last In First Out-Son Giren İlk Çıkar)
Stack Koleksiyon Sınıfı

Sekil 1. Stack Koleksiyon Sınıfının Çalışma Yapısı

Görüldüğü gibi, Stack koleksiyonunda yer alan elemanlardan son girene ulaşmak oldukça kolaydır. Oysaki ilk girdiğimiz elemana ulaşmak için, bu elemanın üstünde yer alan diğer tüm elemanları silmemiz gerekmektedir. Queue koleksiyon sınıfına gelince;



FIFO (First In First Out-İlk Giren İlk Çıkar)
Queue Koleksiyon Sınıfı

Sekil 2. Queue Koleksiyon Sınıfının Çalışma Yapısı

Görüldüğü gibi Queue koleksiyon sınıfında elemanlar koleksiyona arkadan katılırlar ve ilk giren eleman kuyruktan ilk çıkan eleman olur.

Stack ve Queue farklı yapılarda tasarlandıkları için elemanlarına farklı metodlar ile ulaşılmaktadır. Stack koleksiyon sınıfında, en son giren elemanı elde etmek için Pop metodu kullanılır. Koleksiyona bir eleman eklerken Push metodu kullanılır. Elbette eklenen eleman en son elemandır ve Pop metodu çağrıldığında elde edilecek olan ilk eleman halini alır. Ancak Pop metodu son giren elemanı verirken bu elemanı koleksiyondan siler. İşte bunun önüne geçen metod Peek metodudur. Şimdi diyebilirsiniz ki madem son giren elemanı siliniyor, Pop metodunu o zaman niye kullanıyoruz.

Hatırlarsanız, Stack koleksiyonunda, ilk giren elemanı elde etmek için bu elemanın üstünde yer alan tüm elemanları silmemiz gerektiğini söylemiştik. İşte bir döngü yapısında Pop metodu kullanıldığında, ilk giren elemana kadar inebiliriz. Tabi diğer elemanları kaybettikten sonra bunun çok büyük önem taşıyan bir eleman olmasını da istemiş olabiliriz.

Gelelim Queue koleksiyon sınıfının metodlarına. Dequeue metodu ile koleksiyona ilk giren elemanı elde ederiz. Ve bunu yaptığımız anda eleman silinir. Nitekim dequeue metodu pop metodu gibi çalışır. Koleksiyona eleman eklemek için ise, Enqueue metodu kullanılır. İlk giren elemanı elde etmek ve silinmemesini sağlamak istiyorsak yine stack koleksiyon sınıfında olduğu gibi, Peek metodunu kullanırız.

Bu koleksiyonların en güzel yanlarından birisi siz eleman sayısını belirtmediğiniz takdirde koleksiyonun boyutunu otomatik olarak kendilerinin ayarlamalarıdır. Stack koleksiyon sınıfı, varsayılan olarak 10 elemanlı bir koleksiyon dizisi oluşturur.(Eğer biz eleman sayısını yapıcı metodumuzda belirtmez isek).Eğer eleman sayısı 10'u geçerse, koleksiyon dizisinin boyutu otomatik olarak iki katına çıkar. Aynı prensip queue koleksiyon sınıfı içinde geçerli olmakla birlikte, queue koleksiyonu için varsayılan dizi boyutu 32 elemanlı bir dizidir.

Simdi dilerseniz, basit bir console uygulaması ile bu konuyu anlamaya çalışalım.

```
using System;
using System.Collections; /* Uygulamalarımızda koleksiyon sınıflarını kullanabilmek için
Collections isim uzayını kullanmamız gerekir.*/

namespace StackSample1
{
    class Class1
    {
        static void Main(string[] args)
        {
            Stack stc=new Stack(4); /* 4 elemanlı bir Stack koleksiyonu oluşturduk.*/

            stc.Push("Burak"); /*Eleman eklemek için Push metodu kullanılıyor.*/
            stc.Push("Selim");
            stc.Push("SENYURT");
            stc.Push(27);
            stc.Push(true);

            Console.WriteLine("Çıkan eleman {0}",stc.Pop().ToString());/* Pop metodu son
giren(kalan) elemanı verirken, aynı zamanda bu elemanı koleksiyon dizisinden siler.*/
            Console.WriteLine("Çıkan eleman {0}",stc.Pop().ToString());
            Console.WriteLine("Çıkan eleman {0}",stc.Pop().ToString());
            Console.WriteLine("-----");

            IEnumerator dizi=stc.GetEnumerator(); /* Koleksiyonun elemanlarını
IEnumerator arayüzünden bir nesneye aktarıyoruz.*/

            while(dizi.MoveNext()) /* dizi nesnesinde okunacak bir sonraki eleman var
olduğu sürece işleyecek bir döngü.*/
            {
                Console.WriteLine("Güncel eleman {0}",dizi.Current.ToString()); /* Current
metodu ile dizi nesnesinde yer alan güncel elemanı elde ediyoruz. Bu döngüyü
çalıştırdığımızda sadece iki elemanın dizide olduğunu görürüz. Pop metodu sagolsun.*/
```

```

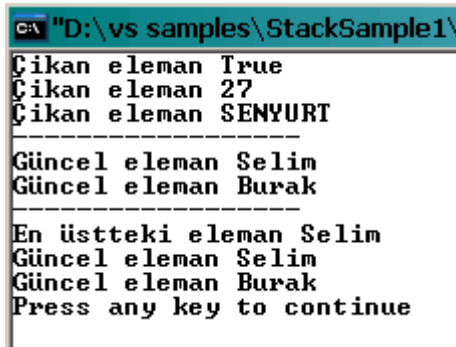
    }

    Console.WriteLine("-----");
    Console.WriteLine("En üstteki eleman {0}",stc.Peek()); /* Peek metodu son
giren elemani veya en üste kalan elemani verirken bu elemani koleksiyondan silmez.*/

    dizi=stc.GetEnumerator();

    while(dizi.MoveNext())
    {
        Console.WriteLine("Güncel eleman {0}",dizi.Current.ToString()); /* Bu
durumda yine iki eleman verildigini Peek metodu ile elde edilen elemanın koleksiyondan
silinmedigini görürüz.*/
    }
}
}
}
}
}

```



```

C:\> "D:\vs samples\StackSample1\
Çıkan eleman True
Çıkan eleman 27
Çıkan eleman SENYURT
-----
Güncel eleman Selim
Güncel eleman Burak
-----
En üstteki eleman Selim
Güncel eleman Selim
Güncel eleman Burak
Press any key to continue

```

3. Stack ile ilgili programın ekran çıktısı

Queue örneğimiz ise aynı kodlardan oluşuyor sadece metod isimleri farklı.

```

using System;
using System.Collections;

namespace QueueSample1
{
    class Class1
    {
        static void Main(string[] args)
        {
            Queue qu=new Queue(4);

            qu.Enqueue("Burak"); /*Eleman eklemek için Enqueue metodu kullaniliyor.*/
            qu.Enqueue("Selim");
            qu.Enqueue("SENYURT");
            qu.Enqueue(27);
            qu.Enqueue(true);

            Console.WriteLine("Çıkan eleman {0}",qu.Dequeue().ToString());/* Dequeue
metodu ilk giren(en alttaki) elemani verirken, aynı zamanda bu elemani koleksiyon
dizisinden siler.*/

```

```

        Console.WriteLine("Çıkan eleman {0}",qu.Dequeue().ToString());
        Console.WriteLine("Çıkan eleman {0}",qu.Dequeue().ToString());
        Console.WriteLine("-----");

        IEnumerator dizi=qu.GetEnumerator(); /* Koleksiyonin elemanlarını
IEnumerator arayüzünden bir nesneye aktarıyoruz.*/
        while(dizi.MoveNext()) /* dizi nesnesinde okunacak bir sonraki eleman var
olduğu sürece işleyecek bir döngü.*/
        {
            Console.WriteLine("Güncel eleman {0}",dizi.Current.ToString()); /* Current
metodu ile dizi nesnesinde yer alan güncel elemanı elde ediyoruz. Bu döngüyü
çalıştırdığımızda sadece iki elemanın dizide olduğunu görürüz. Dequeue metodu
sagolsun.*/
        }

        Console.WriteLine("-----");
        Console.WriteLine("En altta kalan eleman {0}",qu.Peek()); /* Peek metodu son
giren elemanı veya en üste kalan elemanı verirken bu elemanı koleksiyondan silmez.*/

        dizi=qu.GetEnumerator();

        while(dizi.MoveNext())
        {
            Console.WriteLine("Güncel eleman {0}",dizi.Current.ToString()); /* Bu
durumda yine iki eleman verildiğini Peek metodu ile elde edilen elemanın koleksiyondan
silinmediğini görürüz.*/
        }
    }
}
}

```

```

D:\vs samples\QueueSample1\
Çıkan eleman Burak
Çıkan eleman Selim
Çıkan eleman SENYURT
-----
Güncel eleman 27
Güncel eleman True
-----
En altta kalan eleman 27
Güncel eleman 27
Güncel eleman True
Press any key to continue

```

Sekil 4. Queue ile ilgili programın ekran çıktısı

Geldik bir makalemizin daha sonuna. Umuyorumki sizlere faydalı olabilecek bilgiler sunabilmişimdir. Bir sonraki makalemizde görüşmek dileğiyle hepinize mutlu günler dilerim.

Reflection(Yansıma) ile Tiplerin Sırrı Ortaya Çıkıyor

Hiç dotNET 'te yer alan bir tipin üyelerini öğrenebilmek istediniz mi? Örneğin var olan bir dotNET sınıfının veya sizin kendi yazmış olduğunuz yada bir başkasının yazdığı sınıfa ait tüm üyelerin neler olduğuna programatik olarak bakmak istediniz mi?

İşte bugünkü makalemizin konusu bu. Herhangi bir tipe (type) ait üyelerin neler olduğunu anlayabilmek. Bu amaçla, Reflection isim uzayını ve bu uzaya ait sınıfları kullanacağız. Bildiğiniz gibi .NET 'te kullanılan her şey bir tipe aittir. Yani herşeyin bir tipi vardır. Üyelerini öğrenmek isteğimiz bir tipi öncelikle bir Type değişkeni olarak alırız. (Yani tipin tipini alırız. Bu nedenle ben bu tekniğe Tip-i-Tip adını verdim). Bu noktadan sonra Reflection uzayına ait sınıfları ve metodlarını kullanarak ilgili tipe ait tüm bilgileri edinebiliriz. Küçük bir Console uygulaması ile konuyu daha iyi anlamaya çalışalım. Bu örneğimizde, System.Int32 sınıfına ait üyelerin bir listesini alacağız. İşte kodlarımız;

```
using System;

namespace ReflectionSample1
{
    class Class1
    {
        static void Main(string[] args)
        {
            Type tipimiz=Type.GetType("System.Int32");/* Öncelikle String sinifinin tipini
öğreniyoruz. */

            System.Reflection.MemberInfo[] tipUyeleri=tipimiz.GetMembers(); /* Bu satir
ile, System.String tipi içinde yer alana üyelerin listesini Reflection uzayında yer alan,
MemberInfo sinifi tipinden bir diziye aktarıyoruz. */

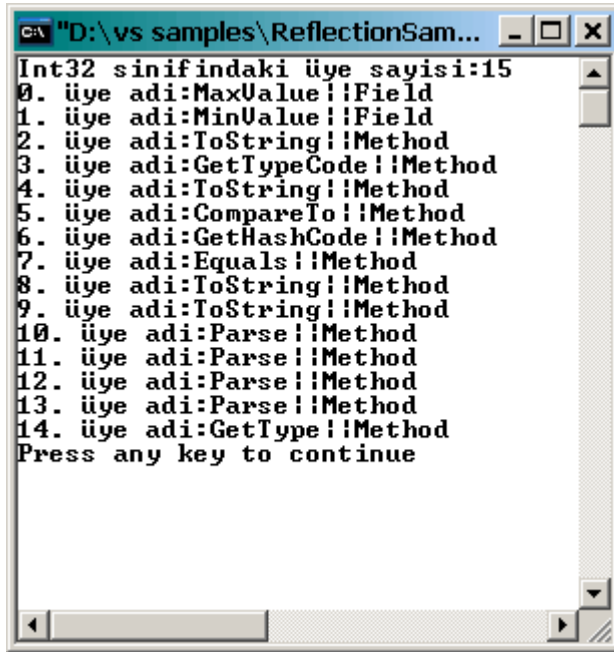
            Console.WriteLine(tipimiz.Name.ToString()+" sinifindeki üye
sayisi:"+tipUyeleri.Length.ToString());/* Length özelligi, MemeberInfo tipindeki dizimizde
yer alan üyelerin sayisini, (dolayisiyla System.String sinifi içinde yer alan
üyelerin sayisini) veriyor.*/
```

```

        /* Izleyen döngü ile, MemberInfo dizininde yer alan üyelerin birtakim bilgilerini
        ekrana yazıyoruz.*/
        for(int i=0;i<TIPUYELERI.LENGTH;++I)
        {
            Console.WriteLine(i.ToString()+" üye adi:"+tipUyeleri[i].Name.ToString()
            +"||"+tipUyeleri[i].MemberType.ToString()); /* Name özelliği üyenin adını verirken,
            MemberType özelliği ile, üyenin tipini alıyoruz. Bu üye tipi metod, özellik, yapıcı metod
            vb... dir.*/
        }
    }
}
}

```

Uygulamayı çalıştırdığımızda aşağıdaki ekran görüntüsünü elde ederiz.



Şekil 1. System.Int32 sınıfının üyeleri.

Üye listesini incelediğimizde 15 üyenin olduğunu görürüz. Metodlar, alanlar vardır. Ayrıca dikkat ederseniz, Parse , ToString metodları birden fazla defa geçmektedir. Bunun nedeni bu metodların overload (aşırı yüklenmiş) versiyonlara sahip olmasıdır.

Kodları incelediğimizde, System.Int32 sınıfına ait tipleri GetMembers metodu ile, System.Reflection uzayında yer alan MemberInfo sınıfı tipinden bir diziye aldığımızı görürüz. İşte olayın önemli kodları bunlardan oluşmaktadır. MemberInfo dışında kullanılabileceğimiz başka Reflection sınıfları da vardır. Bunlar;

ConstructorInfo	Tipe ait yapıcı metod üyelerini ve bu üyelere ait bilgilerini içerir.
EventInfo	Tipe ait olayları ve bu olaylara ait bilgileri içerir.
MethodInfo	Tipe ait metodları ve bu metodlara ait bilgileri içerir.
FieldInfo	Tip içinde kullanılan alanları ve bu alanlara ilişkin bilgileri içerir.
ParameterInfo	Tip içinde kullanılan parametreleri ve bu parametrelere ait bilgileri içerir.

PropertyInfo

Tip içinde kullanılan özellikleri ve bu özelliklere ait bilgileri içerir.

Tablo 1. Reflection Uzayının Diğer Kullanışlı Sınıfları

Şimdi bu sınıflara ait örneklerimizi inceleyelim.

Bu kez bir DataTable sınıfının üyelerini inceleyeceğiz. Örnek olarak, sadece olaylarını ve bu olaylara ilişkin özelliklerini elde etmeye çalışalım. Bu örneğimizde, yukarıda bahsettiğimiz tip-i-tip tekniğini biraz değiştireceğiz. Nitekim bu tekniği uygulamamız halinde bir hata mesajı alırız. Bunun önüne geçmek için, bir DataTable örneği (instance) oluşturup bu örneğin tipinden hareket edeceğiz. Dilerseniz hemen kodlarımıza geçelim.

```
using System;

namespace ReflectionSample2
{
    class Class1
    {
        static void Main(string[] args)
        {
            System.Data.DataTable dt=new System.Data.DataTable(); /* Bir DataTable
örneği(instance) yaratıyoruz.*/

            Type tipimiz=dt.GetType(); /* DataTable örneğimizin GetType metodunu
kullanarak, bu örneğin dolayısıyla DataTable sınıfının tipini elde ediyoruz. */

            System.Reflection.MethodInfo[] tipMetodlari=tipimiz.GetMethods(); /* Bu kez
sadece metodlari incelemek istedigimizden, GetMethods metodunu kullanıyor ve sonuçlari,
MethodInfo sinifi tipinden bir diziye aktariyoruz.*/

            Console.WriteLine(tipimiz.Name.ToString()+" sinifindaki metod
sayisi:"+tipMetodlari.Length.ToString()); /* Metod sayisini Length özelliği ile aliyoruz.*/

            /* Döngümüzü olusturuyor ve Metodlarimizi bir takim özellikleri ile birlikte
yazdiriyoruz.*/
            for(int i=0;i<TIPMETODLARI.LENGTH;++I)
            {
                Console.WriteLine("Metod adi:"+tipMetodlari[i].Name.ToString()+" |Dönüs
degeri:"+tipMetodlari[i].ReturnType.ToString()); /* Metodun ismini name özelliği ile
aliyoruz. Metodun dönüş tipini ReturnType özelliği ile aliyoruz. */

                System.Reflection.ParameterInfo[]
prmInfo=tipMetodlari[i].GetParameters();
                /* Bu satirda ise, i indeksli metoda ait parametre bilgilerini GetParameters
metodu ile aliyor ve Reflection uzayinda bulunan ParameterInfo sinifi tipinden bir diziye
aktariyoruz. Böylece ilgili metodun parametrelerine ve parametre bilgilerine erisebilicez.*/

                Console.WriteLine("-----Parametre
Bilgileri-----"+prmInfo.Length.ToString()+" parametre");
                /* Döngümüz ile i indeksli metoda ait parametrelerin isimlerini ve tiplerini
yazdiriyoruz. Bunun için Name ve ParameterType metodlarini kullaniyoruz.*/

                for(int j=0;j<PRMINFO.LENGTH;++J)
                {
```

```

        Console.WriteLine("P.Adi:"+prmInfo[j].Name.ToString()+" |
P.Tipi:"+prmInfo[j].ParameterType.ToString());
    }

    Console.WriteLine("----");
}
}
}
}
}

```

Şimdi uygulamamızı çalıştıralım ve sonuçlarına bakalım.

```

C:\> "D:\vs samples\ReflectionSample2\bin\Debug\ReflectionSample2.exe"

-----
Metod adi:Select !Dönüs degeri:System.Data.DataRow[]
-----Parametre Bilgileri-----2 parametre
P.Adi:filterExpression !P.Tipi:System.String
P.Adi:sort !P.Tipi:System.String
Metod adi:Select !Dönüs degeri:System.Data.DataRow[]
-----Parametre Bilgileri-----3 parametre
P.Adi:filterExpression !P.Tipi:System.String
P.Adi:sort !P.Tipi:System.String
P.Adi:recordStates !P.Tipi:System.Data.DataViewRowState
-----
Metod adi:BeginLoadData !Dönüs degeri:System.Void
-----Parametre Bilgileri-----0 parametre
-----
Metod adi:EndLoadData !Dönüs degeri:System.Void
-----Parametre Bilgileri-----0 parametre
-----
Metod adi:LoadDataRow !Dönüs degeri:System.Data.DataRow
-----Parametre Bilgileri-----2 parametre
P.Adi:values !P.Tipi:System.Object[]
P.Adi:fAcceptChanges !P.Tipi:System.Boolean
-----
Metod adi:GetType !Dönüs degeri:System.Type
-----Parametre Bilgileri-----0 parametre

```

Örneğin 3 parametrelili
Select metodu

Şekil 2. System.Data.DataTable tipinin metodları ve metod parametrelerine ait bilgiler.

Reflection teknikleri yardımıyla çalıştırdığımız programa ait sınıflarında bilgilerini elde edebiliriz. İzleyen örneğimizde, yazdığımız bir sınıfa ait üye bilgilerine bakacağız.

Üyelerine bakacağımız sınıfın kodları;

```

using System;

namespace ReflectionSample3
{
    public class OrnekSinif
    {
        private int deger;

        public int Deger
        {
            get
            {
                return deger;
            }
            set
            {
                deger=value;
            }
        }

        public string metod(string a)
        {
            return "Burak Selim SENYURT";
        }

        int yas=27;
        string dogum="istanbul";
    }
}

```

ikinci sınıfımızın kodları;

```

using System;
using System.Reflection;

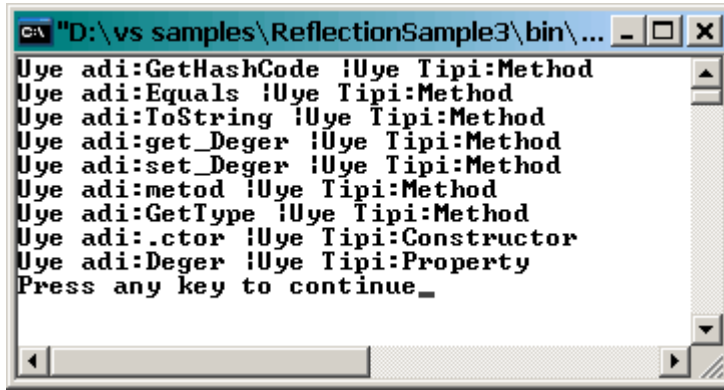
namespace ReflectionSample3
{
    class Class1
    {
        static void Main(string[] args)
        {
            Type tipimiz=Type.GetType("ReflectionSample3.OrnekSinif");

            MemberInfo[] tipUyeleri=tipimiz.GetMembers();

            for(int i=0;i<TIPUYELERI.LENGTH;++I)
            {
                Console.WriteLine("Uye adi:"+tipUyeleri[i].Name.ToString()+" |Uye
Tipi:"+tipUyeleri[i].MemberType.ToString());
            }
        }
    }
}

```

Şimdi uygulamamızı çalıştıralım.



Şekil 3. Kendi yazdığımız sınıf üyelerinde bakabiliriz.

Peki kendi sınıfımıza ait bilgileri edinmenin bize ne gibi bir faydası olabilir. İşte şimdi tam dışımıza göre bir örnek yazacağız. Örneğimizde, bir tablodaki verileri bir sınıf içersinden tanımladığımız özelliklere alacağız. Bu uygulamamız sayesinde sadece tek satırlık bir kod ile, herhangi bir kontrolü veriler ile doldurabileceğiz. Bu uygulamada esas olarak, veriler veritabanındaki tablodan alınacak ve oluşturduğumuz bir koleksiyon sınıfından bir diziye aktarılacak. Oluşturulan bu koleksiyon dizisi, bir DataGridView kontrolü ile ilişkilendirilecek. Teknik olarak kodumuzda, Reflection uzayının PropertyInfo sınıfını kullanarak, oluşturduğumuz sınıfa ait özellikler ile tablodaki alanları karşılaştıracak ve uygun iseler bu özelliklere tabloda karşılık gelen alanlar içindeki değerleri aktaracağız. Dilerseniz kodumuz yazmaya başlayalım.

```
using System;
using System.Reflection;
using System.Data;
using System.Data.SqlClient;

namespace ReflectDoldur
{
    public class Kitap
    {
        private string kitapAdi;
        private string yayimci;

        /* Özelliklerimizin adlarının tablmuzdan alacagimiz alan adlari ile ayni olmasına
dikkat edelim.*/
        public string Adi
        {
            get
            {
                return kitapAdi;
            }
            set
            {
                kitapAdi=value;
            }
        }

        public string BasimEvi
        {

```

```

        get
        {
            return yayimci;
        }
        set
        {
            yayimci=value;
        }
    }

```

/* Yapici metodumuz parametre olarak geçirilen bir DataRow degiskenine sahip. Bu degisken ile o anki satiri aliyoruz.*/

```

public Kitap(System.Data.DataRow dr)
{
    PropertyInfo[] propInfos=this.GetType().GetProperties(); /* this ile bu sinifi
    temsil ediyoruz. Bu sinifin tipini alip bu tipe ait özellikleri elde ediyor ve bunlar bir
    PropertyInfo sinifi tipinden diziye aktariyoruz.*/

```

/* Döngümüz ile tüm özellikleri geziyoruz. Eger metodumuza parametre olarak geçirilen dataRow degiskenimiz, bu özelliğin adında bir alan içeriyorsa, bu özelliğe ait SetValue metodunu kullanarak özelliğimize, ilgili tablo alanının değerini aktariyoruz.*/

```

    for(int i=0;i<PROPINFOS.LENGTH;++I)
    {
        if(propInfos[i].CanWrite) /* Burada özelliğimizin bir Set bloguna sahip olup
        olmadigina bakiliyor. Yani özelliğimizin yazilabilir olup olmadigina. Bunu saglayan
        özelliğimiz CanWrite. Eger özellik yazilabilir ise (yada baska bir deyişle readonly değil ise)
        true değerini döndürür.*/

```

```

        {
            try
            {
                if(dr[propInfos[i].Name]!=null) /* dataRow degiskeninde, özelliğimin
                adındaki alanın değerine bakiliyor. Null değer değil ise SetValue ile alanın değeri
                özelliğimize yazdiriliyor. */

```

```

                {
                    propInfos[i].SetValue(this,dr[propInfos[i].Name],null);
                }
                else
                {
                    propInfos[i].SetValue(this,null,null);
                }
            }
            catch
            {
                propInfos[i].SetValue(this,null,null);
            }
        }
    }
}

```

/* KitapKoleksiyonu sinifimiz bir koleksiyon olacak ve tablomuzdan alacagimiz iki alana ait verileri Kitap isimli nesnelerde saklanmasini sagliyacakti. Bu nedenle, bir CollectionBase sinifinden türetilirdi. Böylece sinifimiz içinde indeksleyiciler kullanabilecegiz.*/

```

public class KitapKoleksiyonu: System.Collections.CollectionBase
{
    public KitapKoleksiyonu()
    {
        SqlConnection conFriends=new SqlConnection("data source=localhost;initial
catalog=Friends;integrated security=sspi");
        SqlDataAdapter da=new SqlDataAdapter("Select Adi,BasimEvi From
Kitaplar",conFriends);
        DataTable dtKitap=new DataTable();
        da.Fill(dtKitap);

        foreach(DataRow drow in dtKitap.Rows)
        {
            this.InnerList.Add(new Kitap(drow));
        }
    }

    public virtual void Add(Kitap _kitap)
    {
        this.List.Add(_kitap);
    }

    public virtual Kitap this[int Index]
    {
        get
        {
            return (Kitap)this.List[Index];
        }
    }
}

```

Ve işte formumuzda kullandığımız tek satırlık kod;

```

private void btnDoldur_Click(object sender, System.EventArgs e)
{
    dataGrid1.DataSource = new ReflectDoldur.KitapKoleksiyonu();
}

```

Şimdi uygulamamızı çalıştıralım ve bakalım.

Adi	BasimEvi
Delphi 5'e Bakış	SECKIN
Delphi 5 Uygulama Geliştirme Kılavuzu	ALFA
Delphi 5 Kullanım Kılavuzu	ALFA
Microsoft Visual Basic 6.0 Geliştirmek Üst	ARKADAS
Visual Basic 6 Temel Başlangıç Kılavuzu	SISTEM
Microsoft Visual Basic 6 Temel Kullanım K	ALFA
ASP ile E-Ticaret Programcılığı	SISTEM
ASP 3.0	SISTEM
ASP ile web programcılığı ve elektronik tic	PUSULA
Herkes İçin ASP İle Veritabanı Yönetimi 3.	ALFA

Doldur

Şekil 4. Programın Çalışmasının Sonucu

Görüldüğü gibi tablomuzdaki iki Alana ait veriler yazdığımız KitapKoleksiyonu sınıfı yardımıyla, her biri Kitap tipinde bir nesne alan koleksiyonumuza eklenmiş ve bu sonuçlarda dataGrid kontrolümüze bağlanmıştır. Siz bu örneği dahada iyi bir şekilde geliştirebilirsiniz. Umuyorumki bu örnekte yapmak istediğimizi anlamışsınızdır. Yansıma tekniğini bu kod içinde kısa bir yerde kullandık. Sınıfın özelliklerinin isminin, tablodaki alanların ismi ile aynı olup olmadığını ve aynı isler yazılabilir olup olmadıklarını öğrenmekte kullandık.

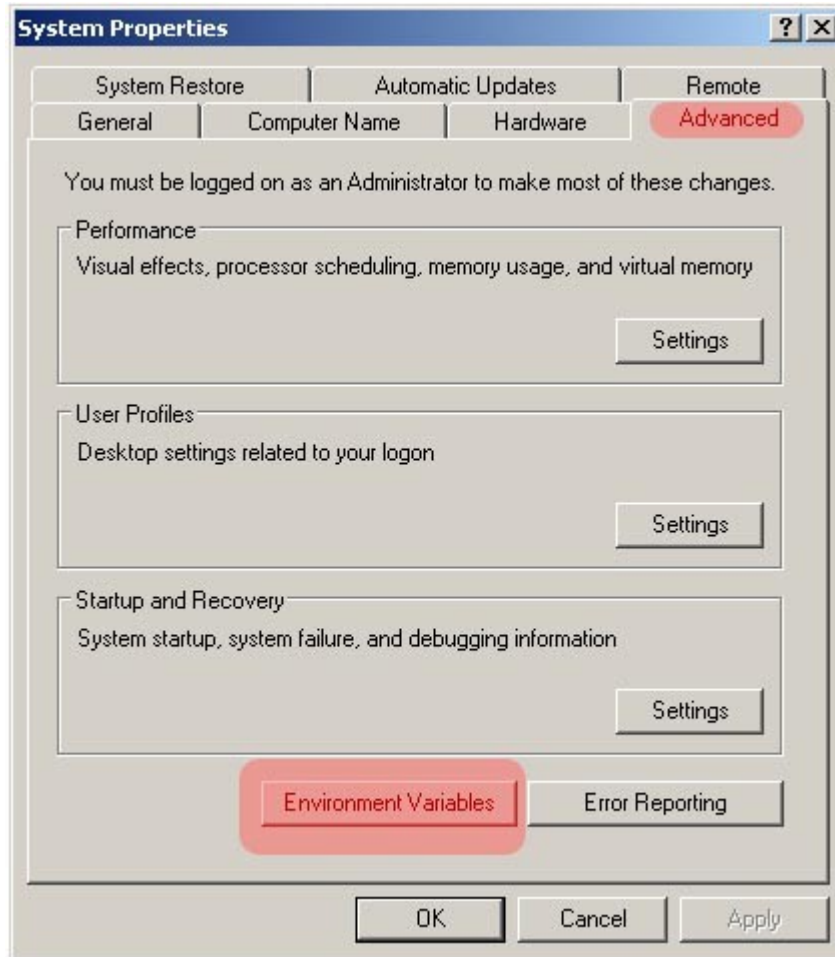
Değerli Okurlarım. Geldik bir makalemizin daha sonuna. Hepinize mutlu günler dilerim.

C# Komut Satırı Derleyicisi(csc.exe) ve Parametreleri

Bu yazıda sizlere önemli bir referans kaynağı olacağını düşündüğüm C# komut satırı derleyicisinin özelliklerini ve parametrelerini bir arada inceleyeceğiz. Böyle bir kaynağı oluşturmamdaki sebep C# komut derleyicisinin kullanımı ile ilgili bana gelen onlarca e-postaya toplu olarak cevap verebilmek.

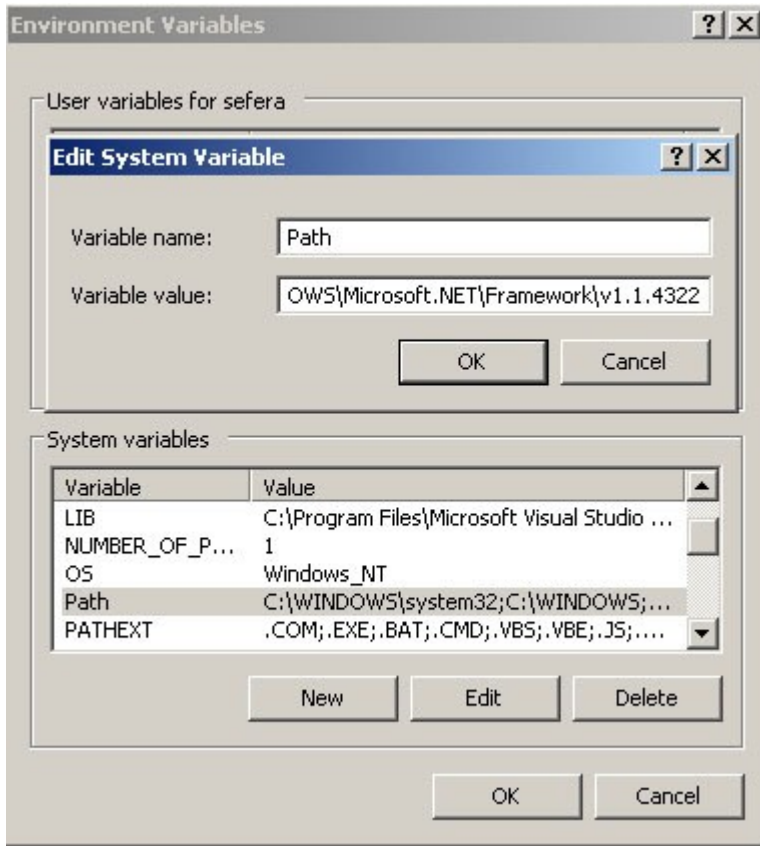
Bildiğiniz üzere .NET ortamında etkili bir şekilde geliştirme yapabilmek için Visual Studio.NET aracına ihtiyaç duyuyoruz. Ancak bu geliştirme ortamı olmadan da her tür .NET uygulamasını geliştirme imkanına sahibiz. Bu imkanı sağlayan en önemli araç elbetteki C#'ın komut satırından da çalışan csc.exe isimli derleyicisidir. C# komut satırı derleyicisi .NET Framework SDK ile birlikte ücretsiz olarak dağıtılmaktadır. Dolayısıyla .NET ortamında uygulama geliştirmek için yapmamız gereken tek şey www.microsoft.com sitesinden .NET Framework SDK'nın son sürümünü bilgisayarınıza indirmek ve kurmaktır.

C# derleyicisi komut satırından basitçe kullanılabilir. C# derleyicisini komut satırından en etkili bir şekilde kullanabilmek için işletim sisteminizde bir takım ayarlar yapmanız gerekmektedir. Örneğin siz komut satırında herhangi bir dizin içerisindeyken bile csc.exe çalıştırılabilir dosyasını çalıştırabilmeniz için csc.exe dosyasının bulunduğu dizini sistem özelliklerinde tanımlamanız gerekmektedir. Bu işlemi yapmak için kontrol panelinden system ikonunu ve ardından advanced sekmesini aşağıdaki gibi seçin.



Advanced sekmesine geçtikten sonra yukarıdaki şekilde kırmızı ile işaretlenen "Environment Variables" düğmesine tıklayın.(Kullandığınız işletim sistemine göre bu butonun yeri değişik olabilir. Yukarıdaki ekran görüntüsü Windows XP işletim sistemine aittir.) Karşınıza çıkacak olan aşağıdaki ekrandan "System Variables" alanından Path

seçeneğini seçin.

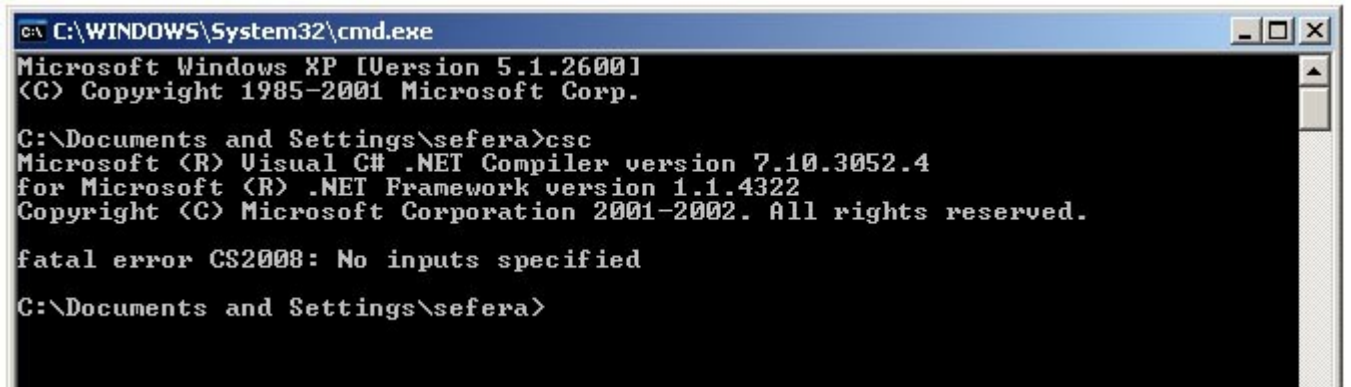


Path seçeneği tıklanarak açılan penceredeki Variable value alanının sonuna ; karakterini ekledikten sonra csc.exe dosyasının bulunduğu dizini yazın. csc.exe dosyası kullandığınız işletim sistemini göre değişiklik gösterebilir ancak genellikle aşağıdaki gibidir.

C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322

Son dizin kullandığınız Framework'un versiyon numarasını göstermektedir.

Yukarıdaki işlemlerin doğru bir şekilde tamamlandığını kontrol etmek için komut satırını açın ve "csc" yazın. Eğer aşağıdaki hata ile karşılaşıyorsanız işlemlerinizi başarı ile gerçekleştirmiş demektir.



Bu hatanın sebebi csc.exe programına derlenecek dosyayı parametre olarak vermememizdir.

Derleme İşlemi

csc.exe ile en basit derleme işlemi bir girdi dosya ve çıktı dosya belirtme ile yapılır. Örneğin kaynakkod.cs dosyasını Program.exe şeklinde derlemek için aşağıdaki komutu çalıştırmamız yeterlidir.

```
> csc kaynakkod.cs /out:Program.exe
```

Eğer out parametresini kullanmayıp komutu

```
> csc kaynakkod.cs
```

şeklinde çalıştırsaydık derleme işlemi başarılı olurdu ancak oluşturulan çalıştırılabilir dosyanın adı kaynakkod.exe olurdu.

Proje Tipleri ve target parametresi

.NET ortamında birden fazla proje tipi vardır ve dolayısıyla her bir proje tipinin derleme biçimi farklıdır. Örneğin bir önceki komutumuz derlem işlemini bir konsol uygulamasına göre yapacaktır. Esasında csc.exe derleyicisinin varsayılan derleme biçimide budur. Eğer derleme işlemini farklı uygulama tipleri için yapacak olursak derleyicinin target parametresini kullanmamız gerekir. Örneğin kaynak kodumuzu bir windows uygulaması olacak şekilde derlemek istiyorsak derleme komutu aşağıdaki gibi olmalıdır.

```
> csc kaynakkod.cs /target:winexe /out:Program.exe
```

yada

```
> csc kaynakkod.cs /t:winexe /out:Program.exe
```

Eğer kaynak kodumuzu çalıştırılabilir bir uygulama yerine bir kütüphane dosyası olacak şekilde derlemek istiyorsak aşağıdaki komutu kullanmalıyız.

```
> csc kaynakkod.cs /target:library /out:Program.exe
```

Diğer bir derleme biçimi ise modül derlemesidir. Modüller içinde manifest dediğimiz metadataları olmayan yalnızca kod bilgilerini içeren dosyalardır. Modüller çalıştırılabilir değildir. Dolayısıyla modüller ancak manifest bilgisi olan başka bir derlenmiş kütüphaneye eklenmek için kullanılabilir. Modül şeklinde derleme için aşağıdaki komut kullanılmalıdır.

```
> csc kaynakkod.cs /target:module /out:Program.exe
```

Referans Bilgileri ve Response Dosyaları

csc.exe derleyicisi derleme işlemini başarı ile gerçekleştirebilmesi için bazı kütüphanelere ihtiyaç duyar. Bu kütüphaneler assembly dosyaları içinde barındırılmıştır. Bu kütüphanelerin projemizde kullanıldığını belirtmek için reference parametresi kullanılır. Eğer referans vermeniz gereken assembly dosyaları fazla ise bu işi otomatikleştirmek için response. dosyaları kullanılır. response dosyasının yerine belirtmek için @ karakteri kullanılır. Örnek bir derleme biçimi aşağıdaki gibidir.

```
csc @response_dosyası /out:Program.exe kaynakdosya.cs
```

Derleyici Parametrelerine Toplu Bakış

Aşağıdaki listede csc derleyicisi ile kullanılabilecek bütün parametrelerin kullanımı ve açıklaması verilmiştir.

Parametre	Kullanımı
/doc:dosya_ismi	Kaynak dosyasındaki XML yorumlarını ayrıştırarak farklı bir dosyaya kaydetmek için kullanılan bir parametredir. Hatırlayacağınız üzere C# ile yazılmış kaynak kodda /// karakterlerinden sonra XML formatında yorum yazılabilmektedir.
/nooutput	Kaynak kodun derlenmesini sağlar ancak herhangi bir çalıştırılabilir dosya oluşturmaz. Bu parametre daha çok kaynak kodda hata ayıklama için kullanılmaktadır.
/optimize /optimize+ /optimize-	Derleme işlemi sırasında kaynak kodda herhangi bir optimizasyonun yapıp yapılmayacağını belirten bir parametredir. optimize ile optimize+ parametresi eşdeğerdedir.
/addmodule:modül_dosyası	Daha önce /module parametresi ile oluşturulan modüllerin herhangi bir çalıştırılabilir dosyaya eklenmesi amacıyla kullanılır. Birden fazla modül dosyasını eklemek için ; karakteri ile modül dosyalarını ayırmak gerekir.
/nostdlib /nostdlib+ /nostdlib-	.NET'in standart kütüphanesi olan System.dll'in otomatik olarak derlenecek koda eklenip eklenmemesini belirten parametredir. Eğer System.dll'i kendi kaynak dosyamızda kullanmayacaksak burdaki sınıfları kendimiz oluşturmalıyız. Çok nadir kullanılabilecek bir parametredir. /nostdlib ile /nostdlib+ parametresi eşdeğerdedir.
/reference:assembly_adi yada /r:assembly_adi	Başka bir assembly dosyasına ait meta verilere referans vermek için kullanılan parametredir. Assembly'nin bulunduğu yer göreceli adres olabileceği gibi tam adres de olabilir. Eğer birden fazla referans dosyası belirtilecekse ; karakteri ile ayrılmalıdır.
/define:SEMBOL /d:SEMBOL	Derleme zamanında önışlemci sembolü oluşturmak için kullanılır. Kaynak kod içerisinde yapılan #define ön işlemci komutuna karşılık gelmektedir.
/warn:<0,1,2,3,4> /w:<0,1,2,3,4>	Derleme işlemi sırasında verilecek uyarıların derecesini belirlemek için kullanılan parametredir. Eğer bütün uyarıların gösterilmesini istiyorsak /warn:4 şeklinde kullanmalıyız. /warn:0 parametresi ise hiçbir uyarının görüntülenmemesini sağlar. 0 ile 4 arasındaki değerler ise farklı tipteki uyarıların gösterilip gösterilmemesini sağlar.
/warnaserror /warnaserror+ /warnaserror-	Derleme sırasındaki uyarıların hata gibi işlenmesini sağlar. Bu genellikle idealist programcıların kullandığı bir parametredir. Eğer uyarı verecek bir durum varsa kodun derlenmemesi sağlanır. /warnaserror ile /warnaserror+ eşdeğerdedir.
/nowarn:uyarı_numarası	Belirtilen numaralı uyarının derleme sırasında verilmemesi için bu parametre kullanılır. Eğer birden fazla uyarının verilmemesini istiyorsak uyarı numaralarını ; karakteri ile

	ayırmamız gerekmektedir.
/fullpaths	Derleme sonrasında eğer herhangi bir dosyada hata var ise hatanın olduğu dosyanın tam adresinin hata ile belirtilmesini sağlayan parametredir.
/debug /debug+ /debug-	Hata ayıklamada kullanılacak dosyaların oluşturulması için kullanılan pparametredir. Eğer debugging işlemini aktif hale getirmek istiyorsak bu parametreyi kullanmamız gerekir. /debug ve /debug+ parametreleri eşdeğerdedir. Hata ayıklama işlemi varsayılan olarak aktif durumda değildir. /debug parametresinin ayrıca full ve pdbonly şeklinde iki seçeneği vardır. Eğer full seçeceği /debug:full şeklinde yazılırsa hata ayıklacı programı çalıştırılan program ile ilişkilendirilir.
/checked /checked+ /checked-	Aritmetik taşma işlemlerinde istisnai bir durumun oluşup oluşmayacağını bildiren parametredir. Varsayılan olarak bu aktif durumda değildir. Kaynak kod içerisinde bu işlemi checked anahtar sözcüklerini kullanarak yapabiliriz. Eğer taşım olduğunda istisnai durumun oluşmasını istiyorsak /checked yada /checked+ parametresini kullanmalıyız.
/bugreport:dosya_adi	Derleme sırasında kaynak kodda oluşabilecek problemlerin ve bu problemlerin önerilen çözümlerinin belirtilen dosyaya yazdırılmasını sağlayan parametredir. Bu parametre ile belirtilen dosyaya çeşitli derleme çıktıları da eklenir.
/unsafe	Kaynak kodda unsafe anahtar sözcüğünün kullanımını geçerli kılınmasını sağlayacak parametredir. Göstercileri kullanmak için unsafe anahtar sözcüğünü kullanmamız gerektiğini hatırlayın.
/recurse:dir /recurse:file	Derleme işlemine katılacak kaynak kodların alt klasörlerde aranmasını sağlayacak parametrelerdir. dir seçeneği ile aramaya başlanacak klasör belirtilir. Bu seçenek ile belirtilen klasör projenin varsayılan çalışma klasörüdür. Eğer file seçeneği kullanılırsa bu durumda belirtilen dosya için arama yapılacaktır. Bu seçenekte wildcard dediğimiz * karakteri kullanılabilir.
/main:sınıf_adi	Eğer kaynak kod dosyamızda birden fazla Main() metodu var ise programımızın hangi sınıftaki Main dosyasından başlayacağını belirten parametredir. Bu da kaynak kodumuzda birden fazla Main metodunun bulunabileceğinin göstergesidir.
/nologo	Derleme sonrasında ekranda gösterilen derleyici bilgilerinin kullanıcıya gösterilmemesini sağlayan parametredir. Kanımca çok faydalı olmayan bir parametredir.
/help yada /?	Derleyici parametreleri ile ilgili yardım bilgilerinin görüntülenmesini sağlayan parametrelerdir.
/incremental	Derleme işleminin optimize edilmiş biçimde meydana

/incremental+ /incremental- yada /incr /incr+ /incr-	gelmesini sağlayan parametrelerdir. Şöyleki, bir önceki derleme bilgileri .dbg ve .pdb dosyalarında tutularak yeni derleme işlemlerinde sadece değiştirilen metotların yeniden derlenmesi sağlanır. Farklı iki derleme işlemi arasındaki farklar ise .incr dosyasında saklanır. Varsayılan olarak bu parametre aktif durumda değildir. /incremental ile /incremental+ parametreleri eşdeğerdedir.
/codepage:id_no	Derleme işlemlerine katılacak kaynak kod dosyaları için bir karakter kodlaması numarası alan parametredir. Bu parametre daha çok kaynak kod dosyalarındaki karakterlerin sizin sisteminizde bulunmayan karakter kodlamasına denk düştüğü durumlarda kullanılır.
/baseaddress:adres	Yüklenecek DLL lerin belirlenecek bir adresten itibaren belleğe yüklenmesini sağlar. adres değeri 8,10 yada 16 lık sayı düzeninde olabilir.
@dosya_adi	Bazı derleyici parametrelerini otomatik olarak derleyiciye bildirmek için bu parametrelerin önceden yazıldığı dosyayı bildirmek için kullanılan parametredir.
/linkresource:dosya_adi /linkres:dosya_adi	Belirtilen .NET kaynak(resource) dosyasına bir bağlantı oluşturmak için bu parametreler kullanılabilir.
/resource:dosya_adi /res:dosya_adi	Belirtilen .NET kaynak(resource) dosyasını çıktı dosyasına gömmek için kullanılan parametredir. Birden fazla kaynak dosyası gömülecekse ; karakteri ile ayırmak gerekir.
/win32icon:dosya_adi	Belirtilen Win32 ikon dosyasını çıktı dosyasına eklemek için kullanılan parametredir.
/win32res:dosya_adi	Belirtilen Win32 kaynak(resource) dosyasını(.res) çıktı dosyasına eklemek için kullanılan parametredir.

Bu yazıda C# komut derleyicisinin parametrelerini ve kullanımlarını inceledik. Yukarıdaki tablonun sizin için iyi bir referans kaynağı olacağını umuyorum.

Boxing(Kutulamak) ve Unboxing(Kutuyu Kaldırmak)

Bugünkü makalemizde, Boxing ve Unboxing kavramlarını incelemeye çalışacağız. Boxing, değer türünden bir değişkeni referans türünden bir nesneye aktarmaktır. Unboxing işlemi ise bunun tam tersidir. Yani referans türü değişkenin işaret ettiği değeri tekrar , değer türü bir değişkene aktarmaktır. Bu tanımlarda karşımıza çıkan ve bilmemiz gereken en önemli noktalar, değer türü değişkenler ile referans türü nesnelerin bellekte tutuluş şekilleridir.

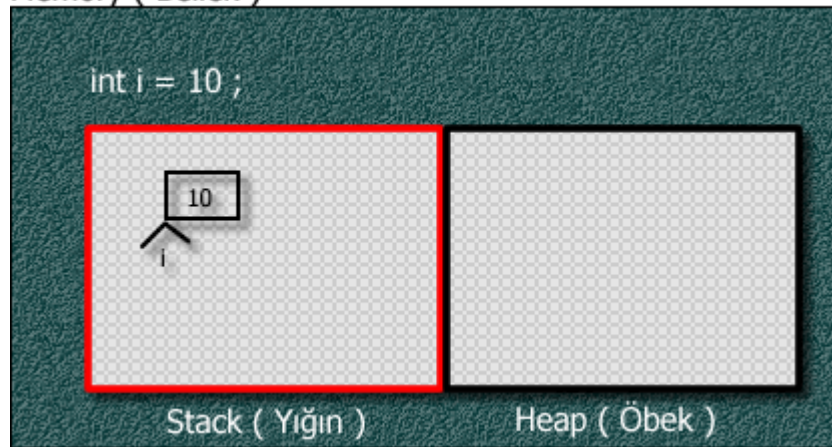
.Net ortamında iki tür veri tipi vardır. Referans tipleri (reference type) ve değer tipleri (value type). İki veri türünün bellekte farklı şekillerde tutulmaları nedeni ile boxing ve unboxing işlemleri gündeme gelmiştir. Bu nedenle öncelikle bu iki farklı veri tipinin bellekte tutuluş şekillerini iyice anlamamız gerekmektedir.

Bu anlamda karşımıza iki önemli bellek bölgesi çıkar. Yığın (stack) ve öbek(heap). Değer tipi değişkenler, örneğin bir integer değişken vb.. belleğin stack adı verilen kısmında tutulurlar. .Net'te yer alan değer türleri aşağıdaki tabloda yer almaktadır. Bu tiplerin stack bölgesinde nasıl tutulduğuna ilişkin açıklayıcı şekli de aşağıda görebilirsiniz.

Value type (Değer Tipleri)	
bool	long
byte	sbyte
char	short
decimal	Struct (Yapılar)
double	uint
Enum (Numaralandırıcılar)	ulong
float	ushort
int	

Tablo 1. Değer Tipleri

Memory (Bellek)



Şekil 1. Değer Tiplerinin Bellekte Tutuluşu

şekildende görüldüğü gibi, Değer Türleri bellekte, Stack dediğimiz bölgede tutulurlar. Şimdi buraya kadar anlaşılmayan bir şey yok. İlginç olan reference(başvuru) tiplerinin bellekte nasıl tutulduğudur. Adındanda anlaşıldığı gibi reference tipleri asıl veriye bir

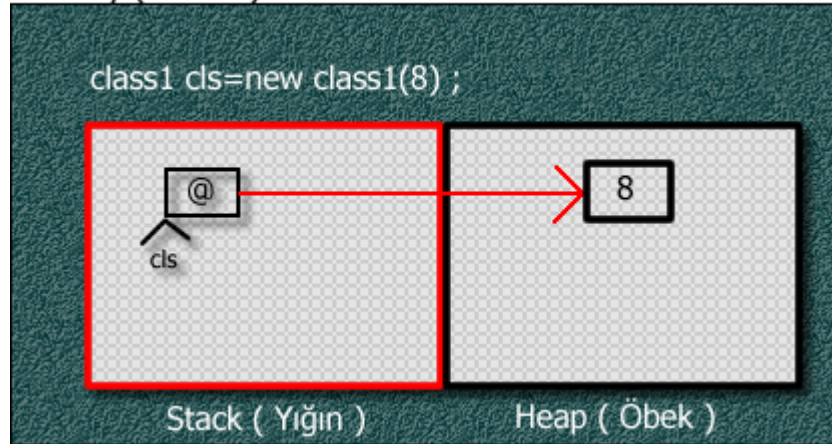
başvuru içerirler. Örneğin sınıflardan türettiğimiz nesneler bu tiplerdendir. Diğer başvuru tipleri ise aşağıdaki tabloda yer almaktadır.

Reference Type (Başvuru Tipleri)
Class (sınıflar)
Interface (arayüzler)
Delegate (delegeler)
Object
String

Tablo 2. Başvuru Tipleri

Şimdi gelin başvuru türlerinin bellekte nasıl tutulduklarına bakalım.

Memory (Bellek)



Şekil 2. Başvuru tiplerinin bellekte tutuluşu.

Görüldüğü gibi, başvuru tipleri hakikatende isimlerinin layığına vermekteler. Nitekim asıl veriler öbek'te tutulurken yığında bu verilere bir başvuru yer almaktadır.

İki veri türü arasındaki bir farkta bu verilerle işlemiz bittiğinde geri iade ediliş şekilleridir. Değer türleri ile işlemiz bittiğinde bunların yığında kapladıkları alanlar otomatik olarak yığına geri verilir. Ancak referans türlerinde sadece yığındaki başvuru sisteme geri verilir. Verilerin tutulduğu öbekteki alanlar, Garbage Collector'un denetimindedirler ve ne zaman sisteme iade edilecekleri tam olarak bilinmez. Bu ayrı bir konu olmakla beraber oldukça karmaşıktır. İlerleyen makalelerimizde bu konudan da bahsetmeye çalışacağım.

Değer türleri ile başvuru türleri arasındaki bu temel farktan sonra gelelim asıl konumuza. .NET'te her sınıf aslında en üst sınıf olan Object sınıfından türer. Yani her sınıf aslında System.Object sınıfından kalıtım yolu ile otomatik olarak türetilmiş olur. Sorun object gibi referans bir türe, değer tipi bir değerın aktarılmasında yaşanır. .NET'te herşey aslında birer nesne olarak düşünülebilir. Bir değer türünü bir nesneye atamaya çalıştığımızda, değer türünün içerdiği verinin bir kopyasının yığından alınıp, öbeğe taşınması ve nesnenin bu veri kopyasına başvurması gerekmektedir. İşte bu olay kutulama (boxing) olarak adlandırılır. Bu durumu minik bir örnek ile inceleyelim.


```

using System;

namespace boxunbox
{
    class Class1
    {
        static void Main(string[] args)
        {
            double db=509809232323;
            object obj;

            obj=db;

            Console.WriteLine(db.ToString());
            Console.WriteLine(obj.ToString());
            db+=1;
            Console.WriteLine(db.ToString());
            Console.WriteLine(obj.ToString());
        }
    }
}

```

Kodumuzun çalışmasını inceleyelim. Db isimli double değişkenimiz bir değer tipidir. Örnekte bu değer tipini object tipinden bir nesneye aktarıyoruz. Bu halde bu değerler içerisindeki verileri ekrana yazdırıyoruz. Sonra db değerimizi 1 arttırıyor ve tekrar bu değerlerin içeriğini ekrana yazdırıyoruz. İşte sonuç;

```

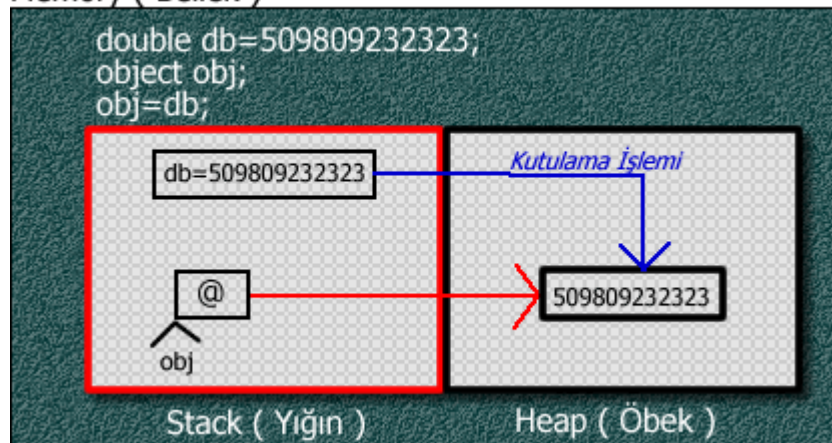
C:\ "D:\vs samples\boxunbox\bir
509809232323
509809232323
509809232324
509809232323
Press any key to continue_

```

şekil 3 Boxing İşlemi

Görüldüğü gibi db değişkenine yapılan arttırım object türünden obj nesnemize yansımamıştır. Çünkü boxing işlemi sonucu, obj nesnesi , db değerinin öbekteki kopyasına başvurmaktadır. Oysaki artım db değişkeninin yığında yer alan orjinal değeri üzerinde gerçekleşmektedir. Bu işlemi açıklayan şekil aşağıda yer almaktadır.

Memory (Bellek)



şekil 4. Boxing İşlemi

Boxing işlemi otomatik olarak yapılan bir işlemdir. Ancak UnBoxing işleminde durum biraz daha değişir. Bu kez , başvuru nesnemizin işaret ettiği veriyi öbekten alıp yığındaki bir değer tipi alanı olarak kopyalanması söz konusudur. İşte burada tip uyumsuzluğu denen bir kavramla karşılaşırız. Öbekten, yığına kopyalanacak olan verinin, yığında kendisi için ayrılan yerin aynı tipte olması veya öbekteki tipi içerebilecek tipte olması gerekmektedir. Örneğin yukarıdaki örneğimize unboxing işlemini uygulayalım. Bu kez integer tipte bir değer türüne atama gerçekleştirelim.

```
using System;

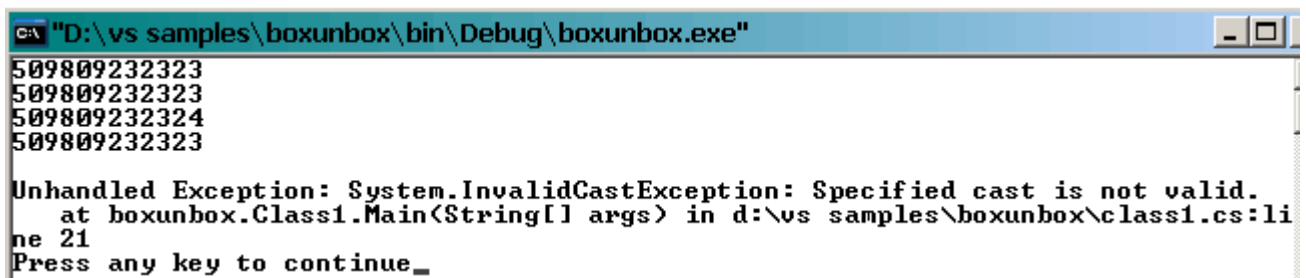
namespace boxunbox
{
    class Class1
    {
        static void Main(string[] args)
        {
            double db=509809232323;
            object obj;

            obj=db;

            Console.WriteLine(db.ToString());
            Console.WriteLine(obj.ToString());
            db+=1;
            Console.WriteLine(db.ToString());
            Console.WriteLine(obj.ToString());

            int intDb;
            intDb=(int)obj;
            Console.WriteLine(intDb.ToString());
        }
    }
}
```

Bu kodu çalıştırdığımızda InvalidCastException istisnasının fırlatılacağını göreceksiniz. Çünkü referenans tipimizin öbekte başvurduğu veri tipi integer bir değer için fazla büyüktür. Bu noktada (int) ile açıkça dönüşümü bildirmiş olsak dahi bu hatayı alırız.



```
C:\> "D:\vs samples\boxunbox\bin\Debug\boxunbox.exe"
509809232323
509809232323
509809232324
509809232323
Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
   at boxunbox.Class1.Main(String[] args) in d:\vs samples\boxunbox\class1.cs:line 21
Press any key to continue.
```

şekil 5. InvalidCastException İstisnası

Ancak küçük tipi, büyük tipe dönüştürmek gibi bir serbestliğimiz vardır. Örneğin,

```
using System;

namespace boxunbox
{
```

```

class Class1
{
    static void Main(string[] args)
    {
        double db=509809232323;
        object obj;

        obj=db;

        Console.WriteLine(db.ToString());
        Console.WriteLine(obj.ToString());
        db+=1;
        Console.WriteLine(db.ToString());
        Console.WriteLine(obj.ToString());

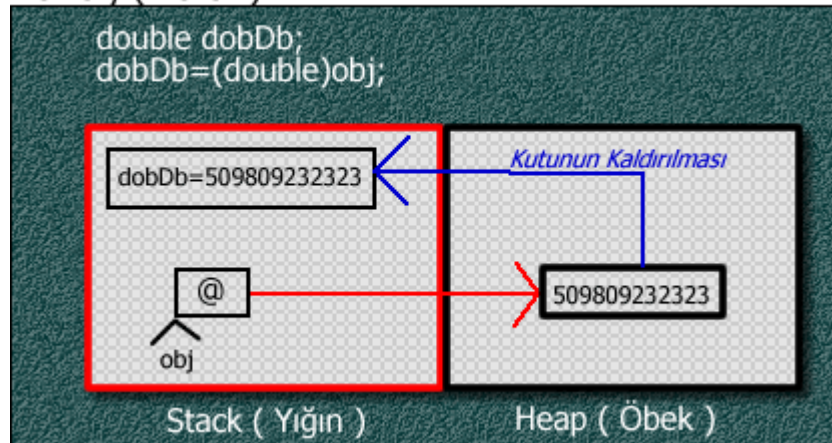
        /*int intDb;
        intDb=(int)obj;
        Console.WriteLine(intDb.ToString());*/

        double dobDb;
        dobDb=(double)obj;
        Console.WriteLine(dobDb.ToString());
    }
}

```

Bu durumda kodumuz sorunsuz çalışacaktır. Çünkü yığında yer alan veri tipi daha büyük boyutlu bir değer türünün içine koyulabilir. İşte buradaki aktarım işlemi unboxing olarak isimlendirilmiştir. Yani boxing işlemi ile kutulanmış bir veri kümesi, öbekten alınıp tekrar yığındaki bir alana konulmuş, dolayısıyla kutudan çıkartılmıştır. Olayın grafiksel açıklaması aşağıdaki gibidir.

Memory (Bellek)



Şekil 6. Unboxing İşlemi

Geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşmek dileğiyle hepinize mutlu günler dilerim.

C# ile Çok Kanallı(Multithread) Uygulamalar – 1

Bugünkü makelemiz ile birlikte threading kavramını en basit haliyle tanımaya çalışacağız, sonraki makalelerimizde de threading kavramını daha üst seviyede işlemeye çalışacağız.

Bugün hepimiz bilgisayar başındayken aynı anda pek çok uygulamanın sorunsuz bir şekilde çalıştığını görürüz. Bir belge yazarken, aynı zamanda müzik dinleyebilir, internet üzerinden program indirebilir ve sistemimizin kaynaklarının elverdiği ölçüde uygulamayla eşzamanlı olarak çalışabiliriz. Bu bize, günümüz işlemcilerinin ve üzerlerinde çalışan işletim sistemlerinin ne kadar yetenekli olduğunu gösterir. Gösterir mi acaba?

Aslında tek işlemcili makineler günümüzün modern sihirbazları gibidirler. Gerçekte çalışan uygulamaların tüm işlemleri aynı anda gerçekleşmemektedir. Fakat işlemciler öylesine büyük saat hızlarına sahiptirlerki. işlemcinin yaptığı, çalıştırılan uygulamaya ait işlemleri iş parçacıkları(thread) halinde ele almaktır. Her bir iş parçacığı bir işlemin birden fazla parçaya bölünmesinden oluşur. İşlemciler her iş parçacığı için bir zaman dilimi belirler. T zaman diliminde bir işlem parçacığı yürütülür ve bu zaman dilim bittiğinde işlem parçacığı geçici bir süre için durur. Ardından kuyrukta bekleyen diğer iş parçacığı başka bir zaman dilimi içinde çalıştırılır. Bu böylece devam ederken, işlemcimiz her iş parçacığına geri döner ve tüm iş parçacıkları sıra sıra çalıştırılır. Dedik ya, işlemciler bu işlemleri çok

yüksek saat ve frekans hızında gerçekleştirir. İşte bu yüksek hız nedeniyle tüm bu olaylar saniyenin milyon sürelerinde gerçekleşir ve sanki tüm bu uygulamalar aynı anda çalışıyor hissi verilir.

Gerçektende uygulamaları birbirleriyle paralel olarak ve eş zamanlı çalıştırmak aslında birden fazla işlemciye sahip sistemler için gerçekleşir.

Bugünkü uygulamamız ile, bahsetmiş olduğumuz threading kavramına basit bir giriş yapacağız. Nitekim threading kavramı ve teknikleri, uygulamalarda profesyonel olarak kod yazmayı gerektirir. Daha açık şekilde söylemek gerekirse bir uygulama içinde yazdığımız kodlara uygulayacağımız thread'ler her zaman avantaj sağlamaz. Bazı durumlarda dezavantaja dönüşüp programların daha yavaş çalışmasına neden olabilir. Nitekim thread'lerin çalışma mantığını iyi kavramak ve uygulamalarda titiz davranmak gerekir.

Örneğin thread'lerin zaman dilimlerine bölündüklerinde sistemin nasıl bir önceki veya daha önceki thread'i çalıştırabildiğini düşünelim. İşlemci zaman dilimini dolduran bir thread için donanımda bir kesme işareti bırakır, bunun ardından thread'e ait bir takım bilgiler belleğe yazılır ve sonra bu bellek bölgesinde Context adı verilen bir veri yapısına depolanır. Sistem bu thread'e döneceği zaman Context'te yer alan bilgilere bakar ve hangi donanımın kesme sinyali verdiğini bulur. Ardından bu sinyal açılır ve işlemin bir sonraki işlem parçasının çalışacağı zaman dilimine girilir. Eğer thread işlemini çok fazla kullanırsanız bu durumda bellek kaynaklarının da fazlası ile tüketmiş olursunuz. Bu thread'leri neden titiz bir şekilde programlamamız gerektiğini anlatan nedenlerden sadece birisidir. Öyleki yanlış yapılan thread programlamaları sistemlerin kilitlenmesine dahi yol açacaktır.

Threading gördüğünüz gibi çok basit olmayan bir kavramdır. Bu nedenle olayı daha iyi açıklayabileceğimi düşündüğüm örneklerime geçmek istiyorum. Uygulamamızın formu aşağıdaki şekildeki gibi olacak.

Şekil 1. Form Tasarımımız.

Şimdi kodlarımızı yazalım.

```
public void z1()
{
    for(int i=1;i<60;++i)
    {
        zaman1.Value+=1;
    }
}
```

```

        for(int j=1;j<100000000;++j)
        {
            j+=1;
        }
    }

public void z2()
{
    for(int k=1;k<100;++k)
    {
        zaman2.Value+=1;

        for(int j=1;j<250000000;++j)
        {
            j+=1;
        }
    }
}

private void btnBaslat_Click(object sender, System.EventArgs e)
{
    z1();
    z2();
}

```

Program kodlarımızı kısaca açıklayalım. z1 ve z2 isimli metodlarımız progressBar kontrollerimizin değerlerini belirli zaman aralıklarıyla arttırıyorlar. Bu işlemleri gerçekleştirmek için, Başlat başlıklı butonumuza tıklıyoruz. Burada önce z1 daha sonrada z2 isimli metodumuz çalıştırılıyor. Bunun sonucu olarak önce zaman1 isimli progressBar kontrolümüz doluyor ve dolması bittikten sonra zaman2 isimli progressBar kontrolümüzün value değeri arttırılarak dolduruluyor.

Şimdi bu programın şöyle çalışmasını istediğimizi düşünelim. Her iki progressBar'da aynı anda dolmaya başlasınlar. İsteddiğimiz zaman z1 ve z2 metodlarının çalışmasını durduralım ve tekrar başlatabilelim. Tekrar başlattığımızda ise progressBar'lar kaldıkları yerden dolmaya devam etsinler. Sözünü ettiğimiz aslında her iki metodunda aynı anda çalışmasıdır. İşte bu işi başarmak için bu metodları sisteme birer iş parçacığı (thread) olarak tanıtmalı ve bu thread'leri yönetmeliyiz.

.Net ortamında thread'ler için System.Threading isim uzayını kullanırız. Öncelikle programımıza bu isim uzayını ekliyoruz. Ardından z1 ve z2 metodlarını birer iş parçacığı olarak tanımlamamız gerekiyor. İşte kodlarımız.

```

public void z1()
{
    for(int i=1;i<60;++i)
    {
        zaman1.Value+=1;

        for(int j=1;j<100000000;++j)
        {
            j+=1;
        }
    }
}

```

```
}  
}
```

```
public void z2()  
{  
    for(int k=1;k<100;++k)  
    {  
        zaman2.Value+=1;  
  
        for(int j=1;j<25000000;++j)  
        {  
            j+=1;  
        }  
    }  
}
```

```
private void btnBaslat_Click(object sender, System.EventArgs e)  
{  
    z1();  
    z2();  
}
```

```
ThreadStart ts1;  
ThreadStart ts2;  
Thread t1;  
Thread t2;
```

```
private void btnBaslat_Click(object sender, System.EventArgs e)  
{  
    ts1=new ThreadStart(z1); /* ThreadStart iş parçacığı olarak kullanılacak metod için  
bir temsilcidir. Bu metod için tanımlanacak thread sınıfı nesnesi için paramtere olacak ve  
bu nesnenin hangi metodu iş parçacığı olarak göreceğini belirtecektir. */
```

```
    ts2=new ThreadStart(z2);
```

```
    t1=new Thread(ts1);
```

```
    t2=new Thread(ts2);
```

```
    t1.Start(); /* İş parçacığını Start metodu ile başlatıyoruz. */
```

```
    t2.Start();
```

```
    btnBaslat.Enabled=false;
```

```
}
```

```
private void btnDurdur_Click(object sender, System.EventArgs e)
```

```
{
```

```
    t1.Suspend(); /* İş parçacığı geçici bir süre için uyku moduna geçer. Uyku  
modundaki bir iş parçacığını tekrar aktif hale getirmek için Resume metodu kullanılır. */
```

```
    t2.Suspend();
```

```

}

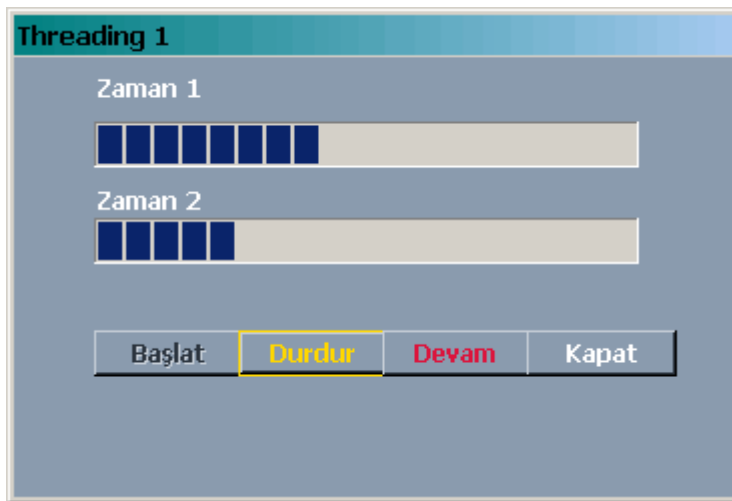
private void btnDevam_Click(object sender, System.EventArgs e)
{
    t1.Resume(); /* Uyku modundaki iş parçacığının kaldığı yerden devam etmesini sağlar. */

    t2.Resume();
}

private void btnKapat_Click(object sender, System.EventArgs e)
{
    if(t1.IsAlive) /* Eğer iş parçacıkları henüz sonlanmamışsa bunlar canlıdır ve IsAlive özellikleri true değerine sahiptir. Programımızda ilk biten iş parçacığı t1 olacağından onun bitip bitmediğini kontrol ediyoruz. Eğer bitmiş ise programımız close metodu sayesinde kapatılabilir. */
    {
        MessageBox.Show("Çalışan threadler var program sonlanamaz.");
    }
    else
    {
        this.Close();
    }
}
}

```

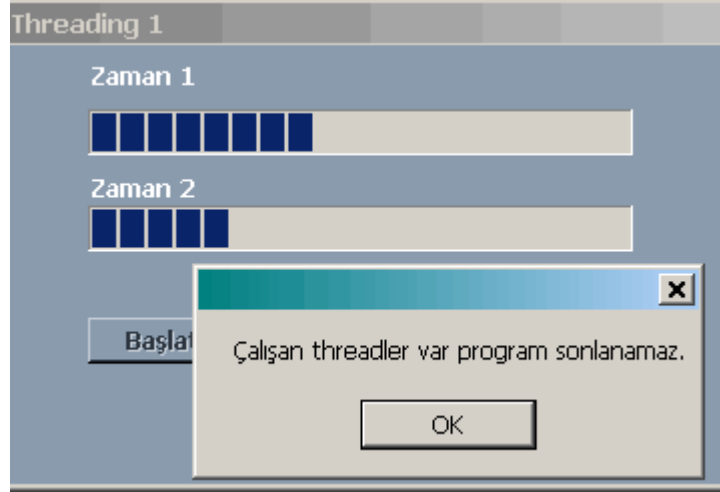
Uygulamamızda z1 ve z2 isimli metodlarımızı birer iş parçacığı (thread) haline getirdik. Bunun için System.Threading isim uzayında yer alan ThreadStart ve Thread sınıflarını kullandık. ThreadStart sınıfı , iş parçacığı olucak metodu temsil eden bir delegate gibi davranır. İş parçacıklarını başlatacak(start), durdurucak(suspend), devam ettirecek(resume) thread nesnelerimizi tanımladığımız yapıcı metod ThreadStart sınıfından bir temsilciyi parametre olarak alır. Sonuç itibarıyla kullanıcı Başlat başlıklı buton kontrolüne tıkladığında, her iki progressBar kontrolünde aynı zamanda dolmaya başladığını ve ilerlediklerini görürüz. Bu aşamada Durdur isimli button kontrolüne tıklarsak her iki progressBar'ın ilerleyişinin durduğunu görürüz. Nitekim iş parçacıklarının Suspend metodu çağırılmış ve metodların çalıştırılması durdurulmuştur.



Şekil 2. Suspend metodu sonrası.

Bu andan sonra tekrar Devam buton kontrolüne tıklarsak thread nesnelerimiz Resume metodu sayesinde çalışmalarına kaldıkları yerden devam ediceklerdir. Dolayısıyla

progressBar kontrollerimizde kaldıkları yerden dolmaya devam ederler. Bu sırada programı kapatmaya çalışmamız henüz sonlanmamış iş parçacıkları nedeni ile hataya neden olur. Bu nedenle Kapat buton kontrolünde IsAlive özelliği ile iş parçacıklarının canlı olup olmadığı yani metodların çalışmaya devam edip etmediği kontrol edilir. Eğer sonlanmamışsa kullanıcı aşağıdaki mesaj kutusu ile uyarılır.



Şekil 3. İş Parçacıkları henüz sonlanmamış ise.

Evet geldik Threading ile ilgili makale dizimizin ilk bölümünün sonuna . Bir sonraki makalemizde Threading kavramını daha da derinlemesine incelemeye çalışacağız. Hepinize mutlu günler dilerim.

C# ile Çok Kanallı(Multithread) Uygulamalar – 2

Bugünkü makalemizde iş parçacıklarının belli süreler boyunca nasıl durgunlaştırılabileceğini yani etkisizleştirebileceğimizi işlemeye çalışacağız. Ayrıca iş parçacıklarının henüz sonlanmadan önce nasıl yokedildiklerini göreceğiz.

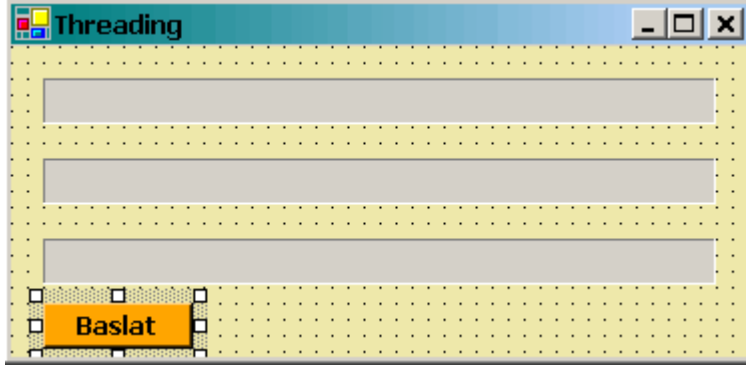
Bir önceki makalemizde hatırlayacak olursanız, iş parçacıkları haline getirdiğimiz metodlarımızda işlemleri yavaşlatmak amacı ile bazı döngüler kullanmıştık. Gerçek hayatta çoğu zaman, iş parçacıklarının belirli süreler boyunca beklemesini ve süre sona erdiğinde tekrardan işlemlerine kaldığı yerden devam etmesini istediğimiz durumlar olabilir. Önceki makalemizde kullandığımız Suspend metodu ile ilgili iş parçacığını durdurabiliyorduk. Bu ilgili iş parçacıklarını geçici süre ile bekletmenin yollarından birisidir. Ancak böyle bir durumda bekletilen iş parçacığını tekrar hareketlendirmek kullanıcının Resume metodunu çalıştırması ile olabilir. Oysaki biz, iş parçacığımızın belli bir süre boyunca beklemsini isteyebiliriz. İşte böyle bir durumda Sleep metodunu kullanırız. Bu metodun iki adet overload edilmiş versiyonu vardır.

```
public static void Sleep(int);
```

```
public static void Sleep(TimeSpan);
```


Biz bugünkü uygulamamızda ilk versiyonu kullanacağız. Bu versiyonda metodumuz parametre olarak int tipinde bir değer almaktadır. Bu değer milisaniye cinsinden süreyi bildirir. Metodun Static bir metod olduğu dikkatinizi çekmiş olmalıdır. Static bir metod olması nedeni ile, Sınıf adı ile birlikte çağırılmak zorundadır. Yani herhangi bir thread nesnesinin ardından Sleep metodunu yazamassınız. Peki o halde bekleme süresinin hangi iş parçacığı için geçerli olacağını nereden bileceğiz. Bu nedenle, bu metod iş parçacığı olarak tanımlanan metod blokları içerisinde kullanılır. Konuyu örnek üzerinden inceleyince daha iyi anlayacağız. Metod çalıştırıldığında parametresinde belirtilen süre boyunca geçerli iş parçacığını bekletir. Bu bekleme diğer parçacıkların çalışmasını engellemez. Süre sona erince, iş parçacığımız çalışmasına devam edecektir. Şimdi dilerseniz örnek bir uygulama geliştirelim ve konuya açıklık getirmeye çalışalım.

Formumuzda bu kez üç adet ProgressBar kontrolümüz var. Baslat başlıklı düğmeye bastığımızda iş parçacıklarımız çalışıyor ve tüm ProgressBar'lar aynı anda değişik süreler ile ilerliyor. Burada iş parçacıkları olarak belirlediğimiz metodlarda kullandığımız Sleep metodlarına dikkat edelim. Tabi kodlarımızı yazmadan önce System.Threading isim uzayını eklemeyi unutmayalım.



Şekil 1. Form Tasarımımız.

```
public void pb1Ileri()
{
    for(int i=1;i<100;++i)
    {
        pb1.Value+=1;
        Thread.Sleep(800);
    }
}

public void pb2Ileri()
{
    for(int i=1;i<100;++i)
    {
        pb2.Value+=1;
        Thread.Sleep(500); /* Metodumuz iş parçacığı olarak başladıktan sonra döngü
içince her bir artımdan sonra 500 milisaniye bekler. */
    }
}

public void pb3Ileri()
{
    for(int i=1;i<100;++i)
```

```

    {
        pb3.Value+=1;
        Thread.Sleep(300);
    }
}

/* ThreadStart temsilcilerimiz ve Thread nesnelerimizi tanımlıyoruz. */

ThreadStart ts1;
ThreadStart ts2;
ThreadStart ts3;

Thread t1;
Thread t2;
Thread t3;

private void btnBaslat_Click(object sender, System.EventArgs e)
{
    /* ThreadStart temsilcilerimizi ve Thread nesnelerimizi oluşturuyoruz. */
    ts1=new ThreadStart(pb1Ileri);

    ts2=new ThreadStart(pb2Ileri);

    ts3=new ThreadStart(pb3Ileri);

    t1=new Thread(ts1);

    t2=new Thread(ts2);

    t3=new Thread(ts3);

    /* Thread nesnelerimizi start metodu ile başlatıyoruz. */

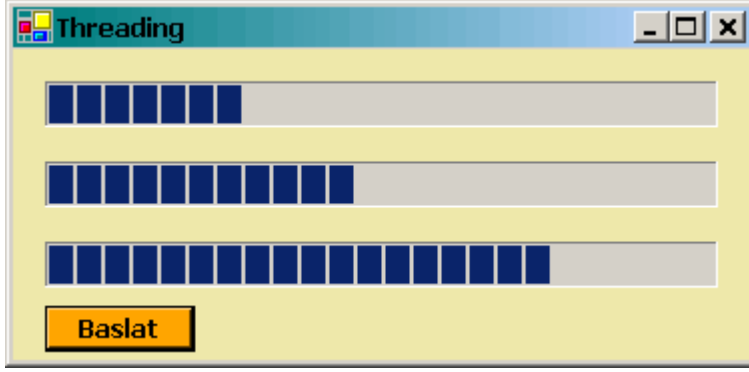
    t1.Start();

    t2.Start();

    t3.Start();
}

```

Uygulamamızı çalıştıralım. Her iş parçacığı Sleep metodu ile belirtilen süre kadar beklemeler ile çalışmasına devam eder. Örneğin pb3Ileri metodunda iş parçacığımız ProgressBar'ın Value değerini her bir arttırdıktan sonra 300 milisaniye bekler ve döngü bir sonraki değerden itibaren devam eder. Sleep metodu ile Suspend metodları arasında önemli bir bağ daha vardır. Bildiğiniz gibi Suspend metodu ile de bir iş parçacığını durdurabilmekteyiz. Ancak bu iş parçacığını tekrar devam ettirmek için Resume metodunu kullanmamız gerekiyor. Bu iki yöntem arasındaki fark idi. Diğer önemli olgu ise şudur; bir iş parçacığı metodu içinde, Sleep metodunu kullanmış olsak bile, programın herhangi bir yerinden bu iş parçacığı ile ilgili Thread nesnesinin Suspend metodunu çağırdığımızda, bu iş parçacığı yine duracaktır. Bu andan itibaren Sleep metodu geçerliliğini, bu iş parçacığı için tekrardan Resume metodu çağırılınca kadar kaybedecektir. Resume çağırısından sonra ise Sleep metodları yine işlemeye devam eder.



Şekil 2. Sleep Metodunun Çalışması

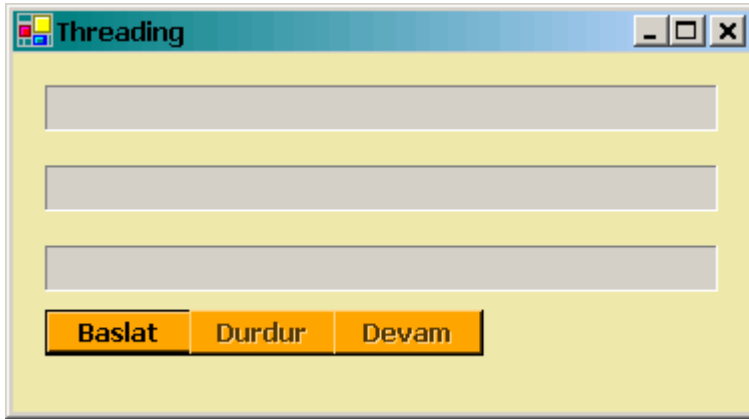
Şimdi gelelim diğer konumuz olan bir iş parçacığının nasıl yok edileceğine. Bir iş parçacığını yoketmek amacı ile Abort metodunu kullanabiliriz. Bu metod çalıştırıldığında derleyici aslında bir ThreadAbortException istisnası üretir ve iş parçacığını yoketmeye zorlar. Abort yöntemi çağırıldığında, ilgili iş parçacığını tekrar resume gibi bir komutla başlatamayız. Diğer yandan Abort metodu iş parçacığı ile ilgili metod için ThreadAbortException istisnasını fırlattığında (throw) , bu metod içinde bir try..catch..finally korumalı bloğunda bu istisnayı yakalayabiliriz veya Catch bloğunda hiç bir şey yazmasa ise program kodumuz kesilmeden çalışmasına devam edecektir.

Abort metodu ile bir iş parçacığı sonlandırıldığında, bu iş parçacığını Start metodu ile tekrar çalıştırmak istersek;

"ThreadStateException' Additional information: Thread is running or terminated; it can not restart."

hatasını alırız. Yani iş parçacığımızı tekrar baştan başlatmak gibi bir şansımız yoktur.

Şimdi bu metodu inceleyeceğimiz bir kod yazalım. Yukarıdaki uygulamamızı aşağıdaki şekilde geliştirelim.



Şekil 3. Form Tasarımımız.

Kodlarımıza geçelim.

```
public void pb1Ileri()
```

```
{  
    try  
    {
```

```

        for(int i=1;i<100;++i)
        {
            pb1.Value+=1;
            Thread.Sleep(800);
        }
    }
    catch(ThreadAbortException hata)
    {

    }
    finally
    {

    }
}

public void pb2Ileri()
{
    for(int i=1;i<100;++i)
    {
        pb2.Value+=1;
        Thread.Sleep(500); /* Metodumuz iş parçacığı olarak başladıktan sonra döngü
içince her bir artımdan sonra 500 milisaniye bekler. */
    }
}

public void pb3Ileri()
{
    for(int i=1;i<100;++i)
    {
        pb3.Value+=1;
        Thread.Sleep(300);
    }
}

/* ThreadStart temsilcilerimiz ve Thread nesnelerimizi tanımlıyoruz. */

ThreadStart ts1;

ThreadStart ts2;

ThreadStart ts3;

Thread t1;

Thread t2;

Thread t3;

private void btnBaslat_Click(object sender, System.EventArgs e)

```

```
{
    /* ThreadStart temsilcilerimizi ve Thread nesnelerimizi oluşturunuz. */
    ts1=new ThreadStart(pb1Ileri);
    ts2=new ThreadStart(pb2Ileri);
    ts3=new ThreadStart(pb3Ileri);
    t1=new Thread(ts1);
    t2=new Thread(ts2);
    t3=new Thread(ts3);

    /* Thread nesnelerimizi start metodu ile başlatıyoruz. */
    t1.Start();
    t2.Start();
    t3.Start();

    btnBaslat.Enabled=false;
    btnDurdur.Enabled=true;
    btnDevam.Enabled=false;
}

private void btnDurdur_Click(object sender, System.EventArgs e)
{
    t1.Abort(); /* t1 isimli Thread'imizi yok ediyoruz. Dolayısıyla pb1Ileri isimli
    metodumuzunda çalışmasını sonlandırmış oluyoruz. */

    /* Diğer iki iş parçacığını uyutuyoruz. */
    t2.Suspend();
    t3.Suspend();

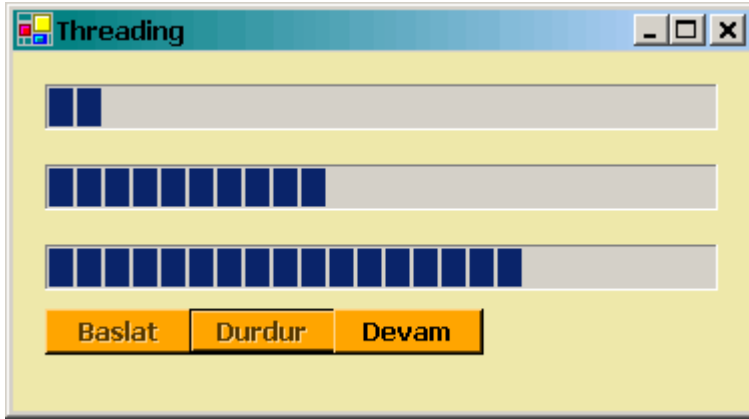
    btnDurdur.Enabled=false;
    btnDevam.Enabled=true;
}

private void btnDevam_Click(object sender, System.EventArgs e)
{
    /* İş parçacıklarını tekrar kaldıkları yerden çalıştırıyoruz. İşte burada t1 thread
    nesnesini Resume metodu ile tekrar kaldığı yerden çalıştıramayız. Bu durumda
    programımız hataya düşecektir. Nitekim Abort metodu ile Thread'imiz sonlandırılmıştır.
```

Aynı zamanda Start metodu ile Thread'imizi baştında başlatamayız.*/*

```
t2.Resume();  
  
t3.Resume();  
  
btnDurdur.Enabled=true;  
  
btnDevam.Enabled=false;  
}
```

Uygulamamızı deneyelim.



Şekil 4. Uygulamanın çalışması sonucu.

Değerli okurlarım geldik bir makalemizin daha sonuna. Bir sonraki makalemizde de Threading konusunu işlemeye devam edeceğiz. Hepinize mutlu günler dilerim.

Visual Basic'ten C#'a Geçiş

Merhaba, bunca zamandır Visual Basic kullanan biri olarak .NET'e geçerken Visual Basic'i seçtim. Daha çok Web Programıyla ilgilendiğim için internette yaptığım aramalarda aspx örneklerinin büyük bir çoğunluğunun C# ile hazırlandığını gördüm. Daha sonra bazı makaleler kurcaladım. C#'ın geleceğin dili olduğu fikrine vardım.Elime geçen bir kaç ufak kod üzerinde inceleme yapmaya başladım. Yaptıklarımı sizlerle paylaşmak istedim. Bunu yaparken VB.Net'i orta seviye bilen, C#'ı hiç bilmeyen programcıları düşünerek hazırladım.

İlk dikkatimizi çekecek olan Syntax(Sözdizimi) olacaktır. VB ile en önemli farklılık buradan geliyor. Bu noktada bazı önemli noktaları açıklayacağım.
Yorum satırları: Yorum yazmak için iki farklı metod var. İlki tek satır diğeri blok halinde yazılır.

//Bu bir satırlık yorumdur.

```
/* Bu da blok halinde  
Hazırlanmış bir yorumdur.*/
```

Yukarıda bulunan her iki örneğimiz de derleyici tarafından gözardı edilecektir. Yorum kullanmak kodlarınızın anlaşılabilirliğini arttırdığı için mutlaka kullanılmalıdır. Bunu da belirtmeden geçmeyim.

Değişkenlerimizi VB'ye göre farklı şekillerde tanımlıyoruz. VB'de değişkenlerle aranızda problem yoksa emin olun C#'ta da olmayacaktır.

int a;

ile tek bir değişken tanımlayabiliyoruz. Aynı türde bir kaç değişken tanımlamak için

int a,b,c;

kullanabilirsiniz. Visual Basic'e .Net ile eklenen değişkene isim atanırken değer belirtmeyi de C# da kullanabiliyoruz.

int a=3;

"Merhaba dünya" demek için ilk adımlarımızı atalım. İlk olarak VS.Net ile Merhaba dünya demeyi öğrenelim.VS.Net ile yeni bir proje açın ama bu sefer C# projesi olsun. Formun ortasına kocaman bir tuş koyalım. Çift tıklayıp alttaki koda göre uyarlayın.

```
private void button1_Click(object sender, System.EventArgs e)  
{  
    MessageBox.Show("Selam", "Merhaba Dünya");  
}
```

İlk satırda button1'in click olayı olduğunu belirttik. Tahmin ettiğiniz gibi : bir mesaj kutusu açılacak ve merhaba dünya diyecek. Merhaba Dünya başlığı olacak. İçinde sadece selam yazacak.

Hemen gözümüze çarpanlar Süslü parantez ve noktalı virgül olmuştur.C#'da her olay süslü parantez içinde yer alıyor. Bir if döngüsü veya örnekteki buttonclick olayı. Ve hemen her satırın sonunda noktalı virgül konuyor. Eğer bilgisayarınız kod yazarken size syntax hatası verirse ilk bunları kontrol edin. Alışması gerçekten zor oluyor.

Bir daha ki konuda konuları biraz daha toplayıp bir uygulama ve bir aspx sayfası hazırlayacağız.

Herkesе çalışmalarında başarılar dilerim.

Tuğrul ARAS

C# ile Çok Kanallı(Multithread) Uygulamalar – 3

İş parçacıklarını işlediğimiz yazı dizimizin bu üçüncü makalesinde, iş parçacıklarının birbirlerine karşı öncelik durumlarını incelemeye çalışacağız. İş parçacıkları olarak tanımladığımız metodların çalışma sıralarını, sahip oldukları öneme göre değiştirmek durumunda kalabiliriz. Normal şartlar altında, oluşturduğumuz her bir iş parçacığı nesnesi aynı ve eşit önceliğe sahiptir. Bu öncelik değeri Normal olarak tanımlanmıştır. Bir iş parçacığının önceliğini değiştirmek istediğimizde, Priority özelliğinin değerini değiştiririz. Priority özelliğinin .NET Framework'teki tanımı aşağıdaki gibidir.

public ThreadPriority Priority {get; set;}

Özelliğimiz ThreadPriority numaralandırıcısı (enumerator) tipinden değerler almaktadır. Bu değerler aşağıdaki tabloda verilmiştir.

Öncelik Değeri
Highest
AboveNormal

Normal
BelowNormal
Lowest

Tablo 1. Öncelik(Priority) Değerleri

Programlarımızı yazarken, iş parçacıklarının çalışma şekli verilen öncelik değerlerine göre değişecektir. Elbette tahmin edeceğimiz gibi yüksek öncelik değerlerine sahip olan iş parçacıklarının işaret ettikleri metodlar diğerlerine göre daha sık aralıklarda çağırılacak, dolayısıyla düşük öncelikli iş parçacıklarının referans ettiği metodlar daha geç sonlanacaktır. Şimdi olayı daha iyi canlandırabilmek için aşağıdaki örneğimizi geliştirelim.

Daha önceden söylediğimiz gibi, bir iş parçacığının Priority özelliğine her hangibir değer vermez isek, standart olarak Normal kabul edilir. Buda tüm iş parçacıklarının varsayılan olarak eşit önceliklere sahip olacakları anlamına gelmektedir. Şimdi aşağıdaki formumuzu oluşturalım. Uygulamamız iki iş parçacığına sahip. Bu parçacıkların işaret ettiği metodlardan birisi 1' den 1000' e kadar sayıp bu değerleri bir label kontrolüne yazıyor. Diğer ise 1000' den 1' e kadar sayıp bu değerleri başka bir label kontrolüne yazıyor. Formumuzun görüntüsü aşağıdakine benzer olmalıdır.

Şekil 1. Form Tasarımımız.

Şimdide program kodlarımızı yazalım.

```
/* Bu metod 1' den 1000' e kadar sayar ve değerleri lblSayac1 isimli label kontrolüne yazar.*/
public void Say1()
{
    for(int i=1;i<1000;++i)
    {
        lblSayac1.Text=i.ToString();

        lblSayac1.Refresh(); /* Refresh metodu ile label kontrolünün görüntüsünü tazeleriz. Böylece
herbir i değerinin label kontrolünde görülebilmesini sağlamış oluyoruz. */

        for(int j=1;j<900000000;++j)
        {
            j+=1;
        }
    }
}
/* Bu metod 1000' den 1' e kadar sayar ve değerleri lblSayac2 isimli label kontrolüne yazar.*/
public void Say2()
```

```

{
    for(int i=1000;i>=1;i--)
    {
        lblSayac2.Text=i.ToString();
        lblSayac2.Refresh();
        {
            j+=1;
        }
    }
} /* ThreadStart ve Thread nesnelerimizi tanımlıyoruz. */ ThreadStart ts1;
ThreadStart ts2;
Thread t1;
Thread t2;

private void btnBaslat1_Click(object sender, System.EventArgs e)
{
    /* Metodlarımızı ThreadStart nesneleri ile ilişkilendiriyoruz ve ThreadStart nesnelerimizi oluşturuyoruz.*/
    ts1=new ThreadStart(Say1);
    ts2=new ThreadStart(Say2);

    /* İş parçacıklarımızı, ilgili metodların temsil eden ThreadStart nesnelerimiz ile oluşturuyoruz.*/

    t1=new Thread(ts1);
    t2=new Thread(ts2);
    /* İş parçacıklarımızı çalıştırıyoruz.*/
    t1.Start();
    t2.Start();
    btnBaslat1.Enabled=false;
    btnIptal.Enabled=true;
}

private void btnIptal_Click(object sender, System.EventArgs e)
{
    /* İş parçacıklarımızı iptal ediyoruz. */
    t1.Abort();
    t2.Abort();
    btnBaslat1.Enabled=true;
    btnIptal.Enabled=false;
}

private void btnKapat_Click(object sender, System.EventArgs e)
{
    /* Uygulamayı kapatmak istediğimizde, çalışan iş parçacığı olup olmadığını kontrol ediyoruz. Bunun için iş parçacıklarının IsAlive özelliğinin değerlerine bakıyoruz. Nitekim kullanıcının, herhangi bir iş parçacığı sonlanmadan uygulamayı kapatmasını istemiyoruz. Ya iptal etmeli yada sonlanmalarını beklemeli. İptal ettiğimizde yani Abort metodları çalıştırıldığında hatırlayacağınız gibi, iş parçacıklarının IsAlive değerleri false durumuna düşüyordu, yani iptal olmuş oluyorlardı.*/

    if((!t1.IsAlive) && (!t2.IsAlive))
    {
        Close();
    }
    else
    {
        MessageBox.Show("Hala kapatılamamış iş parçacıkları var. Lütfen bir süre sonra tekrar deneyin.");
    }
}
}

```

Uygulamamızda şu an için bir yenilik yok aslında. Nitekim iş parçacıklarımız için bir öncelik ayarlaması yapmadık. Çünkü size göstermek istediğim bir husus var. Bir iş parçacığı için herhangi bir öncelik ayarı yapmadığımızda bu değer varsayılan olarak Normal dir. Dolayısıyla her iş parçacığı eşit önceliğe sahiptir. Şimdi örneğimizi çalıştıralım ve kafamıza göre bir yerde iptal edelim.

Şekil 2. Öncelik değeri Normal.

Ben 11 ye 984 değerinde işlemi iptal ettim. Tekrar iş parçacıklarını Başlat başlıklı butona tıklayıp çalıştırsak ve yine aynı yerde işlemi iptal edersek, ya aynı sonucu alırız yada yakın değerleri elde ederiz. Nitekim programımızı çalıştırdığımızda arka planda çalışan işletim sistemine ait pek çok iş parçacığıda çalışma sonucunu etkiler. Ancak aşağı yukarı aynı veya yakın değerle ulaşırız. Oysa bu iş parçacıklarının öncelik değerlerini değiştirdiğimizde sonuçların çok daha farklı olabileceğini söyleyebiliriz. Bunu daha iyi anlayabilmek için örneğimizi geliştirelim ve iş parçacıklarının öncelik değerleri ile oynayalım. Formumuzu aşağıdaki gibi tasarlayalım.

Şekil 3. Formumuzun yeni tasarımı.

Artık iş parçacıklarını başlatmadan önce önceliklerini belirleyeceğiz ve sonuçlarını incelemeye çalışacağız. Kodlarımızı şu şekilde değiştirelim. Önemli olan kod satırlarımız, iş parçacıklarının Priority özelliklerinin değiştiği satırlardır.

```
/* Bu metod 1' den 1000' e kadar sayar ve değerleri lblSayac1 isimli label kontrolüne yazar.*/  
public void Say1()  
{
```

```

        for(int i=1;i<1000;++i)
        {
            lblSayac1.Text=i.ToString();
            lblSayac1.Refresh(); /* Refresh metodu ile label kontrolünün görüntüsünü tazeleriz.
Böylece her bir i değerinin label kontrolünde görülebilmesini sağlamış oluyoruz. */

            for(int j=1;j<900000000;++j)
            {
                j+=1;
            }
        }
    }

```

/* Bu metod 1000' den 1' e kadar sayar ve değerleri lblSayac2 isimli label kontrolüne yazar.*/

```

public void Say2()
{
    for(int i=1000;i>=1;i--)
    {
        lblSayac2.Text=i.ToString();
        lblSayac2.Refresh();

        for(int j=1;j<450000000;++j)
        {
            j+=1;
        }
    }
}

```

ThreadPriority tp1;
/* Priority öncelikleri ThreadPriority tipindedirler. */

ThreadPriority tp2; /* OncelikBelirle metodu, kullanıcının TrackBar'da seçtiği değerleri göz önüne alarak, iş parçacıklarının Priority özelliklerini belirlemektedir. */

```

public void OncelikBelirle()
{
    /* Switch ifadelerinde, TrackBar kontrollünün değerine göre , ThreadPriority değerleri belirleniyor. */
    switch(tbOncelik1.Value)
    {
        case 1:
        {
            tp1=ThreadPriority.Lowest; /* En düşük öncelik değeri. */
            break;
        }
        case 2:
        {
            tp1=ThreadPriority.BelowNormal; /* Normalin biraz altı. */
            break;
        }
        case 3:
        {
            tp1=ThreadPriority.Normal; /* Normal öncelik değeri. Varsayılan değer budur.*/
            break;
        }
        case 4:
        {
            tp1=ThreadPriority.AboveNormal; /* Normalin biraz üstü öncelik değeri. */
            break;
        }
    }
}

```

```

        case 5:
        {
            tp1=ThreadPriority.Highest; /* En üst düzey öncelik değeri. */
            break;
        }

    }

    switch(tbOncelik2.Value)
    {
        case 1:
        {
            tp2=ThreadPriority.Lowest; /* En düşük öncelik değeri. */
            break;
        }
        case 2:
        {
            tp2=ThreadPriority.BelowNormal; /* Normalin biraz altı. */
            break;
        }
        case 3:
        {
            tp2=ThreadPriority.Normal; /* Normal öncelik değeri. Varsayılan değer budur.*/
            break;
        }
        case 4:
        {
            tp2=ThreadPriority.AboveNormal; /* Normalin biraz üstü öncelik değeri. */
            break;
        }
        case 5:
        {
            tp2=ThreadPriority.Highest; /* En üst düzey öncelik değeri. */
            break;
        }
    }

    /* İş Parçacıklarımıza öncelik değerleri aktarılıyor.*/

    t1.Priority=tp1;

    t2.Priority=tp2;
}

/* ThreadStart ve Thread nesnelerimizi tanımlıyoruz. */

ThreadStart ts1;

ThreadStart ts2;

Thread t1;

Thread t2;

private void btnBaslat1_Click(object sender, System.EventArgs e)
{
    /* Metodlarımızı ThreadStart nesneleri ile ilişkilendiriyoruz ve ThreadStart nesnelerimizi

```

```
oluşturuyoruz.*/

    ts1=new ThreadStart(Say1);

    ts2=new ThreadStart(Say2);

    /* İş parçacıklarımızı, ilgili metodların temsil eden ThreadStart nesnelerimiz ile
oluşturuyoruz.*/

    t1=new Thread(ts1);

    t2=new Thread(ts2);

    OncelikBelirle(); /* Öncelik ( Priority ) değerleri, iş parçacıkları Start metodu ile başlatılmadan
önce belirlenmelidir. */

    /* İş parçacıklarımızı çalıştırıyoruz.*/

    t1.Start();

    t2.Start();

    btnBaslat1.Enabled=false;

    btnIptal.Enabled=true;

    tbOncelik1.Enabled=false;

    tbOncelik2.Enabled=false;
}

private void btnIptal_Click(object sender, System.EventArgs e)
{

    /* İş parçacıklarımızı iptal ediyoruz. */

    t1.Abort();

    t2.Abort();

    btnBaslat1.Enabled=true;

    btnIptal.Enabled=false;

    tbOncelik1.Enabled=true;

    tbOncelik2.Enabled=true;
}

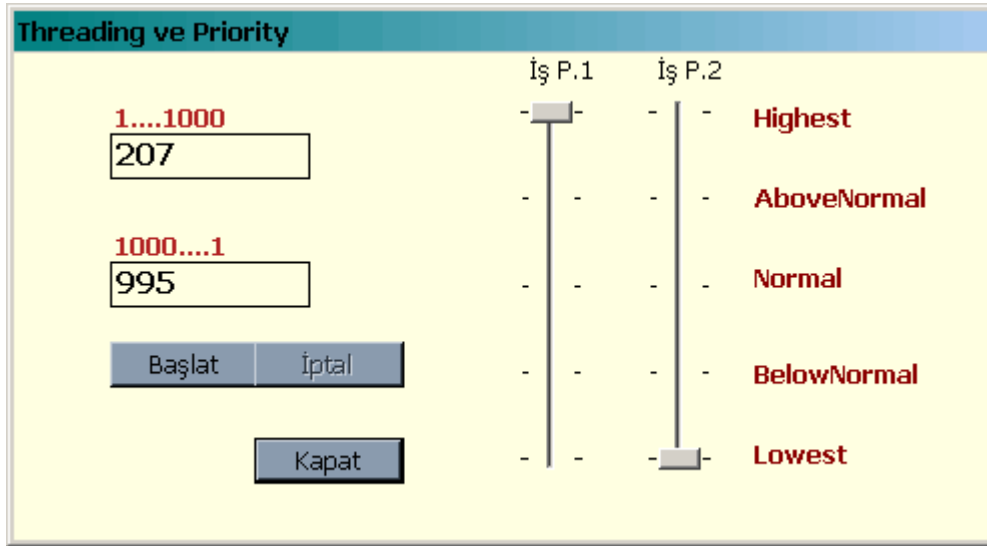
private void btnKapat_Click(object sender, System.EventArgs e)
{

    /* Uygulamayı kapatmak istediğimizde, çalışan iş parçacığı olup olmadığını kontrol ediyoruz.
Bunun için iş parçacıklarının IsAlive özelliğinin değerlerine bakıyoruz. Nitekim kullanıcının,
herhangibir iş parçacığı sonlanmadan uygulamayı kapatmasını istemiyoruz. Ya iptal etmeli yada
```

sonlanmalarını beklemeli. İptal ettiğimizde yani Abort metodları çalıştırıldığında hatırlayacağınız gibi, iş parçacıklarının IsAlive değerleri false durumuna düşüyordu, yani iptal olmuş oluyorlardı.*/

```
if((!t1.IsAlive) && (!t2.IsAlive))
{
    Close();
}
else
{
    MessageBox.Show("Hala kapatılamamış iş parçacıkları var. Lütfen bir süre sonra tekrar deneyin.");
}
```

Şimdi örneğimizi çalıştıralım ve birinci iş parçacığımız için en yüksek öncelik değerini (Highest) ikinci iş parçacığımız içinde en düşük öncelik değerini (Lowest) seçelim. Sonuçlar aşağıdakine benzer olacaktır.



Şekil 4. Önceliklerin etkisi.

Görüldüğü gibi öncelikler iş parçacıklarının çalışmasını oldukça etkilemektedir. Geldik bir makalemizin daha sonuna. Bir sonraki makalemizde iş parçacıkları hakkında ilerlemeye devam edeceğiz. Görüşmek dileğiyle hepinize mutlu günler dilerim.

XOR Operatörü ile Temel Bir Şifreleme Algoritması

Şifreleme günümüzde güvenli iletişim için çok önemli bir konuma gelmiştir, uzun yıllardan beri çok fazla şifreleme algoritması geliştirilmiştir. Bu şifreleme algoritmalarının bir çoğu .NET sınıf kütüphanesinde zaten varsayılan olarak bulunmaktadır, bu yazıda ise kendi şifreleme algoritmalarımızı nasıl oluşturabileceğimiz görmek açısından temel bir şifreleme algoritmasını sizlere göstereceğim.

Bir mesajın yada metnin şifrlenmesi genellikle şifrelenecek mesajın çeşitli operatörler yardımıyla farklı mesajlara dönüştürülmesi ile olmaktadır. Burada bilmemiz gereken nokta şudur : şifrelenecek mesaj ile şifrelenmiş mesajın aynı alfabeden sözcükleri içermesidir. Örneğin ikili(binary) sayılardan oluşturulan bir mesaj şifrelendiği takdirde yine ikili bir sayı olacaktır. Şifreleme yapılırken genellikle anahtar dedğimiz yardımcı bir mesajdan faydalanır. Mesajın şifrlenmesi bu anahtar ile gerçekleşmektedir. Aynı şekilde şifrelenmiş mesajın çözülmesinde de bu anahtar kullanılmaktadır. Şifreleme işlemi ise bir yada daha fazla operatör sayesinde yapılmaktadır. Buradaki operatörler tekil bir operatör olabileceği gibi kullanıcının tanımlayacağı karmaşık değişkenli operatörler de olabilir.

Şifreleme programını yazanlar genellikle şifreyi çözen programıda yazmak zorunda kalırlar. Nede olsa şifreler çözülmek içindir. Çözilemeyen şifreli mesajların pek bir anlamı olmayacağı açıktır. Her ne kadar şifreleme ve şifre çözme programları birbirinin tersi de olsa iki farklı program yazmak yinede zaman kaybettirir. Aynı programın hem şifreleyici hemde şifre çözücü olduğu bir sistem herhalde hepimizin ilgisini çekecektir. Bu yazıda hem şifre çözücü hemde şifreleme işine yarayacak özel bir operatör olan XOR(Bitsel Özel Veya) operatörünü ve bu operatörü kullanarak nasıl şifreleyici ve aynı zamanda şifre çözücü bir programı geliştirebileceğimizi inceleyeceğiz.

XOR(Bitsel Özel Veya) Operatörü

"Özel veya" operatörü iki operandı olan bir operatördür. Özel veya operatörü aldığı operandlarının bütün bitlerini karşılıklı olarak "özel veya(XOR)" işlemine tutar. İsterseniz birçoğumuzun matematik derslerinden hatırlayacağı "özel veya" yani XOR işleminin tanımını görelim. Özel veya operatörü iki operand aldığı için dört farklı durum söz konusudur. Bu durumlar ve sonuçları aşağıdaki tabloda belirtilmiştir.

Operand 1	Operand 2	Sonuç
1	1	0

1	0	1
0	1	1
0	0	0

Tablodan da görüldüğü üzere XOR operatörünün sonucu ancak ve ancak her iki operand da birbirine eşitse 1 değerini almaktadır. Bu sonuç bize şifreleme algoritmasında büyük bir kolaylık sağlayacaktır. XOR operatörü bitsel bir operatör olduğu için her iki operandın da ikili bir sayı olması gerekir. C#'taki veri türleri ile XOR operatörü kullanıldığında veriyi oluşturan her bir değişkenin bütün bitleri karşılıklı olarak XOR işlemine tabi tutulur. Örneğin byte türünden 1 sayısı ile yine byte türünden 2 sayının XOR işlemin sonra hangi değeri oluşturacağını görelim.

Öncelikle 1 ve 2 sayısının bitsel açılımını yazalım :

Not : 1 byte büyüklüğünün 8 bite denk düştüğünü hatırlayalım.

1 --> 0 0 0 0 0 0 0 1

2 --> 0 0 0 0 0 0 1 0

----- 1 ^ 2 (Not : XOR operatörünün simgesi ^ karakteridir.)

3 --> 0 0 0 0 0 0 1 1

Dolayısıyla 1 ve 2 değerini XOR işlemine tabi tutarsak 3 değerini elde ederiz. Bu sonucu programlama yoluyla elde etmek için bir konsol uygulaması açın ve aşağıdaki ifadeyi ekrana yazdırın.

```
Console.WriteLine((1^2));
```

XOR operatörünün diğer önemli bir özelliği ise geri dönüşümlü bir operatör olmasıdır. Yani bir sayıyı "özel veya" işlemine tabi tuttukten sonra sonucu yine aynı sayı ile "özel veya" işlemine tabi tutarsak başlangıçtaki sonucu elde ederiz. Örneğin 3 sayısını 1 ile "özel veya" işlemine tabi tutarsak 2 sayısını, 2 ile "özel veya" işlemine tabi tutarsak bu sefer 1 sayısını elde ederiz. Bu özelliği bir formül ile gösterirsek;

$x = z \wedge b;$

$y = x \wedge b;$

ise

$z = y$ dir.

XOR işleminin bu özelliği yazdığımız programa hem şifre çözücü hemde şifreleyici olma özelliği katacaktır.

Şifreleyici ve Şifre Çözücü Program

Bu bölümde şifre çözücü ve aynı zamanda şifreleyici programı XOR operatörünü kullanarak geliştireceğiz. Program bir dosya şifreleyicisi ve şifre çözücüsü olarak kullanılacaktır. Şifrelenecek dosya bir metin dosyası, çalıştırılabilir exe dosyası olabileceği gibi bir video ve resim dosyasıda olabilir. Çünkü XOR işlemini dosyayı oluşturan byte'lar

düzeyinde gerçekleştireceğiz. Şifreleme işlemi yaparken dosyadaki her bir byte sırayla kullanıcının gireceği bir anahtardan elde edilen sayı ile XOR işlemine tabi tutulacaktır. XOR işlemi sayesinde yazdığımız program aynı zamanda bir şifre çözücü program olarak da çalışacaktır. İlk olarak programımızın en temel halini yazalım ardından programız üzerinde iyileştirme çalışması yapacağız.

Kaynak kodları aşağıda verilen programı yazın ve derleyin.

```
using System;
using System.IO;

namespace XOR
{
    class csharpnedir
    {
        static void Main(string[] args)
        {
            if(args.Length != 2)
            {
                Console.WriteLine("Hatalı kullanım");
                Console.WriteLine("Örnek kullanım : Sifrele xx.text anahtar");
                return ;
            }

            string kaynakDosya = args[0];
            string hedefDosya = args[1];
            string anahtar = "";

            Console.Write("Anahtarı girin :");
            anahtar = Console.ReadLine();

            int XOR = 0;

            for(int i = 0; i
                XOR = XOR + (int)(anahtar[i]);

            FileStream fsKaynakDosya = new FileStream(kaynakDosya, FileMode.Open);
            FileStream fsHedefDosya = new FileStream(hedefDosya, FileMode.CreateNew |
FileMode.CreateNew, FileAccess.Write);

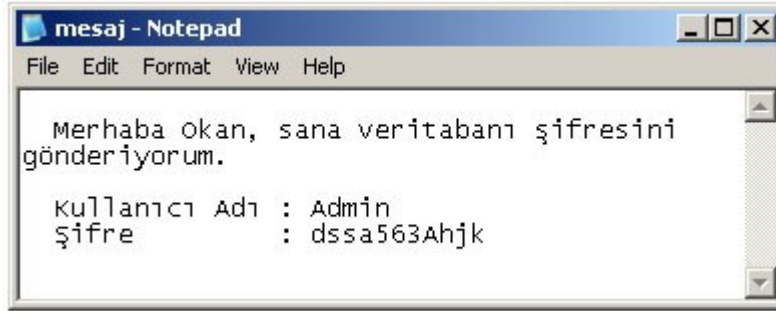
            int kaynakByte; //(3 byte'lık 0 dizisi + kaynakByte)
            byte hedefByte;

            while((kaynakByte = fsKaynakDosya.ReadByte()) != -1)
            {
                hedefByte = (byte)((int)kaynakByte ^ XOR);
                fsHedefDosya.WriteByte(hedefByte);
            }

            fsHedefDosya.Close();
            fsKaynakDosya.Close();
        }
    }
}
```

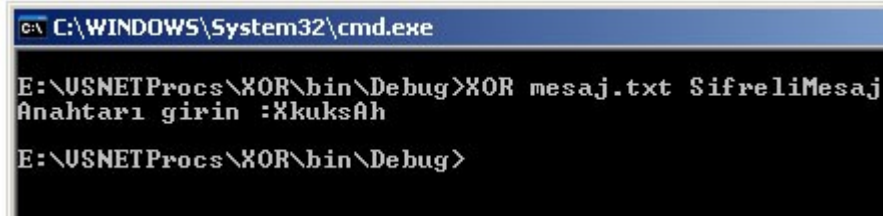
Hemen programın sonucunu görelim :

Aşağıdaki gibi gizlilik derecesi yüksek olan bir metin dosyası oluşturun.



Not : Şifrelenecek dosyanın metin tabanlı olması zorunlu değildir. Çünkü şifreleme işlemini karakter tabanlı değil byte düzeyinde yapmaktayız. Ama sonuçlarını daha iyi görebilmek için örneği metin tabanlı dosya üzerinde gösteriyorum.

Programı aşağıdaki gibi komut satırından çalıştırın.



Programı çalıştırdıktan sonra oluşturulan Sifreli isimli dosyayı Notepad programında görüntülediğimizde aşağıdaki gibi bir ekran ile karşılaşırız.



Dikkat edin, şifreleme işlemini byte düzeyinde yaptığımız için şifreli dosya artık metin dosyası değil binary bir dosya haline gelmiştir.

Şifrelenmiş dosyayı tekrar eski haline getirmek için tek yapmamız gereken komut satırından şifreleme programını diğer bir deyişle şifre çözücü programını çalıştırmamız gerekir. Anahtar olarak ta tabiki şifrelemede kullandığımız anahtar kullanmamız gerekir. Komut satırından aşağıdaki gibi programı çalıştırdığımızda orjinal metin dosyasını elde edebiliriz.

XOR SifreliMesaj OrjinalMesaj.txt
Anahtarı Girin : XkuksAh

Gördüğünüz gibi programımız hem şifreleyici hemde şifre çözücü olarak kullanılabilir.

Sonuçlar

Dikkat ederseniz mesaj dosyasının her byte değeri sabit bir değerle karşılıklı olarak XOR işlemine tabi tutulmuştur. XOR işlemine tabi tutulan değer kullanıcı tarafından girilen anahtardan oluşturulmuştur. Anahtar değerinin her bi karakterinin ASCII karşılığı toplanarak elde edilen değer XOR işleminin sabit operandı olarak ele alınmıştır. Ancak programımızda ufak bir sorun var. Çünkü şifrelemek için girilen anahtar değerini oluşturan karakterlerin hepsini içerecek şekilde oluşturulan bütün kombinasyonlar şifrelenmiş dosyayı çözecektir. Örneğin şifrelemek için kullanılan anahtar değerinin "AxyHMnK2" olduğunu düşünelim. Bu durumda "xynAHMNK2" ve "2MnKHxA" gibi kombinasyonlar dosyanın çözülmesini sağlayacaktır.

Yukarıda bahsi geçen kısıtı engellemek için XOR işlemine tabi tutulacak operandı anahtar değerden elde edilen farklı bir yöntem kullanılır. Bu operandı aşağıdaki gibi yeniden elde edebiliriz.

```
int XOR = 0;

for(int i = 0; i
    XOR = XOR + (int)(anahtar[i] * 10);
```

Yukarıdaki düzenlmeye rağmen şifreyi çözecek anahtar tek değildir. Çünkü farklı karakter kombinasyonlarının toplamı çok düşük bir ihtimalde olsa orjinal XOR değerine eşit olabilir. Ancak bu durum şifreleme tekniğinin güvenilirliğini azaltmaz. Çünkü orjinal XOR değerinin tahmin etme olasılığı çok azdır.

Gelelim diğer bir kısıta : Dikkat ederseniz şifreleme yaparken dosyadaki her bir byte değerini sabit bir değerle XOR işlemine tabi tuttuk. Bir byte değişkenin sınırları 0- 255 arası olduğu için şifreleme programını çözmek için en fazla 256 ihtimal vardır. Tabi burada anahtar değerden XOR işlemine tabi tutulacak değer nasıl elde edildiğinin bilindiği varsayılmaktadır. Eğer bu yöntem bilinmiyorsa şifrenin çözülme olasılığı neredeyse imkansızdır. XOR operandının elde edilme yönteminin bilindiği varsayımı altında 256 sayısını yani şifrenin çözülme olasılığını azaltmak için yapmamız gereken XOR işlemini 1 byte'lık bloklar yerine daha büyük bloklar ile yapmaktır. Örneğin XOR işlemini 4 byte'lık veri blokları ile yaptığımızda XOR işleminin operandı 4.294.967.296 ihtimalden birisidir. Eğer XOR işlemine sokulan veri bloğu artırılırsa operandın alabileceği değerler üstel bir biçimde artacaktır. Bu arada XOR işlemine sokulacak veri bloklarının sayısı arttıkça xor işlemindeki operandın değerini belirlemek için farklı yöntemler kullanılmalıdır. Çünkü eğer aşağıdaki yöntemde elde edilen XOR operandını kullanırsak 1 byte yada 4 byte'lık verilerle çalışmanın çok önemli bir farkı olmayacaktır. (Burada fark, girilen anahtara göre belirlenir. Örneğin oluşturulan xor operandı 256 değerinden küçük ise hiç bir fark meydana gelmeyecektir.)

```
int XOR = 0;

for(int i = 0; i
    XOR = XOR + (int)(anahtar[i]);
```

Bu yöntemle geliştirilecek bir şifreleme programını daha etkili hale getirmek için bir yöntem daha vardır. Programı incellerseniz her bir byte bloğunu sabit bir değerle xor işlemine soktuk. Bu aslında biraz risklidir. Zira büyük bir şifreli metnin çok küçük bölümünün çözülmesi tamamının çözülmesi anlamına gelir. Bu yüzden her bir byte bloğunu farklı bir değerle xor işlemine tabi tutarsak şifreli metnin her bir şifreli bloğu bir diğerinden bağımsız hale gelir. Yani çözülmüş bir şifreli blok diğer bloğun çözülmesine kesin bir bilgi vermez. Dolayısıyla şifre kırıcı programların lineer bir yöntem izlemesi

engellenmiş olur.

Bu tür bir şifreleme yönteminin devlet düzeyinde güvenli olması gereken mesajlarda kullanılması uygun olmasada mesajların başkaları tarafından açıkca görülmeden haberleşme sistemlerinden geçirilmesi için uygun bir yöntemdir. Elbetteki daha basit yöntemlerle de bu işlemi gerçekleştirebiliriz ancak bu yöntemin en önemli özelliği hem şifreleme hemde şifre çözücü olarak kullanılabilmesidir.

Bu yazının kendi şifreleme algortimalarınızı oluşturmada size yol gösterebileceğini umuyor iyi çalışmalar diliyorum.

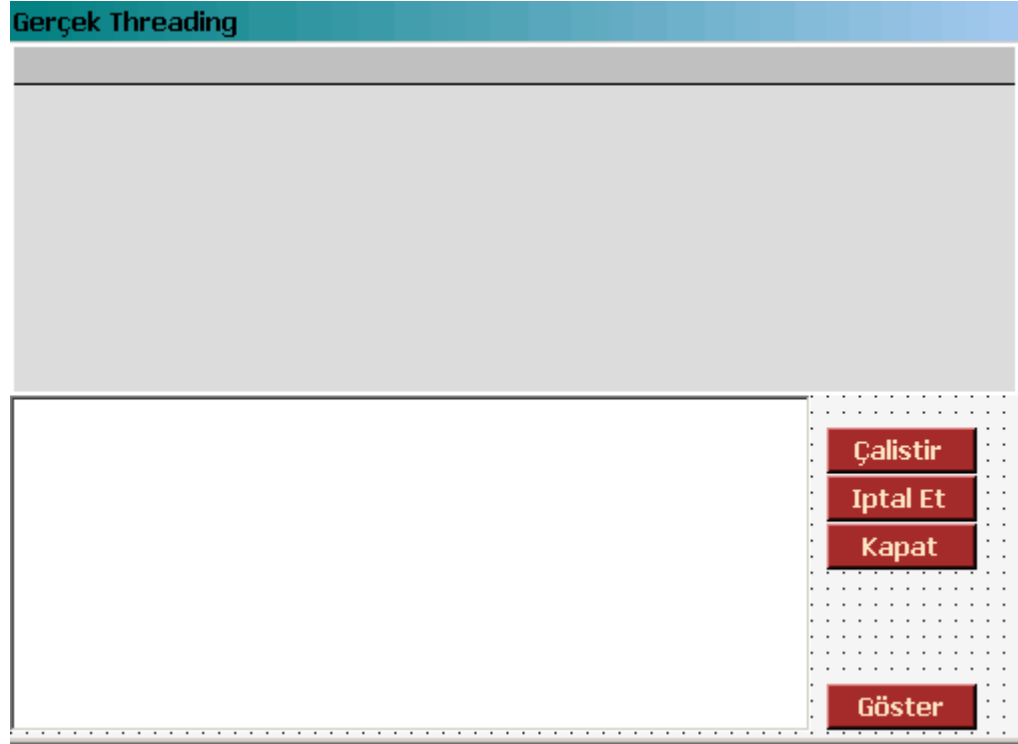
C# ile Çok Kanallı(Multithread) Uygulamalar – 4

Bundan önceki üç makalemizde iş parçacıkları hakkında bilgiler vermeye çalıştım, bu makalemde ise işimize yarayacak tarzda bir uygulama geliştirecek ve bilgilerimizi pekiştireceğiz. Bir iş parçacığının belkide en çok işe yarayacağı yerlerden birisi veritabanı uygulamalarıdır. Bazen programımız çok uzun bir sonuç kümesi döndürecek sorgulara veya uzun sürecek güncelleme ifadeleri içeren sql cümlelerine sahip olabilir. Böyle bir durumda programın diğer öğeleri ile olan aktivitemizi devam ettirebilmek isteyebiliriz. Ya da aynı anda bir den fazla iş parçacığında, birden fazla veritabanı işlemini yaptırarak bu işlemlerin tamamının daha kısa sürelerde bitmesini sağlayabiliriz. İşte bu gibi nedenleri göz önüne alarak bu gün birlikte basit ama faydalı olacağına inandığım bir uygulama geliştireceğiz.

Olayı iyi anlayabilmek için öncelikle bir milat koymamız gerekli. İş parçacığından önceki durum ve sonraki durum şeklinde. Bu nedenle uygulamamızı önce iş parçacığı kullanmadan oluşturacağız. Sonrada iş parçacığı ile. Şimdi programımızdan kısaca bahsedelim. Uygulamamız aşağıdaki sql sorgusunu çalıştırıp, bellekteki bir DataSet nesnesinin referans ettiği bölgeyi, sorgu sonucu dönen veri kümesi ile dolduracak.

```
SELECT Products.* From [Order Details] Cross Join Products
```

Bu sorgu çalıştırıldığında, Sql sunucusunda yer alan Northwind veritabanı üzerinden, **165936** satırlık veri kümesi döndürür. Elbette normalde böyle bir işlemi istemci makinenin belleğine yığmamız anlamsız. Ancak sunucu üzerinde çalışan ve özellikle raporlama amacı ile kullanılan sorguların bu tip sonuçlar döndürmeside olasıdır. Şimdi bu sorguyu çalıştırıp sonuçları bir DataSet'e alan ve bu veri kümesini bir DataGrid kontrolü içinde gösteren bir uygulama geliştirelim. Öncelikle aşağıdaki formumuzu tasarlayalım.



Şekil 1. Form Tasarımımız.

Şimdide kodlarımızı yazalım.

```
DataSet ds;

public void Bagla()
{
    dataGrid1.DataSource=ds.Tables[0];
}

public void Doldur()
{
    SqlConnection conNorthwind=new SqlConnection("data source=localhost;initial
catalog=Northwind;integrated security=sspi");

    conNorthwind.Open();

    SqlDataAdapter daNorthwind=new SqlDataAdapter("SELECT Products.* From [Order
Details] Cross Join Products",conNorthwind);

    ds=new DataSet();

    daNorthwind.Fill(ds);

    conNorthwind.Close();

    MessageBox.Show("DataTable dolduruldu...");
}
```

```

}

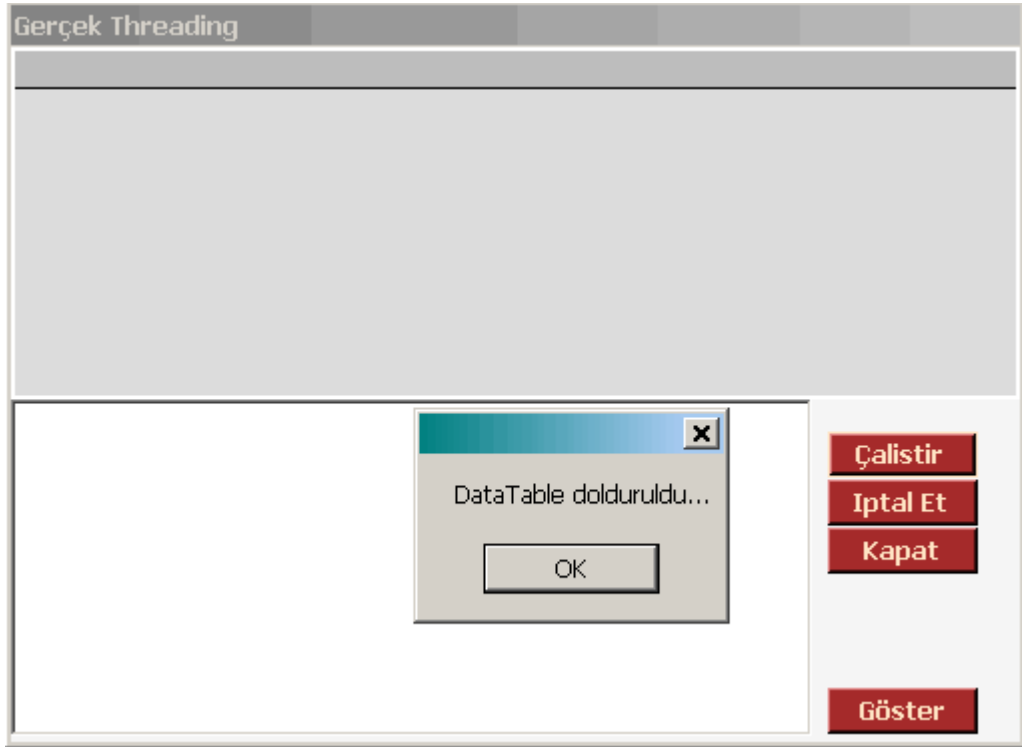
private void btnKapat_Click(object sender, System.EventArgs e)
{
    Close();
}

private void btnCalistir_Click(object sender, System.EventArgs e)
{
    Doldur();
}

private void btnGoster_Click(object sender, System.EventArgs e)
{
    Bagla();
}

```

Yazdığımız kodlar gayet basit. Sorgumuz bir SqlDataAdapter nesnesi ile, SqlConnection'ımız kullanılarak çalıştırılıyor ve daha sonra elde edilen veri kümesi DataSet'e aktarılıyor. Şimdi uygulamamızı bu haliyle çalıştıralım ve sorgumuzu Çalıştır başlıklı buton ile çalıştırdıktan sonra, textBox kontrolüne mouse ile tıklayıp bir şeyler yazmaya çalışalım.



Şekil 2. İş parçacığı olmadan programın çalışması.

Görüldüğü gibi sorgu sonucu elde edilen veri kümesi DataSet'e doldurulana kadar TextBox kontrolüne bir şey yazamadık. Çünkü işlemcimiz satır kodlarını işletmek ile meşguldü ve bizim TextBox kontrolümüze olan tıklamamızı ele almadı. Demekki buradaki sorgumuzu bir iş parçacığı içinde tanımlamalıyız. Nitekim programımız donmasın ve başka işlemleride yapabilelim. Örneğin TextBox kontrolüne bir şeyler yazabilelim (bu noktada pek çok şey söylenebilir. Örneğin başka bir tablonun güncellenmesi gibi). Bu durumda yapmamız gereken kodlamayı inanıyorumki önceki

makalelerden edindiğiniz bilgiler ile biliyorsunuzdur. Bu nedenle kodlarımızı detaylı bir şekilde açıklamadım. Şimdi gelin yeni kodlarımızı yazalım.

```
DataSet ds;

public void Bagla()
{
    if(!t1.IsAlive)
    {
        dataGrid1.DataSource=ds.Tables[0];
    }
}

public void Doldur()
{
    SqlConnection conNorthwind=new SqlConnection("data source=localhost;initial
catalog=Northwind;integrated security=sspi");

    conNorthwind.Open();

    SqlDataAdapter daNorthwind=new SqlDataAdapter("SELECT Products.* From
[Order Details] Cross Join Products",conNorthwind);

    ds=new DataSet();

    daNorthwind.Fill(ds);

    conNorthwind.Close();

    MessageBox.Show("DataTable dolduruldu...");
}

ThreadStart ts1;

Thread t1;

private void btnKapat_Click(object sender, System.EventArgs e)
{
    if(!t1.IsAlive)
    {
        Close();
    }

    else
    {
        MessageBox.Show("Is parçacigi henüz sonlandırılmadi...Daha sonra
tekrar deneyin.");
    }
}

private void btnCalistir_Click(object sender, System.EventArgs e)
{
}
```



```

        ts1=new ThreadStart(Doldur);

        t1=new Thread(ts1);

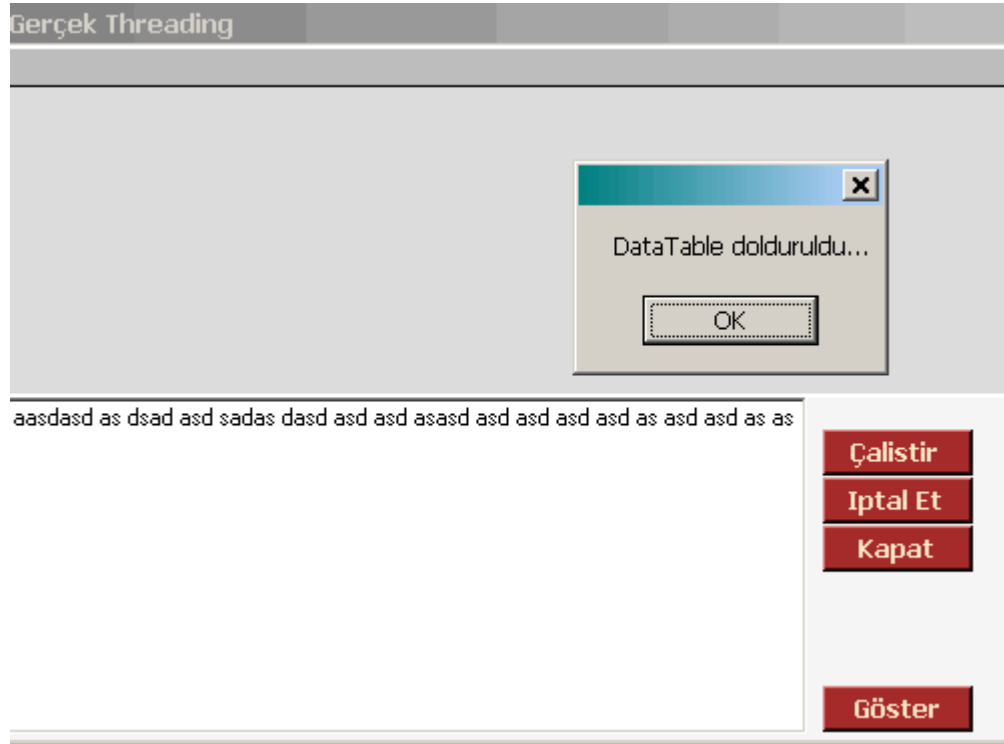
        t1.Start();
    }

    private void btnIptalEt_Click(object sender, System.EventArgs e)
    {
        t1.Abort();
    }

    private void btnGoster_Click(object sender, System.EventArgs e)
    {
        Bagla();
    }

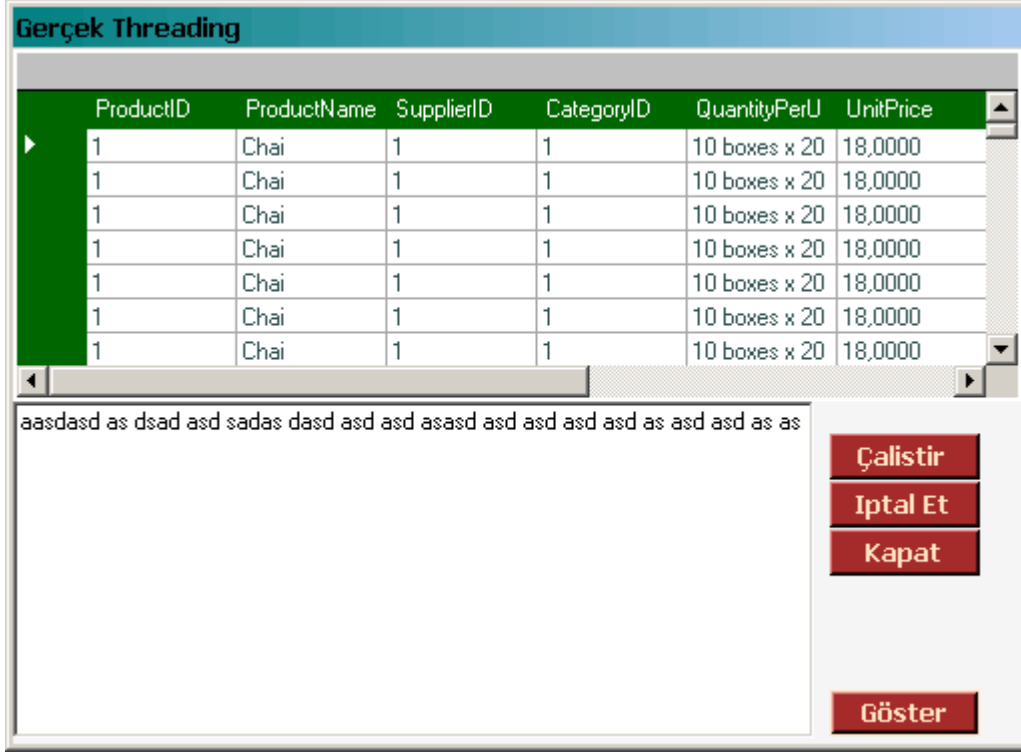
```

Şimdi programımızı çalıştıralım.



Şekil 3. İş Parçacığının sonucu.

Görüldüğü gibi bu yoğun sorgu çalışırken TextBox kontrolüne bir takım yazılar yazabildik. Üstelik programın çalışması hiç kesilmeden. Şimdi Göster başlıklı butona tıkladığımızda veri kümesinin DataGrid kontrolüne alındığını görürüz.



Şekil 4. Programın Çalışmasının Sonucu.

Geldik bir makalemizin daha sonuna. İlerleyen makalelerimizde Thred'leri daha derinlemesine incelemeye devam edeceğiz. Hepinize mutlu günler dilerim.

Arayüz(Interface) Kullanımına Giriş

Bugünkü makalemizde, nesneye dayalı programlamanın önemli kavramlarından birisi olan arayüzleri incelemeye çalışacağız. Öncelikle, arayüz'ün tanımını yapalım.

Bir arayüz, başka sınıflar için bir rehberdir. Bu kısa tanımın arkasında, deryalar gibi bir kavram denizi olduğunu söylemekte yarar buluyorum. Arayüzün ne olduğunu tam olarak anlayabilmek için belkide asıl kullanım amacına bakmamız gerekmektedir.

C++ programlama dilinde, sınıflar arasında çok kalıtsımlılık söz konusu idi. Yani bir sınıf, kalıtsımsal olarak, birden fazla sınıftan türetililebiliyordu . Ancak bu teknik bir süre sonra kodların dahada karmaşılaşmasına ve anlaşılabilirliğin azalmasına neden oluyordu. Bu sebeben ötürü değerli Microsoft mimarları, C# dilinde, bir sınıfın sadece tek bir sınıfı kalıtsımsal olarak alabileceği kısıtılmasını getirdiler. **Çok kalıtsımlık görevini ise anlaşılması daha kolay arayüzlere bıraktılar.** İşte arayüzleri kullanmamızın en büyük nedenlerinden birisi budur.

Diğer yandan, uygulamalarımızın geleceği açısından da arayüzlerin çok kullanışlı olabileceğini söylememiz gerekiyor. Düşününkü, bir ekip tarafından yazılan ve geliştirilen bir uygulamada görevlisiniz. Kullandığınız nesnelerin, türetildiği sınıflar zaman içerisinde, gelişen yeniliklere adapte olabilmek amacıyla, sayısız yeni metoda, özelliğe vb.. sahip olduklarını farzedin. Bir süre sonra, nesnelerin türetildiği sınıflar içerisinde yer alan kavram kargaşasını, "bu neyi yapıyordu?, kime yapıyordu? , ne için yapıyordu?" gibi soruların ne kadar çok sorulduğunu düşünün. Oysa uygulamanızdaki sınıfların izleyeceği yolu gösteren rehber(ler) olsa fena mı olurdu? İşte size arayüzler. Bir arayüz oluşturun ve bu arayüzü uygulayan sınıfların hangi metodları, özellikleri vb kullanması gerektiğine karar verin. Programın gelişmesini gerekiyor? Yeni niteliklere mi ihtiyacın var? İster

kullanılan arayüzleri, birbirlerinden kalıtsal olarak türetin, ister yeni arayüzler tasarlayın. Tek yapacağınız sınıfların hangi arayüzlerini kullanacağını belirtmek olacaktır.

Bu açıklamalar ışığında bir arayüz nasıl tanımlanır ve hangi üyelere sahiptir bundan bahsedelim. Bir arayüz tanımlanması aşağıdaki gibi yapılır. Yazılan kod bloğunun bir arayüz olduğunu **Interface** anahtar sözcüğü belirtmektedir. Arayüz isminin başında I harfi kullanıldığına dikkat edin. Bu kullanılan sınıfın bir arayüz olduğunu anlamamıza yarayan bir isim kullanma tekniğidir. Bu sayede, sınıfların kalıtsal olarak aldığı elemanların arayüz olup olmadığını daha kolayca anlayabiliriz.

```
public interface IArayuz
{
}
}
```

Tanımlama görüldüğü gibi son derece basit. Şimdi arayüzlerin üyelerine bir göz atalım. Arayüzler, sadece aşağıdaki üyelere sahip olabilirler:

Arayüz Üyeleri
A. Özellikler (properties)
B. Metodlar (methods)
C. Olaylar (events)
D. İndeksleyiciler (indexers)

Tablo 1. Arayüzlerin sahip olabileceği üyeler

Diğer yandan, arayüzler içerisinde aşağıdaki üyeler kesinlikle kullanılamazlar:

Arayüzlerde Kullanılamayan Üyeler
i. Yapıcılar (constructors)
ii. Yokediciler (destructors)
iii. Alanlar (fields)

Tablo 2. Arayüzlerde kullanılamayan üyeler.

Arayüzler *Tablo1* deki üyelere sahip olabilirler. Peki bu üyeler nasıl tanımlanır. Herşeyden önce arayüzler ile ilgili en önemli kural onun bir rehber olmasıdır. Yani arayüzler sadece, kendisini rehber alan sınıfların kullanacağı üyeleri tanımlarlar. Herhangi bir kod satırı içermezler. Sadece özelliğin, metodun, olayın veya indeksleyicinin tanımı vardır. Onların kolay okunabilir olmalarını sağlayan ve çoklu kalıtım için tercih edilmelerine neden olan sebepte budur. Örneğin;

```
public interface IArayuz
{
    /* double tipte bir özellik tanımı. get ve set anahtar
    sözcüklerinin herhangi bir blok {} içermediğine dikkat edin. */

    double isim
    {
        get;
    }
}
```

```

        set;
    }

    /* Yanlız okunabilir (ReadOnly) string tipte bir özellik tanımı. */
    string soyisim
    {
        get ;
    }

    /* integer değer döndüren ve ili integer parametre alan bir metod
    tanımı. Metod tanımlarındada metodun dönüş tipi, parametreleri, ismi
    dışında herhangi bir kod satırı olmadığına dikkat edin. */

    int topla(int a, int b);

    /* Dönüş değeri olmayan ve herhangi bir parametre almayan bir metod
    tanımı. */

    void yaz();

    /* Bir indeksleyici tanımı */
    string this [ int index]
    {
        get;
        set;
    }
}

```

Görüldüğü gibi sadece tanımlamalar mevcut. Herhangi bir kod satırı mevcut değil. Bir arayüz tasarlarırken uymamız gereken bir takım önemli kurallar vardır. Bu kurallar aşağıdaki tabloda kısaca listelenmiştir.

- 1 **Bir arayüz'ün tüm üyeleri public kabul edilir.** Private, Protected gibi belirteçler kullanamayız. Bunu yaptığımız takdirde örneğin bir elemanı private tanımladığımız takdirde, derleme zamanında şu hatayı alırız. **"The modifier 'private' is not valid for this item"**
- 2 **Diğer yandan bir metodu public olarakta tanımlayamayız.** Çünkü zaten varsayılan olarak bütün üyeler public tanımlanmış kabul edilir. Bir metodu public tanımladığımızda yine derleme zamanında şu hatayı alırız. **"The modifier 'public' is not valid for this item"**
- 3 **Bir arayüz, bir yapı(struct)'dan veya bir sınıf(class)'tan kalıtımla türetilemez.** Ancak, bir arayüzü başka bir arayüzden veya arayüzlerden kalıtımsal olarak türetebiliriz.
- 4 **Arayüz elemanlarını static olarak tanımlayamayız.**
- 5 **Arayüzlerin uygulandığı sınıflar, arayüzde tanımlanan bütün üyeleri kullanmak zorundadır.**

Tablo 3. Uyulması gereken kurallar.

Şimdi bu kadar açıklamadan sonra konuyu daha iyi anlayabilmek için basit ve açıklayıcı bir örnek geliştirelim. Önce arayüzümüzü tasarlayalım:

```
public interface IArayuz
{
    void EkranaYaz();
    int Yas
    {
        get;
        set;
    }

    string isim
    {
        get;
        set;
    }
}
```

Şimdide bu arayüzü kullanacak sınıfımızı tasarlayalım.

```
public class Kisiler:IArayuz
{
}
```

Şimdi bu anda uygulamayı derlersek, IArayuz'ündeki elemanları sınıfımız içinde kullanmadığımızdan dolayı aşağıdaki derleme zamanı hatalarını alırız.

- *Interfaces1.Kisiler' does not implement interface member 'Interfaces1.IArayuz.EkranaYaz()'*
- *Interfaces1.Kisiler' does not implement interface member 'Interfaces1.IArayuz.isim'*
- *Interfaces1.Kisiler' does not implement interface member 'Interfaces1.IArayuz.Yas'*

Görüldüğü gibi kullanmadığımız tüm arayüz üyeleri için bir hata mesajı oluştu. Bu noktada şunu tekrar hatırlatmak istiyorum,

Arayüzlerin uygulandığı sınıflar, arayüzde(lerde) tanımlanan tüm üyeleri kullanmak, yani kodlamak zorundadır.

Şimdi sınıfımızı düzgün bir şekilde geliştirelim:

```
public class Kisiler:IArayuz /* Sınıfın kullanacağı arayüz burada belirtiliyor.*/
{
    private int y;
    private string i;

    /* Bir sınıfa bir arayüz uygulamamız, bu sınıfa başka üyeler eklememizi engellemez. Burada örneğin sınıfın yapıcı metodlarında düzenledik. */

    public Kisiler()
```

```

        {
            y=18;
            i="Yok";
        }

        /* Dikkat ederseniz özelliğin herşeyi, arayüzdeki ile aynı
        olmalıdır. Veri tipi, ismi vb... Bu tüm diğer arayüz üyelerinin, sınıf
        içerisinde uygulanmasında da geçerlidir. */

        public Kisiler(string ad,int yas)
        {
            y=yas;
            i=ad;
        }

        public int Yas
        {
            get
            {
                return y;
            }

            set
            {
                y=value;
            }
        }

        public string Isim
        {
            get
            {
                return i;
            }

            set
            {
                i=value;
            }
        }

        public void EkranaYaz()
        {
            Console.WriteLine("Adım:"+i);
            Console.WriteLine("Yaşım:"+y);
        }
    }

```

Şimdi oluşturduğumuz bu sınıfı nasıl kullanacağımıza bakalım.

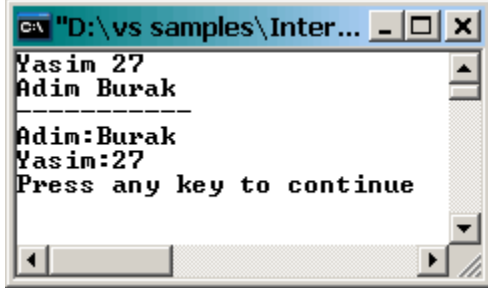
```

class Arayuz_Deneme
{
    {
        Kisiler kisi=new Kisiler("Burak",27);
        Console.WriteLine("Yaşım "+kisi.Yas.ToString());
    }
}

```

```
        Console.WriteLine("Adım "+kisi.Isim);  
        Console.WriteLine("-----");  
        kisi.EkranaYaz();  
    }  
}
```

Uygulamamızı çalıştırdığımızda aşağıdaki sonucu elde ederiz.



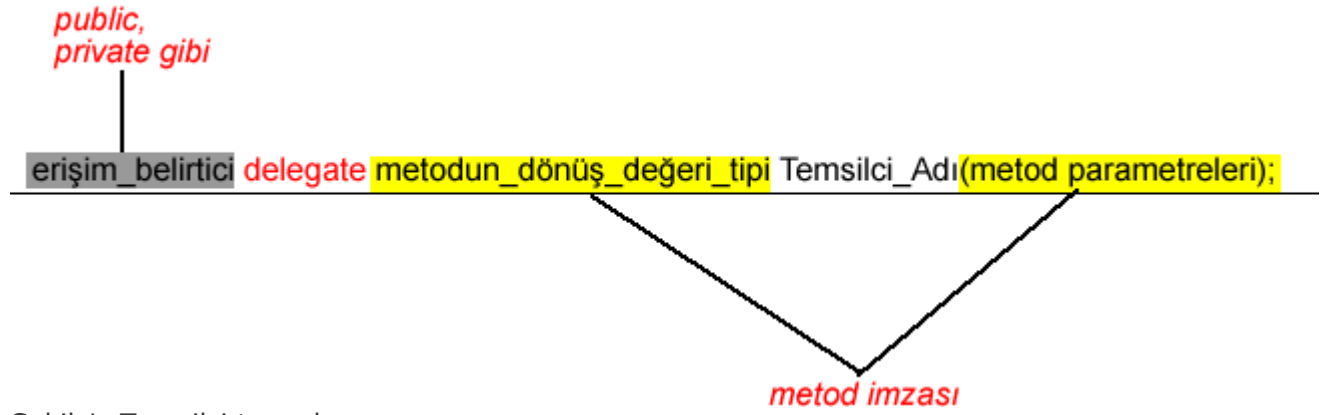
Şekil 1. Uygulamanın Çalışması Sonucu.

Bu makalemizde arayüzlere kısa bir giriş yaptık. Bir sonraki makalemizde ise, bir sınıfa birden fazla arayüzün nasıl uygulanacağını inceleyeceğiz. Hepinize mutlu günler dilerim.

Temsilci(Delegate) Kavramına Giriş

Bugünkü makalemizde, C# programlama dilinde ileri seviye kavramlardan biri olan Temsilcileri(delegates) incelemeye başlayacağız. Temsilciler ileri seviye bir kavram olmasına rağmen, her seviyeden C# programcısının bilmesi gereken unsurlardandır. Uygulamalarımızı temsilciler olmadan da geliştirebiliriz. Ancak bu durumda, yapamıyacaklarımız, yapabileceklerimizin önüne geçecektir. Diğer yandan temsilcilerin kullanımını gördükçe bize getireceği avantajları daha iyi anlayacağımız kanısındayım. Bu makalemizde temsilcileri en basit haliyle anlamaya çalışacağız.

Temsilci (delegate), program içerisinde bir veya daha fazla metodu gösteren(işaret eden), referans türünden bir nesnedir. Programlarımızda temsilciler kullanmak istediğimizde, öncelikle bu temsilcinin tanımını yaparız. Temsilci tanımları, arayüzlerdeki metod tanımlamaları ile neredeyse aynıdır. Tek fark delegate anahtar sözcüğünün yer almasıdır. Bununla birlikte, bir temsilci tanımlandığında, aslında işaret edebileceği metod(ların) imzalarının da belirlemiş olur. Dolayısıyla, bir temsilciyi sadece tanımladığı metod imzasına uygun metodlar için kullanılabileceğimizi söyleyebiliriz. Temsilci tanımları tasarım zamanında yapılır. Bir temsilciyi, bir metodu işaret etmesi için kullanmak istediğimizde ise, çalışma zamanında onu new yapılandırıcısı ile oluşturur ve işaret etmesini istediğimiz metodu ona parametre olarak veririz. Bir temsilci tanımı genel haliyle, aşağıdaki şekilde gibidir.



Şekil 1. Temsilci tanımlaması.

Şekildende görüldüğü gibi, temsilciler aslında bir metod tanımlarlar fakat bunu uygulamazlar. İşte bu özellikleri ile arayüzlerdeki metod tanımlamalarına benzerler. Uygulamalarımızda, temsilci nesneleri ile göstermek yani işaret etmek istediğimiz metodlar bu imzaya sahip olmalıdır. Bildiğiniz gibi metod imzaları, metodun geri dönüş tipi ve aldığı parametreler ile belirlenmektedir.

Bir temsilcinin tanımlanması, onu kullanmak için yeterli değildir elbette. Herşeyden önce bir amacımız olmalıdır. Bir temsilciyi çalışma zamanında oluşturabiliriz ve kullanabiliriz. Bir temsilci sadece bir tek metodu işaret edebileceği gibi, birden fazla metod için tanımlanmış ve oluşturulmuş temsilcileride kullanabiliriz. Diğer yandan, tek bir temsilcide birden fazla temsilciyi toplayarak bu temsilcilerin işaret ettiği, tüm metodları tek bir seferde çalıştırma lüksünede sahibizdir. Ancak temsilciler gerçek anlamda iki amaçla kullanılırlar. Bunlardan birincisi olaylardır(events). Diğer yandan, bugünkü makalemizde işleyeceğimiz gibi, bir metodun çalışma zamanında, hangi metodların çalıştırılacağına karar vermesi gerektiği durumlarda kullanırız. Elbette bahsetmiş olduğumuz bu amacı, herhangi bir temsilye ihtiyaç duymadan da gerçekleştirebiliriz. Ancak temsilcileri kullanmadığımızda, bize sağladığı üstün programlama tekniği, kullanım kolaylığı ve artan verimliliğide göz ardı etmiş oluruz.

Şimdi dilerseniz bahsetmiş olduğumuz bu amaçla ilgili bir örnek verelim ve konuyu daha iyi kavramaya çalışalım. Örneğin, personelimizin yapmış olduğu satış tutarlarına göre, prim hesabı yapan ve ilgili yerlere bu değişiklikleri yazan bir projemiz olsun. Burada primlerin hesaplanması için değişik katsayılar, yapılan satışın tutarına göre belirlenmiş olabilir. Örneğin bu oranlar düşük, orta ve yüksek olarak tanımlanmış olsun. Personel hangi gruba giriyorsa, metodumuz ona uygun metodu çağırırsın. İşte bu durumda karar verici metodumuz, çalıştırabileceği metodları temsil eden temsilci nesnelerini parametre olarak alır. Yani, çalışma zamanında ilgili metodlar için temsilci nesneleri oluşturulur ve karar verici metoda , hangi metod çalıştırılacak ise onun temsilcisi gönderilir. Böylece uygulamamız çalıştığında, tek yapmamız gereken hangi metodun çalıştırılması isteniyorsa, bu metoda ilişkin temsilcinin, karar verici metoda gönderilmesi olacaktır.

Oldukça karışık görünüyor. Ancak örnekleri yazdıkça daha iyi kavrayacağınıza inanıyorum. Şimdiki örneğimizde, temsilcilerin tasarım zamanında nasıl tanımlandığını, çalışma zamanında nasıl oluşturulduklarını ve karar verici bir metod için temsilcilerin nasıl kullanılacağını incelemeye çalışacağız.

```
using System;

namespace Delegates1
{
    public class Calistir
```



```
{  
    public static int a;  
    public delegate void temsilci(int deger); /* Temsilci tanımlamamızı yapıyoruz. Aynı  
zamanda temsilcimiz , değer döndürmeyen ve integer tipte tek bir parametre alan bir  
metod tanımlıyor. Temsilcimizin adı ise temsilci.*/
```

* Şimdi bu temsilciyi kullananak bir metod yazıyoruz. İşte karar verici metodumuz budur. Dikkat ederseniz metodumuz parametre olarak, temsilci nesnemiz tipinden bir temsilci(Delegate) alıyor. Daha sonra metod bloğu içinde, parametre olarak geçirilen bu temsilcinin işaret ettiği metod çağırılıyor ve bu metoda parametre olarak integer tipte bir değer geçiriliyor. Kısaca, metod içinden, temsilcinin işaret ettiği metod çağırılıyor. Burada, temsilci tanımına uygun olan metodun çağırılması garanti altına alınmıştır. Yani, programın çalışması sırasında, new yapılandırıcısı kulllanarak oluşturacağımız bir temsilci(delegate), kendi metod tanımı ile uyuşmayan bir metod için yaratılmaya çalışıldığında bir derleyici hatası alacağız. Dolayısıyla bu, temsilcilerin yüksek güvenlikli işaretçiler olmasını sağlar. Bu , temsilcileri, C++ dilindeki benzeri olan işaretçilerden ayıran en önemli özelliktir. */

```
        public void Metod1(Calistir.temsilci t)  
        {  
            t(a);  
        }  
    }
```

```
class Class1  
{
```

/* İkiKat ve UcKat isimli metodlarımız, temsilcimizin programın çalışması sırasında işaret etmesini istediğimiz metodlar. Bu nedenle imzaları, temsilci tanımımızdaki metod imzası ile aynıdır. */

```
        public static void İkiKat(int sayi)  
        {  
            sayi=sayi*2;  
            Console.WriteLine("İkiKat isimli metodun temsilcisi tarafından  
çagirildi."+sayi.ToString());  
        }
```

```
        public static void UcKat(int sayi)  
        {  
            sayi=sayi*3;  
            Console.WriteLine("UcKat isimli metodun temsilcisi tarafından  
çagirildi."+sayi.ToString());  
        }
```

```
        static void Main(string[] args)  
        {
```

/* Temsilci nesnelerimiz ilgili metodlar için oluşturuluyor. Burada, new yapılandırıcısı ile oluşturulan temsilci nesneleri parametre olarak, işaret edecekleri metodun ismini alıyorlar. Bu noktadan itibaren t1 isimli delegate nesnemiz İkiKat isimli metodu, t2 isimli delegate nesnemizde UcKat isimli metodu işaret ediceklerdir. */

```
            Calistir.temsilci t1=new Delegates1.Calistir.temsilci(İkiKat);
```

```
Calistir.temcilci t2=new Delegates1.Calistir.temcilci(UcKat);
```

```
Console.WriteLine("1 ile 20 arası değer girin");
```

```
Calistir.a=System.Convert.ToInt32(Console.ReadLine());
```

```
Calistir c= new Calistir();
```

/* Kullanıcının Console penceresinden girdiği değer göre, Calistir sınıfının a isimli integer tipteki değerini 10 ile karşılaştırılıyor. 10 dan büyükse, karar verici metodumuza t1 temsilcisi gönderiliyor. Bu durumda Metod1 isimli karar verici metodumuz, kendi kod bloğu içinde t1 delegate nesnesinin temsil ettiği İkiKat metodunu, Calistir.a değişkeni ile çağırıyor. Aynı işlem tarzı t2 delegate nesnesi içinde geçerli.*/

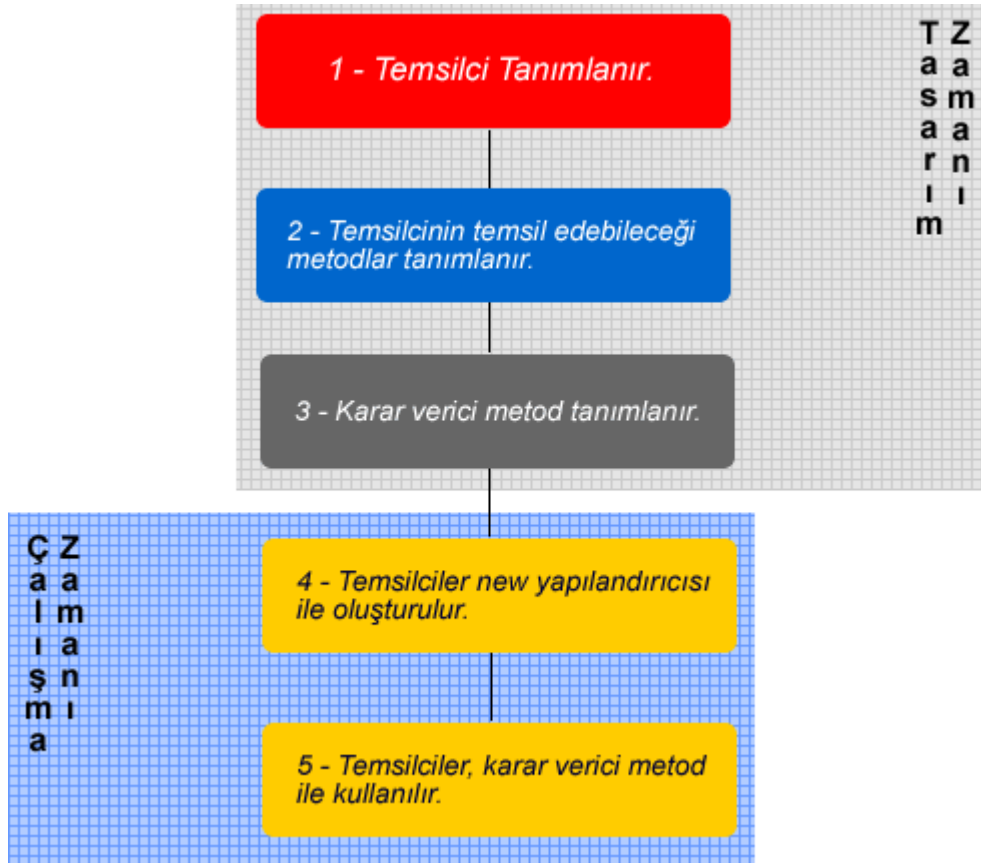
```
        if(Calistir.a>=10)
        {
            c.Metod1(t1);
        }
        else
        {
            c.Metod1(t2);
        }
    }
}
s}
```

Uygulamamızı çalıştıralım ve bir değer girelim.



Şekil 2. Programın çalışmasının sonucu.

Bu basit örnek ile umarım temsilciler hakkında biraz olsun bilgi sahibi olmuşsunuzdur. Şimdi temsilciler ile ilgili kavramlarımıza devam edelim. Yukarıdaki örneğimiz ışığında temsilcileri programlarımızda temel olarak nasıl kullandığımızı aşağıdaki şekil ile daha kolay anlayabileceğimizi sanıyorum.



Şekil 3. Temsilcilerin Karar Verici metodlar ile kullanımı.

Yukarıdaki örneğimizde, her bir metod için tek bir temsilci tanımladık ve temsilcileri teker teker çağırdık. Bu Single-Cast olarak adlandırılmaktadır. Ancak programlarımız da bazen, tek bir temsilciye birden fazla temsilci ekleyerek, birden fazla metodu tek bir temsilci ile çalıştırmak isteyebiliriz. Bu durumda Multi-Cast temsilciler tanımlarız. Şimdi multi-cast temsilciler ile ilgili bir örnek yapalım. Bu örneğimizde t1 isimli temsilcimiz, multi-cast temsilcimiz olacak.

```
using System;
namespace Delegates2
{
    public class temsilciler
    {
        public delegate void dgTemsilci(); /* Temsilcimiz tanımlanıyor. Geri dönüş değeri olmayan ve parametre almayan metodları temsil edebilir. */

        /* Metod1, Metod2 ve Metod3 temsilcilerimizin işaret etmesini istediğimiz metodlar olacaktır.*/
        public static void Metod1()
        {
            Console.WriteLine("Metod 1 çalıştırıldı.");
        }

        public static void Metod2()
        {
            Console.WriteLine("PI değeri 3.14 alınsın");
        }

        public static void Metod3()
```

```

    {
        Console.WriteLine("Mail gönderildi...");
    }

    /* Temsilcilerimizi çalıştıran metodumuz. Parametre olarak gönderilen temsilciyi,
    dolayısıyla bu temsilcinin işaret ettiği metodu alıyor. */

    public static void TemsilciCalistir(temsilciler.dgTemsilci dt)
    {
        dt(); /* Temsilcinin işaret ettiği metod çalıştırılıyor.*
    }
}

class Class1
{
    static void Main(string[] args)
    {
        /* Üç metodumuz içinde temsilci nesnelerimiz oluşturuluyor .*/

        temsilciler.dgTemsilci t1=new
Delegates2.temsilciler.dgTemsilci(temsilciler.Metod1);

        temsilciler.dgTemsilci t2=new
Delegates2.temsilciler.dgTemsilci(temsilciler.Metod2);

        temsilciler.dgTemsilci t3=new
Delegates2.temsilciler.dgTemsilci(temsilciler.Metod3);

        Console.WriteLine("sadece t1");

        temsilciler.TemsilciCalistir(t1);

        Console.WriteLine("---");

        /* Burada t1 temsilcimize, t2 temsilcisi ekleniyor. Bu durumda,
        t1 temsilcimiz hem kendi metodunu hemde, t2 temsilcisinin işaret ettiği metodu işaret
        etmeye başlıyor. Bu halde iken TemsilciCalistir metodumuza t1 temsilcisini göndermemiz
        her iki temsilcinin işaret ettiği metodların çalıştırılmasına neden oluyor.*/

        t1+=t2;

        Console.WriteLine("t1 ve t2");

        temsilciler.TemsilciCalistir(t1);

        Console.WriteLine("---");

        t1+=t3; /* Şimdi t1 temsilcimiz hem t1, hem t2, hem de t3 temsilcilerinin
        işaret ettiği metodları işaret etmiş olacak.*/

        Console.WriteLine("t1,t2 ve t3");
    }
}

```

```

        temsilciler.TemsilciCalistir(t1);

        Console.WriteLine("---");

        t1-=t2; /* Burada ise t2 metodunu t1 temsilcimizden çıkartıyoruz.
Böylece, t1 temsilcimiz sadece t1 ve t3 temsilcilerini içeriyor. */
        Console.WriteLine("t1 ve t3");

        temsilciler.TemsilciCalistir(t1);

        Console.WriteLine("---");
    }
}

```

Uygulamamızı çalıştırdığımızda aşağıdaki sonucu elde ederiz.

```

sadece t1
Metod 1 çalıştırıldı.
---
t1 ve t2
Metod 1 çalıştırıldı.
PI degeri 3.14 alinsin
---
t1,t2 ve t3
Metod 1 çalıştırıldı.
PI degeri 3.14 alinsin
Mail gönderildi...
---
t1 ve t3
Metod 1 çalıştırıldı.
Mail gönderildi...
---
Press any key to continue

```

Şekil 4. Multi-Cast temsilciler.

Geldik bir makalemizin daha sonuna. Bir sonraki makalemizde temsilcilerin kullanıldığı olaylar(events) kavramına gireceğiz. Hepinize mutlu günler dilerim.

Huffman Veri Sıkıştırma Algoritması ve Uygulaması

Bu makalede bilgisayar bilimlerinin önemli konularından biri olan veri sıkıştırma algoritmalarından Huffman algoritmasını inceledikten sonra uygulamasını gerçekleştirip sonuçlarını göreceğiz.

Sayısal haberleşme tekniklerinin önemli ölçüde arttığı günümüzde, sayısal verilen iletilmesi ve saklanması bir hayli önem kazanmıştır. Sayısal veriler çeşitli saklayıcılarda saklanırken hedef daima minimum alanda maksimum veriyi saklamadır. Veriler çeşitli yöntemlerle sıkıştırılarak kapladığı alandan ve iletim zamanından tasarruf edilir. Sayısal

iletiřim(digital communication) kuramında veriler çok çeřitli yöntemlerle sıkıřtırılabilir. Bu yöntemlerden en çok bilineni **David Huffman** tarafından öne sürölmüřtür. Bu yazıda bu teknik "Huffman algoritması" olarak adlandırılacaktır. Bu yazıda Huffman Algoritması detaylı olarak açıklandıktan sonra bu algoritmanın C# dili ile ne řekilde uygulanacađı gösterilecektir.

Sıkıřtırma algoritmaları temel olarak iki kategoride incelenir. Bunlar, kayıplı ve kayıpsız sıkıřtırma algoritmalarıdır. Kayıplı algoritmalarda sıkıřtırılan veriden orjinal veri elde edilemezken kayıpsız sıkıřtırma algoritmalarında sıkıřtırılmıř veriden orjinal veri elde edilebilir. Kayıplı sıkıřtırma tekniklerine verilebilecek en güzel örnekler MPEG ve JPEG gibi standartlarda kullanılan sıkıřtırmalardır. Bu tekniklerde sıkıřtırma oranı artırıldıđında orjinal veride bozulmalar ve kayıplar görölür. Örneđin sıkıřtırılmıř resim formatı olan JPEG dosyalarının kaliteli yada az kaliteli olmasının nedeni sıkıřtırma katsayısıdır. Yani benzer iki resim dosyasından daha az disk alanı kaplayan daha kötü kalitededir deriz. Kayıpsız veri sıkıřtırmada durum çok farklıdır. Bu tekniklerde önemli olan orjinal verilerin aynen sıkıřtırılmıř veriden elde edilmesidir. Bu teknikler daha çok metin tabanlı verilen sıkıřtırılmasında kullanılır. Bir metin dosyasını sıkıřtırdıktan sonra metindeki bazı cümlelerin kaybolması istenmediđi için metin sıkıřtırmada bu yöntemler kullanılır.

Bu yazının konusu olan Huffman sıkıřtırma algoritması kayıpsız bir veri sıkıřtırma tekniđini içerir. Bu yüzden bu yöntemin en elverişli olduđu veriler metin tabanlı verilerdir. Bu yazıda verilecek örnek programdaki hedef metin tabanlı verilerin sıkıřtırılması olacaktır.

Huffman algoritması, bir veri kümesinde daha çok rastlanan sembolü daha düşük uzunluktaki kodla, daha az rastlanan sembolleri daha yüksek uzunluktaki kodlarla temsil etme mantıđı üzerine kurulmuřtur. Bir örnekten yola çıkacak olursak : Bilgisayar sistemlerinde her bir karakter 1 byte yani 8 bit uzunluđuunda yer kaplar. Yani 10 karakterden oluřan bir dosya 10 byte büyüklüđuündedir. Çünkü her bir karakter 1 byte büyüklüđuündedir. Örneđimizdeki 10 karakterlik veri kümesi "aaaaaaaccs" olsun. "a" karakteri çok fazla sayıda olmasına rađmen "s" karakteri tektir. Eđer bütün karakterleri 8 bit deđilde veri kümesindeki sıklıklarına göre kodlarsak veriyi sembolize etmek için gereken bitlerin sayısı daha az olacaktır. Söz gelimi "a" karakteri için "0" kodunu "s" karakteri için "10" kodunu, "c" karakteri için "11" kodunu kullanabiliriz. Bu durumda 10 karakterlik verimizi temsil etmek için

$(a \text{ kodundaki bit sayısı}) * (\text{verideki a sayısı}) + (c \text{ kodundaki bit sayısı}) * (\text{verideki c sayısı}) + (s \text{ kodundaki bit sayısı}) * (\text{verideki s sayısı}) = 1 * 7 + 2 * 2 + 2 * 1 = 12 \text{ bit}$

gerekecektir. Halbuki bütün karakterleri 8 bit ile temsil etseydik $8 * 10 = 80$ bite ihtiyacımız olacaktı. Dolayısıyla %80 'in üzerinde bir sıkıřtırma oranı elde etmiř olduk. Burada dikkat edilmesi gereken nokta řudur : Veri kümesindeki sembol sayısına ve sembollerin tekrarlanma sıklıklarına bađlı olarak Huffman sıkıřtırma algoritması %10 ile %90 arasında bir sıkıřtırma oranı sağlayabilir. Örneđin içinde 2000 tane "a" karakteri ve 10 tane "e" karakteri olan bir veri kümesi Huffman tekniđi ile sıkıřtırılırsa %90'lara varan bir sıkıřtırma oranı elde edilir.

Huffman tekniđinde semboller(karakterler) ASCII'de olduđu gibi sabit uzunluktaki kodlarla kodlanmazlar. Her bir sembol deđiřken sayıda uzunluktaki kod ile kodlanır.

Bir veri kümesini Huffman tekniđi ile sıkıřtırabilmek için veri kümesinde bulunan her bir sembolün ne sıklıkta tekrarlandıđını bilmemiz gerekir. Örneđin bir metin dosyasını sıkıřtırıyorsak her bir karakterin metin içerisinde kaç adet geçtiđini bilmemiz gerekiyor. Her bir sembolün ne sıklıkta tekrarlandıđını gösteren tablo **frekans tablosu** olarak adlandırılmaktadır. Dolayısıyla sıkıřtırma işlemine geçmeden önce frekans tablosunu çıkarmamız gerekmektedir. Bu yöntem Statik Huffman tekniđi de denilmektedir. Diđer

bir teknik olan Dinamik Huffman tekniğinde sıkıştırma yapmak için frekans tablosuna önceden ihtiyaç duyulmaz. Frekans tablosu her bir sembolle karşılaştıkça dinamik olarak oluşturulur. Dinamik Huffman tekniği daha çok haberleşme kanalları gibi hangi verinin geleceği önceden belli olmayan sistemlerde kullanılmaktadır. Bilgisayar sistemlerindeki dosyaları sıkıştırmak için statik huffman metodu yeterlidir. Nitekim bir dosyayı baştan sona tarayarak her bir sembolün hangi sıklıkla yer aldığını tespit edip frekans tablosunu elde etmemiz çok basit bir işlemdir.

Huffman sıkıştırma tekniğinde frekans tablosunu elde etmek için statik ve dinamik yaklaşımlarının olduğunu söyledik. Eğer statik yöntem seçilmişse iki yaklaşım daha vardır. Birinci yaklaşım, metin dosyasının diline göre sabit bir frekans tablosunu kullanmaktır. Örneğin Türkçe bir metin dosyasında "a" ve "e" harflerine çok sık rastlanırken "ğ" harfine çok az rastlanır. Dolayısıyla "ğ" harfi daha fazla bitle "a" ve "e" harfi daha az bitle kodlanır. Frekans tablosunu elde etmek için kullanılan diğer bir yöntem ise metni baştan sona tarayarak her bir karakterin frekansını bulmaktır. Sizde takdir edersiniz ki ikinci yöntem daha gerçekçi bir çözüm üretmekle beraber metin dosyasının dilinden bağımsız bir çözüm üretmesi ile de ön plandadır. Bu yöntemin dezavantajı ise sıkıştırılan verilerde geçen sembollerin frekansının da bir şekilde saklanma zorunluluğunun olmasıdır. Sıkıştırılan dosyada her bir sembolün frekansında saklanmalıdır. Bu da küçük boyutlu dosyalarda sıkıştırma yerine genişletme etkisi yaratabilir. Ancak bu durum Huffman yönteminin kullanılabilirliğini zedelemeyiz. Nitekim küçük boyutlu dosyaların sıkıştırılmaya pek fazla ihtiyacı yoktur zaten.

Frekans tablosunu metin dosyasını kullanarak elde ettikten sonra yapmamız gereken **"Huffman Ağacını"** oluşturmaktır. Huffman ağacı hangi karakterin hangi bitlerle temsil edileceğini(kodlanacağını) belirlememize yarar. Birazdan örnek bir metin üzerinden "Huffman Ağacını" teorik olarak oluşturup algoritmanın derinliklerine ineceğiz. Bu örneği iyi bir şekilde incelediğinizde Huffman algoritmasının aslında çok basit bir temel üzerine kurulduğunu göreceksiniz.

Huffman Ağacının Oluşturulması

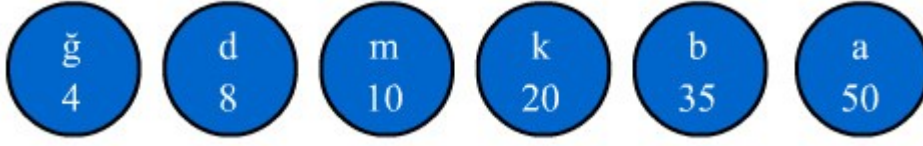
Bir huffman ağacı aşağıdaki adımlar izlenerek oluşturulabilir.

Bu örnekte aşağıdaki frekans tablosu kullanılacaktır.

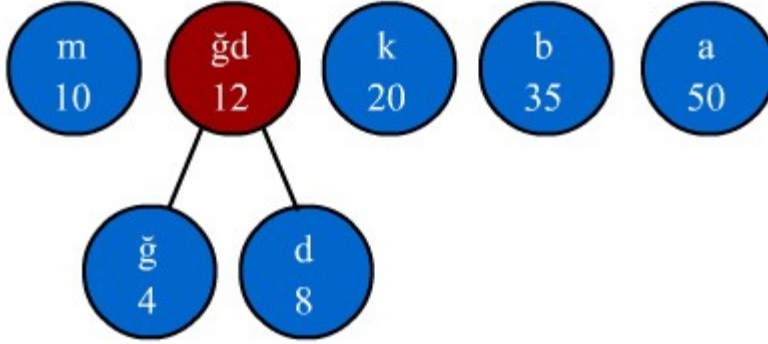
Sembol(Karakter)	Sembol Frekansı
a	50
b	35
k	20
m	10
d	8
ğ	4

Bu tablodan çıkarmamız gereken şudur : Elimizde öyle bir metin dosyası var ki "a" karakteri 50 defa, "b" karakteri 35 defa "ğ" karakteri 2 defa geçiyor. Amacımız ise her bir karakteri hangi bit dizileriyle kodlayacağımızı bulmak.

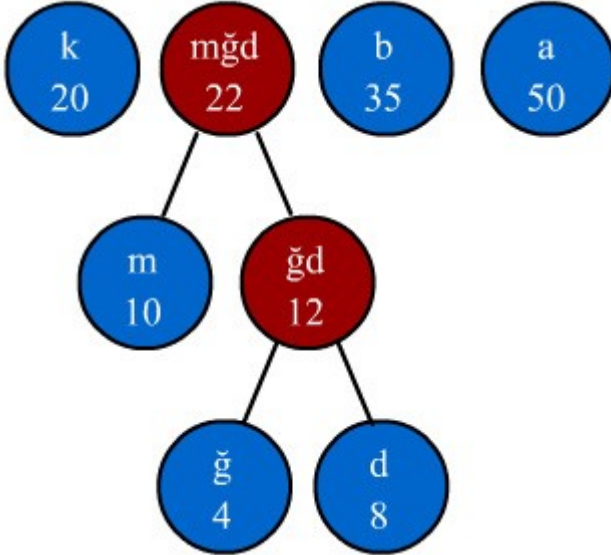
1 - Öncelikle "Huffman Ağacını" ndaki en son düğümleri(dal) oluşturacak bütün semboller frekanslarına göre aşağıdaki gibi küçükten büyüğe doğru sıralanırlar.



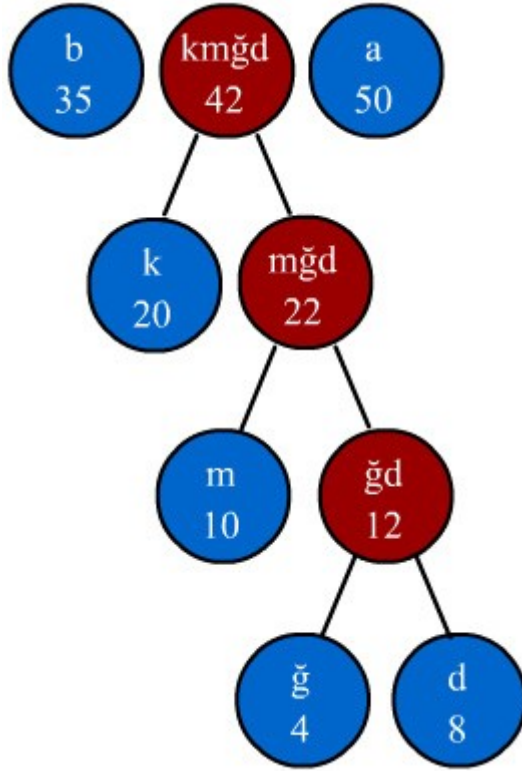
2 - En küçük frekansa sahip olan iki sembolün frekansları toplanarak yeni bir düğüm oluşturulur. Ve oluşturulan bu yeni düğüm diğer varolan düğümler arasında uygun yere yerleştirilir. Bu yerleştirme frekans bakımından küçüklük ve büyüklüğe göredir. Örneğin yukarıdaki şekilde "ğ" ve "d" sembolleri toplanarak "12" frekansında yeni bir "ğd" düğümü elde edilir. "12" frekanslı bir sembol şekilde "m" ve "k" sembolleri arasında yerleştirilir. "ğ" ve "d" düğümleri ise yeni oluşturulan düğümün dalları şeklinde kalır. Yeni dizimiz aşağıdaki şekilde olacaktır.



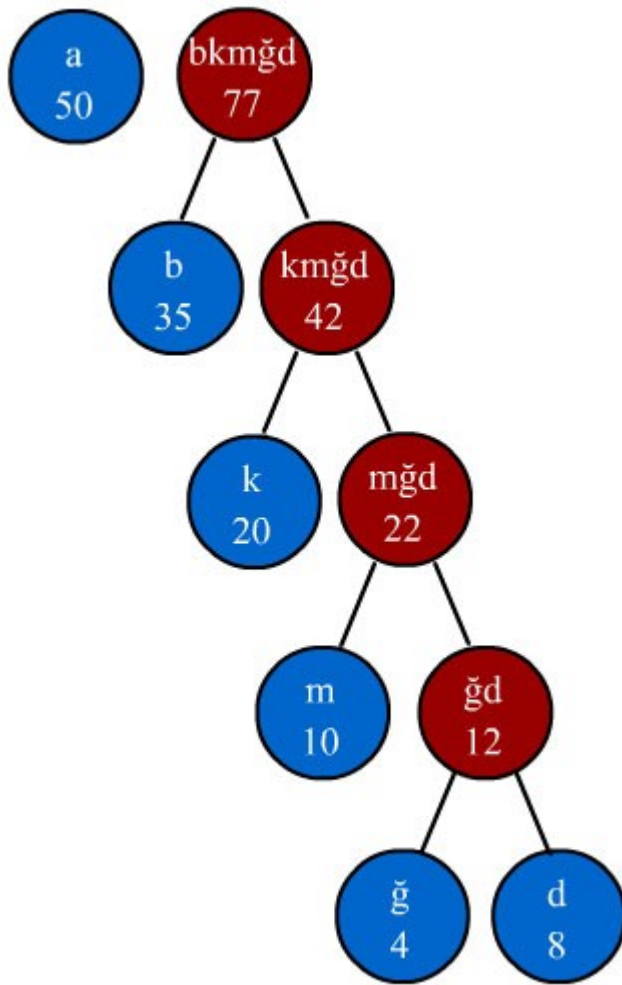
3 - 2.adımdaki işlem tekrarlanarak en küçük frekanslı iki düğüm tekrar toplanır ve yeni bir düğüm oluşturulur. Bu yeni düğümün frekansı 22 olacağı için "k" ve "b" düğümleri arasına yerleşecektir. Yeni dizimiz aşağıdaki şekilde olacaktır.



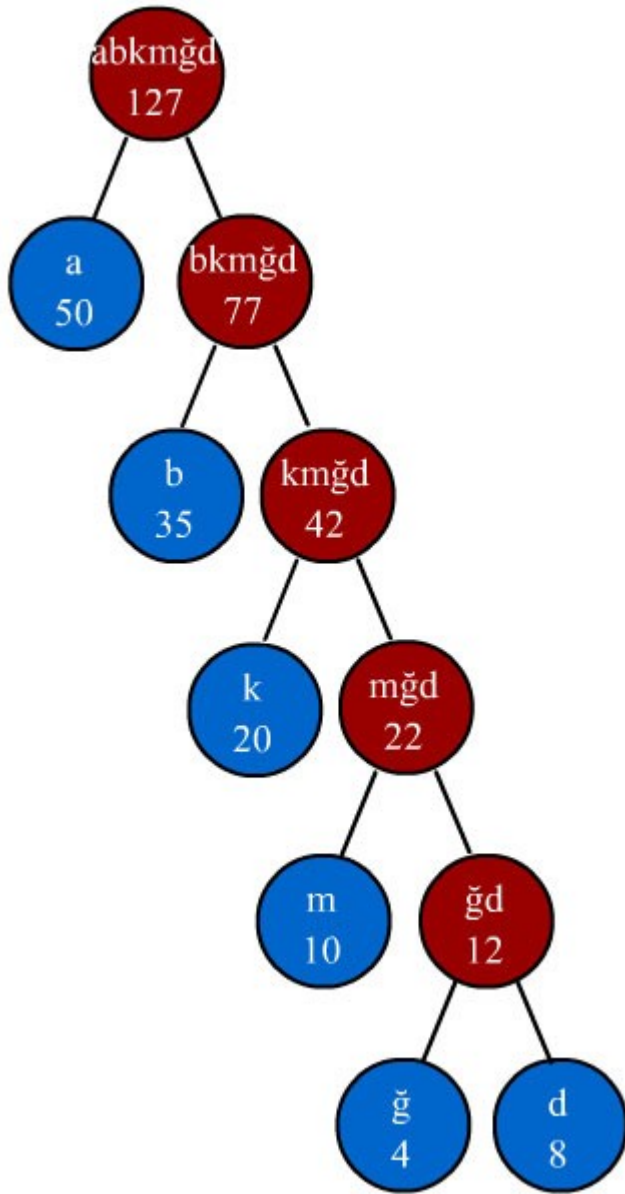
4 - 2.adımdaki işlem tekrarlanarak en küçük frekanslı iki düğüm tekrar toplanır ve yeni bir düğüm oluşturulur. Bu yeni düğümün frekansı 42 olacağı için "b" ve "a" düğümleri arasına yerleşecektir. Yeni dizimiz aşağıdaki şekilde olacaktır. Dikkat ederseniz her dalın en ucunda sembollerimiz bulunmaktadır. Dalların ucundaki düğümlere özel olarak **yaprak(leaf)** denilmektedir. Sona yaklaştıkça Bilgisayar bilimlerinde önemli bir veri yapısı olan Tree(ağaç) veri yapısına yaklaştığımızı görüyoruz.



5 - 2.adımdaki işlem tekrarlanarak en küçük frekanslı iki düğüm tekrar toplanır ve yeni bir düğüm oluşturulur. Bu yeni düğümün frekansı 77 olacağı için "a" düğümünden sonra yerleşecektir. Yeni dizimiz aşağıdaki şekilde olacaktır. Dikkat ederseniz her bir düğümün frekansı o düğümün sağ ve sol düğümlerinin frekanslarının toplamına eşit olmaktadır. Aynı durum düğüm sembolleri içinde geçerlidir.

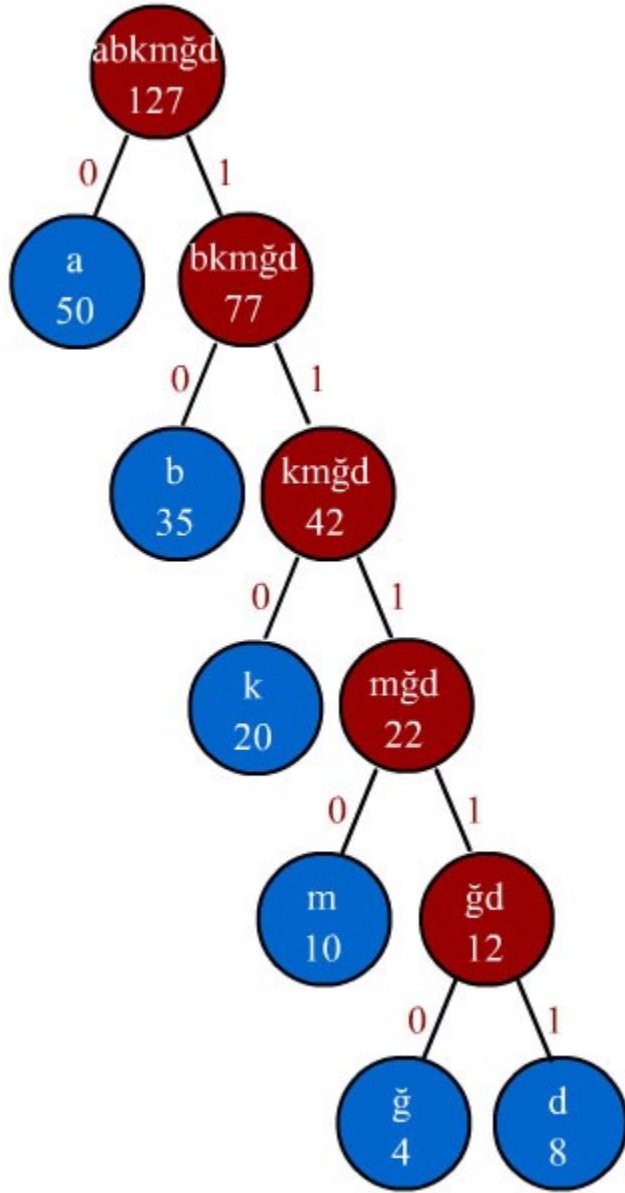


6 - 2.adımdaki işlem en tepede tek bir düğüm kalana kadar tekrar edilir. En son kalan düğüm Huffman ağacının kök düğümü(root node) olarak adlandırılır. Son düğümün frekansı 127 olacaktır. Böylece huffman ağacının son hali aşağıdaki gibi olacaktır.



Not : Dikkat ederseniz Huffman ağacının son hali simetrik bir yapıda çıktı. Yani yaprak düğümler hep sol tarafta çıktı. Bu tamamen seçtiğimiz sembol frekanslarına bağlı olarak rastlantısal bir sonuçtur. Bu rastlantının oluşması oluşturduğumuz her bir yeni düğümün yeni dizide ikinci sıraya yerleşmesinden kaynaklanmaktadır. Sembol frekanslarında değişiklik yaparak ağacın şeklindeki değişiklikleri kendinizde görebilirsiniz.

7 - Huffman ağacının son halini oluşturduğumuza göre her bir sembolün yeni kodunu oluşturmaya geçebiliriz. Sembol kodlarını oluştururken Huffman ağacının en tepesindeki kök düğümden başlanır. Kök düğümün sağ ve sol düğümlerine giden dala sırasıyla "0" ve "1" kodları verilir. Sırası ters yönde de olabilir. Bu tamamen seçime bağlıdır. Ancak ilk seçtiğiniz sırayı bir sonraki seçimlerde korumanız gerekmektedir. Bu durumda "a" düğümüne gelen dal "0", "bkmğd" düğümüne gelen dal "1" olarak seçilir. Bu işlem ağaçtaki tüm dallar için yapılır. Dallarin kodlarla işaretlenmiş hali aşağıdaki gibi olacaktır.



8 - Sıra geldi her bir sembolün hangi bit dizisiyle kodlanacağını bulmaya. Her bir sembol dalların ucunda bulunduğu için ilgili yaprağa gelene kadar dallardaki kodlar birleştirilip sembollerin kodları oluşturulur. Örneğin "a" karakterine gelene kadar yalnızca "0" dizisi ile karşılaşırız. "b" karakterine gelene kadar önce "1" dizisine sonra "0" dizisi ile karşılaşırız. Dolayısıyla "b" karakterinin yeni kodu "10" olacaktır. Bu şekilde bütün karakterlerin sembol kodları çıkarılır. Karakterlerin sembol kodları aşağıda bir tablo halinde gösterilmiştir.

Frekans	Sembol(Karakter)	Bit Sayısı	Huffman Kodu
50	a	1	0
35	b	2	10
20	k	3	110

10	m	4	1110
8	d	5	11111
4	ğ	5	11110

Sıkıştırılmış veride artık ASCII kodları yerine Huffman kodları kullanılacaktır. Dikkat ederseniz hiçbir Huffman kodu bir diğer Huffman kodunun ön eki durumunda değildir. Örneğin ön eki "110" olan hiç bir Huffman kodu mevcut değildir. Aynı şekilde ön eki "0" olan hiç bir Huffman kodu yoktur. Bu Huffman kodları ile kodlanmış herhangi bir veri dizisinin **"tek çözülebilir bir kod"** olduğunu göstermektedir. Yani sıkıştırılmış veriden orjinal verinin dışında başka bir veri elde etme ihtimali sıfırdır. (Tabi kodlamanın doğru yapıldığını varsayıyoruz.)

Şimdi örneğimizdeki gibi bir frekans tablosuna sahip olan metnin Huffman algoritması ile ne oranda sıkışacağını bulalım :

Sıkıştırma öncesi gereken bit sayısını bulacak olursak : Her bir karakter eşit uzunlukta yani 8 bit ile temsil edildiğinden toplam karakter sayısı olan $(50+35+20+10+8+4) = 127$ ile 8 sayısını çarpmamız lazım. Orjinal veriyi sıkıştırmadan saklarsak **$127 \times 8 = 1016$** bit gerekmektedir.

Huffman algoritmasını kullanarak sıkıştırma yaparsak kaç bitlik bilgiye ihtiyaç duyacağımızı hesaplayalım : 50 adet "a" karakteri için 50 bit, 35 adet "b" karakteri için 70 bit, 20 adet "k" karakteri için 60 bit....4 adet "ğ" karakteri için 20 bite ihtiyaç duyarız. (yukarıdaki tabloya bakınız.) Sonuç olarak gereken toplam bit sayısı = $50 \times 1 + 35 \times 2 + 20 \times 3 + 10 \times 4 + 8 \times 5 + 4 \times 5 = 50 + 70 + 60 + 40 + 40 + 20 = 280$ bit olacaktır.

Sonuç : 1016 bitlik ihtiyacımızı 280 bite indirdik. Yani yaklaşık olarak %72 gibi bir sıkıştırma gerçekleştirmiş olduk. Gerçek bir sistemde sembol frekanslarının da saklamak gerektiği için sıkıştırma oranı %72'ten biraz daha az olacaktır. Bu fark genelde sıkıştırılan veriye göre çok çok küçük olduğu için ihmal edilebilir.

Huffman Kodunun Çözülmesi

Örnekte verilen frekans tablosuna sahip bir metin içerisindeki "aabkdğmma" veri kümesinin sıkıştırılmış hali her karakter ile karakterin kodu yer değiştirilerek aşağıdaki gibi elde edilir.

a a b k d ğ m m a
0 0 10 110 11111 11110 1110 1110 0 --> 0010110111111110111011100

Eğer elimizde frekans tablosu ve sıkıştırılmış veri dizisi varsa işlemlerin tersini yaparak orjinal veriyi elde edebiliriz. Şöyleki; sıkıştırılmış verinin ilk biti alınır. Eğer alınan bit bir kod sözcüğüne denk geliyorsa, ilgili kod sözcüğüne denk düşen karakter yerine koyulur, eğer alınan bit bir kod sözcüğü değilse sonraki bit ile birlikte ele alınır ve yeni dizinin bir kod sözcüğü olup olmadığına bakılır. Bu işlem dizinin sonuna kadar yapılır ve huffman kodu çözülür. Huffman kodları tek çözülebilir kod olduğu için bir kod dizisinden farklı semboller elde etmek olanaksızdır. Yani bir huffman kodu ancak ve ancak bir şekilde çözülebilir. Bu da aslında yazının başında belirtilen kayıpsız sıkıştırmanın bir sonucudur.

C# ile Huffman Algoritmasının Gerçekleştirilmesi

Huffman algoritması bilgisayar bilimlerinde genellikle metin dosyalarının sıkıştırılmasında kullanılır. Bu yazıda örnek bir program ile Huffman algoritmasının ne şekilde uygulanabileceğini de göreceğiz. Dil olarak tabiki C#'ı kullanılacaktır.

Huffman algoritmasının uygulanmasındaki ilk adım frekans tablosunun bulunmasıdır. Bu yüzden bir dosyayı sıkıştırmadan önce dosyadaki her bir karakterin ne sıklıkla yer aldığını bulmak gerekir. Örnek programda sembol frekanslarını bellekte tutmak için System.Collections isim alanından bulunan Hashtable koleksiyon yapısı kullanılmıştır. Örneğin "c" karakterinin frekansı 47 ise,

CharFrequencies["c"] = 47

şeklinde sembolize edilir. Örnek programda frekans tablosu aşağıdaki metot ile elde edilebilir. Sıkıştırılacak dosya baştan sona taranarak frekanslar hashtable nesnesine yerleştirilir.

[Not : Örnek uygulamada kullandığım değişken ve metot isimlerini bazı özel nedenlerden dolayı İngilizce yazmak zorunda kaldığım için tüm okurlardan özür dilerim.]

```
private Hashtable CharFrequencies = new Hashtable();

private void MakeCharFrequencies()
{
    FileStream fs = File.Open(file, FileMode.Open, FileAccess.Read);

    StreamReader sr = new StreamReader(fs, System.Text.Encoding.ASCII);

    int iChar;
    char ch;
    while((iChar = sr.Read()) != -1)
    {
        ch = (char)iChar;
        if(!CharFrequencies.ContainsKey(ch.ToString()))
        {
            CharFrequencies.Add(ch.ToString(), 1);
        }
        else
        {
            int oldFreq = (int)CharFrequencies[ch.ToString()];
            int newFreq;
            newFreq = oldFreq + 1;
            CharFrequencies[ch.ToString()] = newFreq;
        }
    }

    sr.Close();
    fs.Close();
    sr = null;
    fs = null;
}
```

Frekans tablosunu yukarıdaki metot yardımıyla oluşturduktan sonra huffman algoritmasının en önemli adımı olan huffman ağacının oluşturulmasına sıra geldi. Huffman ağacının oluşturulması en önemli ve en kritik noktadır. Huffman ağacı oluşturulurken Tree(ağaç) veri yapısından faydalanılmıştır. Ağaçtaki her bir düğümü temsil etmek için bir sınıf yazılmıştır. Huffman ağacındaki düğümleri temsil etmek için kullanılabilecek en basit düğüm sınıfının yapısı aşağıdaki gibidir.



Yukarıdaki düğüm yapısı algoritmayı uygulamak için gereken en temel düğüm yapısıdır. Bu yapı istenildiği gibi genişletilebilir. Önemli olan minimum gerekliliği sağlamaktır. Düğüm sınıfındaki SagDüğüm ve Soldüğüm özellikleri de düğüm türündendir. Başlangıçtaki düğümler için bu değerler null dır. Her bir yeni düğüm oluşturulduğunda yeni düğümün sağ ve sol düğümleri oluşur. Ancak unutmamak gerekir ki sadece sembollerimizi oluşturan yaprak düğümlerde bu özellikler null değerdedir.

Örnek uygulamadaki **HuffmanNode** sınıfında ayrıca ilgili düğümün yaprak olup olmadığını gösteren IsLeaf ve ilgili düğümün ana(parent) düğümünü gösteren HuffmanNode türünden ParentNode özellikleri bulunmaktadır.

HuffmanNode düğüm sınıfının yapısı aşağıdaki gibidir.

```
using System;

namespace Huffman
{
    public class HuffmanNode
    {
        private HuffmanNode leftNode;
        private HuffmanNode rightNode;

        private HuffmanNode parentNode;

        private string symbol;
        private int frequency;
        private string code = "";

        private bool isLeaf;

        public HuffmanNode LeftNode
        {
            get{return leftNode;}
            set{leftNode = value;}
        }

        public HuffmanNode RightNode
        {
            get{return rightNode;}
            set{rightNode = value;}
        }

        public HuffmanNode ParentNode
        {
            get{return parentNode;}
        }
    }
}
```

```

        set{parentNode = value;}
    }

    public string Symbol
    {
        get{return symbol;}
        set{symbol = value;}
    }

    public string Code
    {
        get{return code;}
        set{code = value;}
    }

    public int Frequency
    {
        get{return frequency;}
        set{frequency = value;}
    }

    public bool IsLeaf
    {
        get{return isLeaf;}
        set{isLeaf = value;}
    }

    public HuffmanNode()
    {
    }
}

public class NodeComparer : IComparer
{
    public NodeComparer()
    {
    }

    public int Compare(object x, object y)
    {
        HuffmanNode node1 = (HuffmanNode)x;
        HuffmanNode node2 = (HuffmanNode)y;

        return node1.Frequency.CompareTo(node2.Frequency);
    }
}

```

Başlangıçta sembol sayısı kadar HuffmanNode nesnesi yani huffman düğümü oluşturmamız gerekecektir. Oluşturulan bu düğümler ArrayList koleksiyon yapısında tutulmaktadır. Her bir yeni düğüm oluşturulduğunda ArrayList kolaksiyonun yeni bir düğüm eklenir ve iki düğüm çıkarılır. Yeni düğümün sağ ve sol düğümleri çıkarılan

düğüm olacaktır. İlk oluşturulan düğümlerin sağ ve sol düğümlerinin null olduğunu hatırlayınız. Yeni düğüm eklendiğinde ArrayList sınıfının Sort() metodu yardımıyla düğümler frekanslarına göre küçükten büyüğe doğru sıralanır. Yani ArrayList elemanını ilk elemanı en küçük frekanslı düğümdür. Sort() metodu IComparer arayüzünü kullanan NodeComparer sınıfını kullanmaktadır. Yani bu sınıftaki Compare() metodu yukarıdaki kodda belirtildiği gibi düğüm nesnelerinin neye göre sıralanacağını tanımlar.

Yukarıdaki tanımlar ışığında Huffman ağacı aşağıdaki metot yardımıyla bulunabilir. Dikkat ettiyseniz, metodun icrası bittiğinde Nodes koleksiyonunda tek bir düğüm nesnesi kalacaktır, bu da huffman ağacındaki kök düğümden(root node) başkası değildir.

```
private ArrayList Nodes;

public void MakeHuffmanTree()
{
    Nodes = new ArrayList();

    //Başlangıç düğümleri oluşturuluyor.
    foreach(Object Key in CharFrequencies.Keys)
    {
        HuffmanNode node = new HuffmanNode();
        node.LeftNode = null;
        node.RightNode = null;
        node.Frequency = (int)(CharFrequencies[Key]);
        node.Symbol = (string)(Key);
        node.IsLeaf = true;
        node.ParentNode = null;

        Nodes.Add(node);
    }

    //Düğümün neye göre sıralanacağı NodeComparer sınıfındaki Compare metodunda tanımlıdır.
    Nodes.Sort(new NodeComparer());

    while(Nodes.Count > 1)
    {
        HuffmanNode node1 = (HuffmanNode)Nodes[0];
        HuffmanNode node2 = (HuffmanNode)Nodes[1];

        HuffmanNode newnode1 = new HuffmanNode();
        newnode1.Frequency = node1.Frequency + node2.Frequency;
        newnode1.IsLeaf = false;
        newnode1.LeftNode = node1;
        newnode1.RightNode = node2;
        newnode1.Symbol = node1.Symbol + node2.Symbol;

        node1.ParentNode = newnode1;
        node2.ParentNode = newnode1;

        Nodes.Add(newnode1);

        Nodes.Remove(node1);
        Nodes.Remove(node2);
    }
}
```

```
        Nodes.Sort(new NodeComparer());
    }
}
```

Algoritmanın son adımı ise Huffman kodlarının oluşturulmasıdır. Programatik olarak kodlamanın en zor olduğu bölüm burasıdır. Zira burada huffman ağacı öz-yineli(recursive) olarak taranarak her bir düğümün huffman kodu atanacaktır. Performansı artırmak için semboller ve kod sözcükleri ayrıca bir Hashtable koleksiyonunda saklanmıştır. Böylece her bir kodun çözülmesi sırasında ağaç yapısı taranmayacak, bunun yerine Hashtable'dan ilgili kod bulunacaktır. Kodlamayı yapan metot aşağıdaki gibidir.

```
private void FindCodeWords(HuffmanNode Node)
{
    HuffmanNode leftNode = Node.LeftNode;
    HuffmanNode rightNode = Node.RightNode;

    if(leftNode == null && rightNode == null)
        Node.Code = "1";

    if(Node == this.RootHuffmanNode)
    {
        if(leftNode != null)
            leftNode.Code = "0";

        if(rightNode != null)
            rightNode.Code = "1";
    }
    else
    {
        if(leftNode != null)
            leftNode.Code = leftNode.ParentNode.Code + "0";

        if(rightNode != null)
            rightNode.Code = rightNode.ParentNode.Code + "1";
    }

    if(leftNode != null)
    {
        if(!leftNode.IsLeaf)
            FindCodeWords(leftNode);
        else
        {
            CodeWords.Add(leftNode.Symbol[0].ToString(), leftNode.Code);
        }
    }

    if(rightNode != null)
    {
        if(!rightNode.IsLeaf)
            FindCodeWords(rightNode);
        else
        {

```

```

        CodeWords.Add(rightNode.Symbol[0].ToString(),rightNode.Code);
    }
}

```

Huffman algoritmasının temel basamakları bitmiş durumda. Bundan sonraki kodlar elde edilen sıkıştırılmış verinin ne şekilde dosyaya kaydedileceği ve sıkıştırılmış dosyanın nasıl tekrar geri elde edileceği ile ilgilidir. Bu noktada uygulamamız için bir dosya formatı belirlememiz gerekmektedir. Hatırlanacağı üzere sıkıştırılmış veriyi tekrar eski haline getirmek için frekans tablosuna ihtiyacımız vardır. Dolayısıyla sıkıştırılmış veriyi dosyaya yazmadan önce dosya formatımızın kurallarına göre sembol frekanslarını dosyaya yazmamız gerekir.

Benim programı yazarken geliştirdiğim dosya formatı aşağıdaki gibidir.

dosya.huff

- 1-)** İlk 4 byte --> Orjinal verideki toplam sembol sayısı.(ilk byte yüksek anlamlı byte olacak şekilde)
- 2 -)** Sonraki 1 byte --> Data bölümünde bulunan son byte'taki ilk kaç bitin data'ya dahil olduğunu belirtir. (Data bölümündeki bitlerin sayısı 8'in katı olmayabilir.)
- 3 -)** 2 byte Sembol + 4 byte sembol frekansı (bu bölüm ilk 4 byte'ta belirtilen sembol sayısı kadar tekrar edecektir.-ilk byte yüksek anlamlı byte olacak şekilde-)
- 4 -)** Son bölümde ise sıkıştırılmış veri bit dizisi şeklinde yer alır.

Aşağıdaki metot yukarıda belirlenen kurallara göre sıkıştırılmış veriyi dosyaya .huff uzantılı olarak kaydeder.

[Not : Yazılacak programa göre bu dosya formatı değişebilir. Örneğin birden fazla dosyayı aynı anda sıkıştıran bir program için bu format tamamen değişecektir.]

```

public void WriteCompressedData()
{
    /// 4 byte ->(uint) Sembol Sayısı(K adet)
    /// 1 byte ->(uint) Data bölümünde bulunan son byte'taki ilk kaç bitin data'ya dahil
    olduğunu belirtir.
    ///
    /// -----SEMBOL FREKANS BÖLÜMÜ-----
    ///
    /// 2 byte Sembol + 4 byte Sembol Frekansı
    /// .
    /// . K adet
    /// .
    /// 2 byte Sembol + 4 byte Sembol Frekansı
    ///
    /// -----DATA BÖLÜMÜ-----
    ///
    /// Bu bölümde Sıkıştırılmış veriler byte dizisi şeklinde yer alır.
    ///
    /// Data bölümü (6*K + 6) numaralı byte'tan itibaren başlar.
    ///

    int K = CharFrequencies.Keys.Count;

```

```

/*K = 4 ise
*
* byte[0] = 0 0 0 0 0 1 0 0
* byte[1] = 0 0 0 0 0 0 0 0
* byte[2] = 0 0 0 0 0 0 0 0
* byte[3] = 0 0 0 0 0 0 0 0
*
* K(bits) = 00000100 00000000 00000000 00000000
* */

byte[] byteK = BitConverter.GetBytes(K);

FileStream fs = File.Open(this.NewFile, FileMode.Create, FileAccess.ReadWrite);

byte[] CompressedByteStream = GenerateCompressedByteStream();
WriteByteArrayToStream(byteK, fs);
fs.WriteByte(RemainderBits);

foreach(string sembol in CharFrequencies.Keys)
{
    byte[] byteC = BitConverter.GetBytes((char)sembol[0]);
    byte[] bytesFreqs = BitConverter.GetBytes((int)CharFrequencies[sembol]);

    WriteByteArrayToStream(byteC, fs);
    WriteByteArrayToStream(bytesFreqs, fs);
}

WriteByteArrayToStream(CompressedByteStream, fs);

fs.Flush();
fs.Close();
}

```

Sıkıştırılan bir veri geri elde edilemediği sürece sıkıştırmanın bir anlam ifade etmeyeceği düşünülürse sıkıştırma işleminin tersinide yazmamız gerekmektedir. Sıkıştırma işlemi yaparken yaptığımız işlemlerin tersini yaparak orjinal veriyi elde edebiliriz. Sıkıştırılmış bir dosyayı açmak için aşağıdaki adımlar izlenir.

- 1 -) .huff uzantılı dosyadan sembol frekansları okunur ve Hashtable nesnesine yerleştirilir.
- 2 -) Frekans tablosu kullanılarak Huffman ağacı oluşturulur.
- 3 -) Huffman ağacı kullanılarak Huffman kodları oluşturulur.
- 4 -) .huff uzantılı dosyanın data bölümünden sıkıştırılmış veri dizisi okunarak huffman kodları ile karşılaştırılarak orjinal metin bulunur ve .txt uzantılı dosyaya yazılır.

2. ve 3. adımlardaki işlemler sıkıştırma yapıldığında kullanılan işlemler ile aynıdır. Burada ayrıca değenmeye gerek yoktur. Bu aşamada önemli olan verilen okunması ve yazılmasında izlenen yoldur. Verilerin okunmasında ve tekrar yazılmasında **StringBuilder** sınıfı kullanılmıştır. Bu sınıf string nesnesine göre oldukça iyi performans göstermektedir. Örneğin aynı işlemi string ile yaptığımda 75 K'lık bir dosya açma işlemi 30 sn sürerken StringBuilder kullandığımda 2 sn sürmektedir.

1.aşamdaki işlemi yapacak metot aşağıdaki gibidir.

```

private System.Text.StringBuilder CompressedData = new
System.Text.StringBuilder("");

private void ReadCompressedFile()
{
    FileStream fs = File.Open(file, FileMode.Open, FileAccess.Read);

    //İlk önce sembol sayısını bulalım.
    byte[] byteK = new byte[4];

    for(int i=0 ; i<4 ; i++)
    {
        byteK[i] = (byte)fs.ReadByte();
    }

    int SymbolCount = BitConverter.ToInt32(byteK,0);

    int RemainderBitCount = (byte)fs.ReadByte();

    for(int k = 0; k < SymbolCount ; k++)
    {
        //Sembollerin elde edilmesi(2 byte)
        byte[] symbolB = new byte[2];
        symbolB[0] = (byte)fs.ReadByte();
        symbolB[1] = (byte)fs.ReadByte();

        char cSymbol = BitConverter.ToChar(symbolB,0);

        //Frekans bilgisinin elde edilmesi(4 byte)
        byte[] freqB = new byte[4];
        for(int j=0 ; j<4 ; j++)
        {
            freqB[j] = (byte)fs.ReadByte();
        }

        int freq = BitConverter.ToInt32(freqB,0);

        CharFrequencies.Add(cSymbol.ToString(),freq);
    }

    //Data bölümünün okunması
    int readByte;
    while((readByte = fs.ReadByte()) != -1)
    {
        byte b = (byte)readByte;

        bool[] bits = new bool[8];
        bits = GetBitsOfByte(b);
        BitArray ba = new BitArray(bits);

        for(int m=0 ; m < ba.Length ; m++)
        {
            if(ba[m])
                CompressedData.Append("1");
            else

```

```

        CompressedData.Append("0");
    }
}

//Son byte'taki fazla veriler atılıyor
int removingBits = 8 - RemainderBitCount;

if(RemainderBitCount != 0)
{
    CompressedData.Remove(CompressedData.Length - removingBits, removingBits);
}

fs.Flush();
fs.Close();
}

```

Dosyayı açarken son yapılması gereken adım okunan verilerin dosyaya yazılmasıdır. Bir önceki adımda okunan veriler aşağıdaki metot aracılığıyla yeni bir dosyaya kaydedilir.

```

System.Text.StringBuilder OriginalData = new System.Text.StringBuilder("");

private void WriteOriginalData()
{
    FileStream fs = File.Open(this.NewFile, FileMode.Create, FileAccess.ReadWrite);

    StreamWriter sw = new StreamWriter(fs, System.Text.Encoding.ASCII);

    System.Text.StringBuilder sb = new System.Text.StringBuilder("");

    //Hashtable'ın performansından faydalanmak için kod sözcükleri ters çevrilip yeni bir
    hashtable oluşturuluyor.
    ReverseCodeWordsHashtable();

    for(int i=0 ; i < CompressedData.Length ; i++)
    {
        sb.Append(CompressedData[i]);

        string symbol = "";
        bool found = false;

        if(CodeWords.ContainsValue(sb.ToString()))
        {
            symbol = (string)ReversedCodeWords[sb.ToString()];
            found = true;
        }

        if(found)
        {
            OriginalData.Append(symbol);
            sb.Remove(0,sb.Length);
        }
    }
}

```

```
sw.Write(OriginalData.ToString());

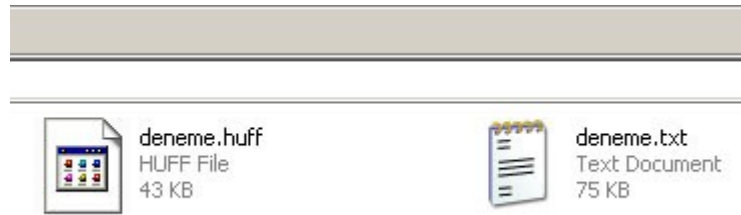
sw.Flush();
fs.Flush();
sw.Close();
fs.Close();
}
```

Yukarıdaki metodların tamamı Huffman isimli bir metotta toplanmıştır. Dosya sıkıştırma ve dosya açma için farklı nesnelerin kullanılması gerekmektedir. Huffman sınıfının örnek kullanımı aşağıda verilmiştir.

```
//Sıkıştırma
Huffman hf1 = new Huffman("deneme.txt");
hf1.Compress();

//Açma
Huffman hf2 = new Huffman("deneme.huff");
hf2.Decompress();
```

Uygulamayı 75 K'lık bir İngilizce metin tabanlı dosya üzerinde çalıştırdığımda dosyanın 43 K'ya düştüğünü aşağıdaki gibi gözlemledim.



Umarım faydalı bir prgram ve faydalı bir yazı olmuştur.

Uygulamanın bütün kodlarını ve proje dosyasını indirmek için tıklayınız.

Not : Bu makalenin son cümlesini yazdığımda programın açma modülünde Türkçe karakterler ile ilgili bir bug'ın olduğunu farkettim. Yani şu anda herhangi bir türkçe karakter içeren dosyayı sıkıştırıp dosyayı tekrar açtığınızda Türkçe karakterler yerine "?" karakterini göreceksiniz. En kısa zamanda bug'ı düzelterip kaynak kodları yeniden yükleyeceğim. Bug'ın nedenini bulup düzelten kişiye süpriz bir ödül vereceğimide belirtmek isterim.

Bir Arayüz, Bir Sınıf ve Bir Tablo

Bugünkü makalemizde, bir arayüzü uygulayan sınıf nesnelerinden faydalanarak, bir Sql tablosundan nasıl veri okuyacağımızı ve değişiklikleri veritabanına nasıl göndereceğimizi incelemeye çalışacağız. Geliştireceğimiz örnek, arayüzlerin nasıl oluşturulduğu ve bir sınıfa nasıl uygulandığını incelemekle yetinmeyecek, Sql veritabanımızdaki bir tablodaki belli bir kayda ait verilerin bu sınıf nesnelerine nasıl aktarılacağını da işleyecek. Kısacası uygulamamız, hem arayüzlerin hem sınıfların hemde Sql nesnelerinin kısa bir tekrarı olacak.

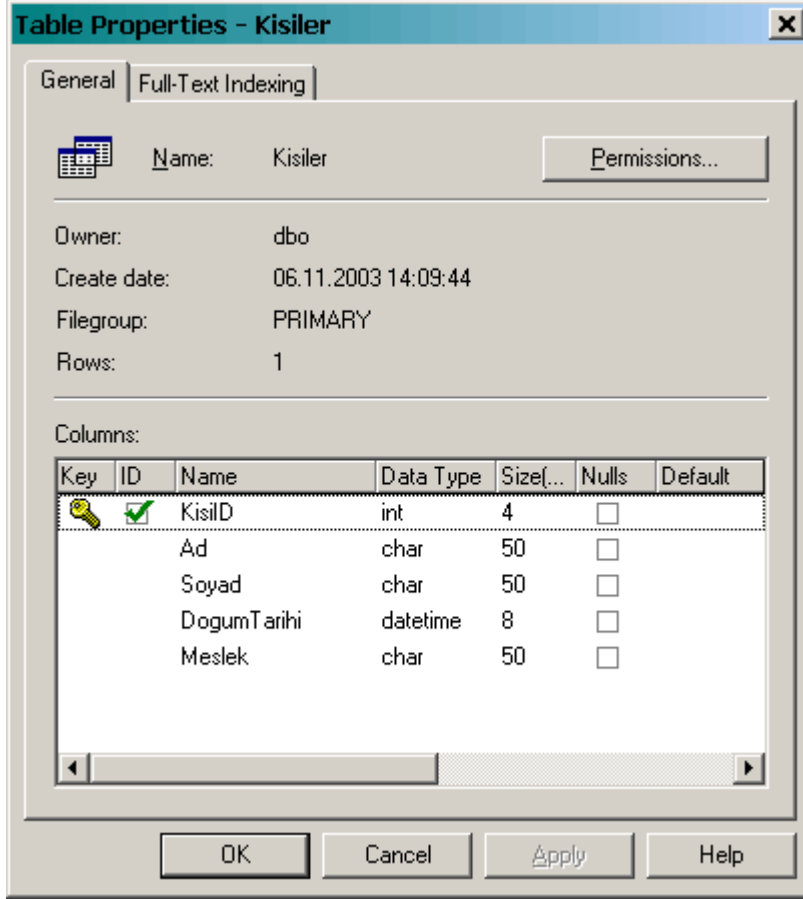
Öncelikle uygulamamızın amacından bahsedelim. Uygulamamızı bir Windows uygulaması şeklinde geliştireceğiz. Kullanacağımız veri tablosunda arkadaşlarımızla ilgili bir kaç veriyi tutuyor olacağız. Kullanıcı, Windows formunda, bu tablodaki alanlar için Primary Key niteliği taşıyan bir ID değerini girerek, buna karşılık gelen tablo satırına ait verilerini elde edecek. İsteddiği değişiklikleri yaptıktan sonra ise bu değişiklikleri tekrar veritabanına gönderecek. Burada kullanacağımız teknik makalemizin esas amacı olacak. Bu kez veri tablosundan çekip aldığımız veri satırının programdaki eşdeğeri, oluşturacağımız sınıf nesnesi olacak. Bu sınıfımız ise, yazmış olduğumuz arayüzü uygulayan bir sınıf olacak. Veriler sınıf nesnesine, satırdaki her bir alan değeri, aynı isimli özelliğe denk gelecek şekilde yüklenecek. Yapılan değişiklikler yine bu sınıf nesnesinin özelliklerinin sahip olduğu değerlerin veri tablosuna gönderilmesi ile gerçekleştirilecek.

Uygulamamızda, verileri Sql veritabanından çekmek için, SqlConnection isim uzayında yer alan SqlConnection ve SqlDataReader nesnelerini kullanacağız. Hatırlayacağınız gibi SqlConnection nesnesi ile , bağlanmak istediğimiz veritabanına, bu veritabanının bulunduğu sunucu üzerinden bir bağlantı tanımlıyoruz. SqlDataReader nesnemiz ile de, sadece ileri yönlü ve yalnız okunabilir bir veri akımı sağlayarak, aradığımız kayda ait verilerin elde edilmesini sağlıyoruz. Şimdi uygulamamızı geliştirmeye başlayalım. Öncelikle vs.net ortamında bir Windows Application oluşturalım. Burada aşağıdaki gibi bir form tasarlayalım.



Şekil 1. Form tasarımıımız.

Kullanıcı bilgilerini edinmek istediği kişinin ID'nosunu girdikten sonra, Getir başlıklı butona tıklayarak ilgili satırın tüm alanlarına ait verileri getirecek. Ayrıca, kullanıcı veriler üzerinde değişiklik yapabilecek ve bunlarıda Güncelle başlıklı butona tıklayarak Sql veritabanındaki tablomuza aktarabilecek. Sql veritabanında yer alan Kisiler isimli tablomuzun yapısı aşağıdaki gibidir.



Şekil 2. Tablomuzun yapısı.

Şimdi gelelim işin en önemli ve anahtar kısımlarına. Program kodlarımız. Öncelikle arayüzümüzü tasarlayalım. Arayüzümüz, sonra oluşturacağımız sınıf için bir rehber olacak. Sınıfımız, veri tablomuzdaki alanları birer özellik olarak taşıyacağına göre arayüzümüzde bu özellik tanımlarının yer alması gerektiğini söyleyebiliriz. Ayrıca ilgili kişiye ait verileri getirecek bir metodumuzda olmalıdır. Elbette bu arayüze başka amaçlar için üye tanımlamalarıda ekleyebiliriz. Bu konuda tek sınır bizim hayal gücümüz. İşin gerçeği bu makalemizde hayal gücümü biraz kısıdım konunun daha fazla dağılmaması amacıyla :) . (Ama siz, örneğin kullanıcının yeni girdiği verileri veritabanına yazacak bir metod tanımınıda bir üye olarak ekleyebilir ve gerekli kodlamaları yapabilirsiniz.) İşte arayüzümüzün kodları.

```
public interface IKisi
{
    /* Öncelikle tablomuzdaki her alana karşılık gelen özellikler için tanımlamalarımızı yapıyoruz.*/
    int KisiID /* KisiID, tablomuzda otomatik artan ve primary key olan bir alandır. Dolayısıyla programcının var olan bir KisiID'sini değiştirmemesi gerekir. Bu nedenle sadece okunabilir bir özellik olarak tanımlanmasına izin veriyoruz. */
}
```

```

{
    get;
}
string Ad /* Tablomuzdaki char tipindeki Ad alanımız için string tipte bir alan.*/
{
    get;set;
}
string Soyad
{
    get;set;
}
DateTime DogumTarihi /* Tablomuzda, DogumTarihi alanımız datetime tipinde
olduğundan, DateTime tipinde bir özellik tanımlanmasına izin veriyoruz.*/
{
    get;set;
}
string Meslek
{
    get;set;
}
void Bul(int KID); /* Bul metod, KID parametresine göre, tablodan ilgili satıra ait
verileri alacak ve alanlara karşılık gelen özelliklere atayacak metodumuzdur.*/
}

```

Şimdide bu arayüzümüzü uygulayacağımız sınıfımızı oluşturalım. Sınıfımız IKisi arayüzünde tanımlanan her üyeyi uygulamak zorundadır. Bu bildiğiniz gibi arayüzlerin bir özelliğidir.

```

public class CKisi:IKisi /* IKisi arayüzünü uyguluyoruz.*/
{
    /* Öncelikle sınıftaki özelliklerimiz için, verilerin tutulacağı alanları tanımlıyoruz.*/
    private int kisiID;
    private string ad;
    private string soyad;
    private DateTime dogumTarihi;
    private string meslek;
    /* Arayüzümüzde yer alan üyeleri uygulamaya başlıyoruz.*/
    public int KisiID
    {
        get{ return kisiID;}
    }
    public string Ad
    {
        get{return ad;}set{ad=value;}
    }
    public string Soyad
    {
        get{return soyad;}set{soyad=value;}
    }
}

```

```

    }
    public DateTime DogumTarihi
    {
        get{return dogumTarihi;}set{dogumTarihi=value;}
    }
    public string Meslek
    {
        get{return meslek;}set{meslek=value;}
    }
    public void Bul(int KID)
    {
        /* Öncelikle Sql Veritabanımıza bir bağlantı açıyoruz.*/
        SqlConnection conFriends=new SqlConnection("data source=localhost;integrated
security=sspi;initial catalog=Friends");
        /* Tablomuzdan, kullanıcının bu metoda parametre olarak gönderdiği KID değerini
        baz alarak, ilgili KisiID'ye ait verileri elde edecek sql kodunu yazıyoruz.*/
        string sorgu="Select * From Kisiler Where KisiID="+KID.ToString();
        /* SqlCommand nesnemiz yardımıyla sql sorgumuzu çalıştırılmak üzere
        hazırlıyoruz.*/
        SqlCommand cmd=new SqlCommand(sorgu,conFriends);
        SqlDataReader rd;/* SqlDataReader nesnemizi yaratıyoruz.*/
        conFriends.Open(); /* Bağlantımızı açıyoruz. */
        rd=cmd.ExecuteReader(CommandBehavior.CloseConnection); /* ExecuteReader
        ile sql sorgumuzu çalıştırıyoruz ve sonuç kümesi ile SqlDataReader nesnemiz arasında bir
        akım(stream) açıyoruz. CommandBehavior.CloseConnection sayesinde, SqlDataReader
        nesnemizi kapattığımızda, SqlConnection nesnemizinde otomatik olarak kapanmasını
        sağlıyoruz.*/
        while(rd.Read())
        {
            /* Eğer ilgili KisiID'ye ait bir veri satırı bulunursa, SqlDataReader nesnemizin
            Read metodu sayesinde, bu satıra ait verileri sınıfımızın ilgili alanlarına aktarıyoruz.
            Böylece, bu alanların atandığı sınıf özellikleride bu veriler ile dolmuş oluyor.*/
            kisiID=(int)rd["KisiID"];
            ad=rd["Ad"].ToString();
            soyad=rd["Soyad"].ToString();
            dogumTarihi=(DateTime)rd["DogumTarihi"];
            meslek=rd["Meslek"].ToString();
        }
        rd.Close();
    }
    public CKisi()
    {
    }
}

```

Artık IKisi arayüzünü uygulayan, CKisi isimli bir sınıfımız var.Şimdi Formumuzun kodlarını yazmaya başlayabiliriz. Öncelikle module düzeyinde bir CKisi sınıf nesnesi tanımlayalım.

```
CKisi kisi=new CKisi();
```

Bu nesnemiz veri tablosundan çektiğimiz veri satırına ait verileri taşıyacak. Kullanıcı Getir başlıklı button kontrolüne bastığında olacak olayları gerçekleştirecek kodları yazalım.

```
private void btnGetir_Click(object sender, System.EventArgs e)
{
    int id=Convert.ToInt32(txtKisiID.Text.ToString()); /* Kullanıcının TextBox kontrolüne girdiği ID değeri Convert sınıfının ToInt32 metodu ile Integer'a çeviriyoruz.*/
    kisi.Bul(id); /* Kisi isimli CKisi sınıfından nesne örneğimizin Bul metodunu çağırıyoruz.*/
    Doldur(); /* Doldur Metodu, kisi nesnesinin özellik değerlerini, Formumuzdaki ilgili kontrollere alarak, bir nevi veri bağlama işlemini gerçekleştirmiş oluyor.*/
}
```

Şimdide Doldur metodumuzun kodlarını yazalım.

```
public void Doldur()
{
    txtAd.Text=kisi.Ad.ToString(); /* txtAd kontrolüne, kisi nesnemizin Ad özelliğinin şu anki değeri yükleniyor. Yani ilgili veri satırının ilgili alanı bu kontrole bağlanmış oluyor.*/
    txtSoyad.Text=kisi.Soyad.ToString();
    txtMeslek.Text=kisi.Meslek.ToString();
    txtDogumTarihi.Text=kisi.DogumTarihi.ToShortDateString();
    lblKisiID.Text=kisi.KisiID.ToString();
}
```

Evet görüldüğü gibi artık aradığımız kişiye ait verileri formumuzdaki kontrollere yükleyebiliyoruz. Şimdi TextBox kontrollerimizin TextChanged olaylarını kodlayacağız. Burada amacımız, TextBox'larda meydana gelen değişikliklerin anında, CKisi sınıfından türettiğimiz Kisi nesnesinin ilgili özelliklerine yansıtılabilmesi. Böylece yapılan değişiklikler anında nesnemize yansıyacak. Bu nedenle aşağıdaki kodları ekliyoruz.

```
/* Metodumuz bir switch case ifadesi ile, aldığı ozellikAdi parametresine göre, CKisi isimli sınıfımıza ait Kisi nesne örneğinin ilgili özelliklerini değiştiriyor.*/
public void Degistir(string ozellikAdi,string veri)
{
    switch(ozellikAdi)
    {
        case "Ad":
        {
            kisi.Ad=veri;
            break;
        }
        case "Soyad":
        {
            kisi.Soyad=veri;
            break;
        }
        case "Meslek":
```

```

        {
            kisi.Meslek=veri;
            break;
        }
        case "DogumTarihi":
        {
            kisi.DogumTarihi=Convert.ToDateTime(veri);
            break;
        }
    }
}
private void txtAd_TextChanged(object sender, System.EventArgs e)
{
    Degistir("Ad",txtAd.Text);
}
private void txtSoyad_TextChanged(object sender, System.EventArgs e)
{
    Degistir("Soyad",txtSoyad.Text);
}
private void txtDogumTarihi_TextChanged(object sender, System.EventArgs e)
{
    Degistir("DogumTarihi",txtDogumTarihi.Text.ToString());
}
private void txtMeslek_TextChanged(object sender, System.EventArgs e)
{
    Degistir("Meslek",txtMeslek.Text);
}
private void btnGuncelle_Click(object sender, System.EventArgs e)
{
    int id;
    id=Convert.ToInt32(lblKisiID.Text.ToString());
    Guncelle(id);
}

```

Görüldüğü gibi kodlarımız gayet basit. Şimdi güncelleme işlemlerimizi gerçekleştireceğimiz kodları yazalım. Kullanıcımız, TextBox kontrollerinde yaptığı değişikliklerin veritabanınada yansıtılmasını istiyorsa Guncelle başlıklı button kontrolüne tıklayacaktır. İşte kodlarımız.

```

private void btnGuncelle_Click(object sender, System.EventArgs e)
{
    /* Güncelleme işlemi, şu anda ekranda olan Kişi için yapılacağından, bu kişiye ait KisiID sini ilgili Lab
    alıyoruz ve Guncelle isimli metodumuza parametre olarak gönderiyoruz. Asıl güncelleme işlemi Guncelle
    yapılıyor. */
    int id;
    id=Convert.ToInt32(lblKisiID.Text.ToString());
    Guncelle(id);
}

```

```

public void Guncelle(int ID)
{
    /* Sql Server'ımıza bağlantımızı oluşturuyoruz.*/
    SqlConnection conFriends=new SqlConnection("data source=localhost;integrated security=sspi;initial catalog=ArkadasVeritabani;")
    /* Update sorgumuzu oluşturuyoruz. Dikkat edecek olursanız alanlara atanacak değerler, kisi isimli nesne
    değerleridir. Bu özellik değerleri ise, TextBox kontrollerinin TextChanged olaylarına eklediğimiz kodlar ile
    tutulmaktadır. En ufak bir değişiklik dahi buraya yansıyabilecektir.*/
    string sorgu="Update Kisiler Set
    Ad='"+kisi.Ad+"',Soyad='"+kisi.Soyad+"',Meslek='"+kisi.Meslek+"',DogumTarihi='"+kisi.DogumTarihi+"'
    Where KisiID="+ID;
    SqlCommand cmd=new SqlCommand(sorgu,conFriends); /* SqlCommand nesnemizi sql cümleciğimizi
    bağlantımız ile oluşturuyoruz. */
    conFriends.Open(); /* Bağlantımızı açıyoruz.*/
    try
    {
        cmd.ExecuteNonQuery(); /* Komutumuzu çalıştırıyoruz.*/
    }
    catch
    {
        MessageBox.Show("Başarısız");
    }
    finally /* Update işlemi herhangi bir neden ile başarısız olsada, olmasada sonuç olarak(finally) açık olan
    bağlantımızı kapatıyoruz. */
    {
        conFriends.Close();
    }
}

```

İşte uygulama kodlarımız bu kadar. Şimdi gelin uygulamamızı çalıştırıp deneyelim. Öncelikle KisiID değeri 1000 olan satıra ait verileri getirelim.

Şekil 3. KisiID=1000 Kaydına ait veriler Kisi nesnemize yüklenir.

Şimdi verilerde bir kaç değişiklik yapalım ve güncelleyelim. Ben Ad alanında yer alan "S." değerini "Selim" olarak değiştirdim. Bu durum sonucunda yapılan değişikliklerin veritabanına yazılıp yazılmadığını ister programımızdan tekrar 1000 nolu satırı getirerek bakabiliriz istersekde Sql Server'dan direkt olarak bakabiliriz. İşte sonuçlar.

	KisiID	Ad	Soyad	DogumTarihi	Meslek
►	1000	Burak Selim	Senyurt	12.04.1976	Matematik Müh.
*					

Şekil 4. Güncelleme işleminin sonucu.

Programımız elbette gelişmeye çok, ama çok açık. Örneğin kodumuzda hata denetimi yapmadığımız bir çok ölü nokta var. Bunların geliştirilmesini siz değerli okurlarımıza bırakıyorum. Bu makalemizde özetle, bir arayüzü bir sınıfa nasıl uyguladığımızı, bu arayüzü nasıl yazdığımızı hatırlamaya çalıştık. Ayrıca, sınıfımıza ait bir nesne örneğine, bir tablodaki belli bir veri satırına ait verileri nasıl alabileceğimizi, bu nesne özelliklerinde yaptığımız değişiklikleri tekrar nasıl veri tablosuna gönderebileceğimizi inceledik. Böylece geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşmek dileğiyle hepinize mutlu günler dilerim.

Arayüz(Interface), Sınıf(Class) ve Çoklu Kalıtım

Bugünkü makalemizde, arayüzleri incelemeye devam edeceğiz. Arayüzlerin anlatıldığı ilk makalemizde, arayüzleri kullanmanın en büyük nedenlerinden birisinin sınıflara çoklu kalıtım desteği vermesi olduğunu söylemiştik. İşte bu makalemizde, arayüzlerin, sınıflara çoklu kalıtım desteğini nasıl sağladığını çok basit bir şekilde incelemeye çalışacağız. Hiç vakit kaybetmeden, basit bir uygulama üzerinde bunu gösterelim. Bu uygulamamızda, sınıfımıza, tanımlamış olduğumuz iki arayüzü uygulayacağız. Böylece sınıfımız bu iki arayüzü kalıtsal olarak almış, çoklu kalıtımı uygulamış olacak.

```
using System;
namespace Interfaces2
{
    public interface IMusteri /* İlk arayüzümüzü tanımlıyoruz. */
    {
        void MusteriDetay();
        int ID{get;}
        string Isim{get;set;}
        string Soyisim{get;set;}
        string Meslek{get;set;}
    }
    public interface ISiparis /* İkinci arayüzümüzü tanımlıyoruz. */
    {
        int SiparisID{get;}
        string Urun{get;set;}
        double BirimFiyat{get;set;}
        int Miktar{get;set;}
        void SiparisDetay();
    }
    public class Sepet:IMusteri,ISiparis /* Sepet isimli sınıfımız hem IMusteri
    arayüzünü hemde ISiparis arayüzünü uygulayacaktır. */
    {
        private int id,sipId,mkt;
        private string ad,soy,mes,ur;
        private double bf;
        public int ID
        {
            get{return id;}
        }
        public string Isim
        {
            get{return ad;}
            set{ad=value;}
        }
        public string Soyisim
        {
            get{return soy;}
        }
    }
}
```



```

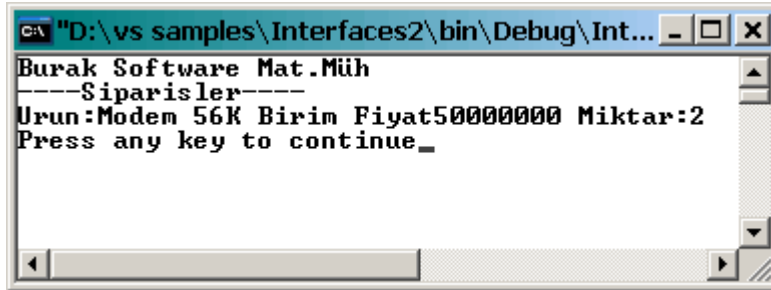
        set{soy=value;}
    }
    public string Meslek
    {
        get{return mes;}
        set{mes=value;}
    }
    public void MusteriDetay()
    {
        Console.WriteLine(ad+" "+soy+" "+mes);
    }
    public int SiparisID
    {
        get{return sipId;}
    }
    public string Urun
    {
        get{return ur;}
        set{ur=value;}
    }
    public double BirimFiyat
    {
        get{return bf;}
        set{bf=value;}
    }
    public int Miktar
    {
        get{return mkt;}
        set{mkt=value;}
    }
    public void SiparisDetay()
    {
        Console.WriteLine("----Siparisler----");
        Console.WriteLine("Urun:"+ur+" Birim Fiyat"+bf.ToString()+
Miktar:"+mkt.ToString());
    }
}
class Class1
{
    static void Main(string[] args)
    {
        Sepet spt1=new Sepet();
        spt1.Isim="Burak";
        spt1.Soyisim="Software";
        spt1.Meslek="Mat.Müh";
        spt1.Urun="Modem 56K";
        spt1.BirimFiyat=50000000;
        spt1.Miktar=2;
    }
}

```

```

        spt1.MusteriDetay();
        spt1.SiparisDetay();
    }
}

```



Şekil 1. Programın Çalışmasının Sonucu.

Yukarıdaki kodlarda aslında değişik olarak yaptığımız bir şey yok. Sadece oluşturduğumuz arayüzleri bir sınıfa uyguladık ve çoklu kalıtımlılığı gerçekleştirmiş olduk. Ancak bu noktada dikkat etmemiz gereken bir unsur vardır. Eğer arayüzler aynı isimli metodlara sahip olurlarsa ne olur? Bu durumda arayüzlerin uygulandığı sınıfta, ilgili metodu bir kez yazmamız yeterli olacaktır. Söz gelimi, yukarıdaki örneğimizde, Baslat isimli ortak bir metodun arayüzlerin ikisi içinde tanımlanmış olduğunu varsayalım.

```

public interface IMusteri
{
    void MusteriDetay();
    int ID{get;}
    string Isim{get;set;}
    string Soyisim{get;set;}
    string Meslek{get;set;}
    void Baslat();
}

public interface ISiparis
{
    int SiparisID{get;}
    string Urun{get;set;}
    double BirimFiyat{get;set;}
    int Miktar{get;set;}
    void SiparisDetay();
    void Baslat();
}

```

Şimdi bu iki arayüzde aynı metod tanımına sahip. Sınıfımızda bu metodları iki kez yazmak anlamsız olacaktır. O nedenle sınıfımıza aşağıdaki gibi tek bir Baslat metodu ekleriz. Sınıf nesnemizi oluşturduğumuzda, Baslat isimli metodu aşağıdaki gibi çalıştırabiliriz.

```
spt1.Baslat();
```

Fakat bazı durumlarda, arayüzlerdeki metodlar aynı isimli olsalar, arayüzlerin uygulandığı sınıf içerisinde söz konusu metod, arayüzlerin her biri için ayrı ayrı yazılmak istenebilir. Böyle bir durumda ise sınıf içerisindeki metod yazımlarında arayüz isimlerini de belirtiriz.Örneğin;

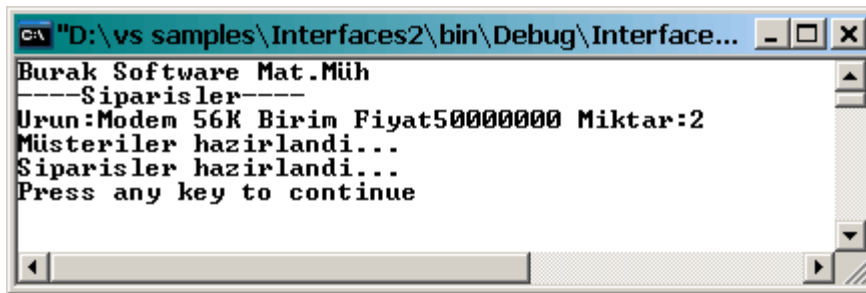
```
void IMusteri.Baslat()
{
    Console.WriteLine("Müşteriler hazırlandı...");
}
void ISiparis.Baslat()
{
    Console.WriteLine("Siparişler hazırlandı...");
}
```

Metodların isimleri başında hangi arayüz için yazıldıklarına dikkat edelim. Diğer önemli bir nokta public belirtecini kullanılmayıdır. Public belirtecini kullanmamız durumunda, "The modifier 'public' is not valid for this item" (public belirleyicisi bu öğe için geçerli değildir) derleyici hatasını alırız. Çünkü, metodumuzu public olarak tanımlamaya gerek yoktur. Nitekim, bu metodların kullanıldığı sınıflara ait nesnelerden, bu metodları çağırmak istediğimizde doğrudan çağıramadığımızı görürüz. Çünkü derleyici hangi arayüzde tanımlanmış metodun çağırılması gerektiğini bilemez. Bu metodları kullanabilmek için, nesne örneğini ilgili arayüz tiplerine dönüştürmemiz gerekmektedir. Bu dönüştürmenin yapılması ilgili sınıf nesnesinin, arayüz tipinden değişkenlere açık bir şekilde dönüştürülmesi ile oluşur. İşte bu yüzden bu tip metodlar, tanımlandıkları sınıf içinde public yapılamazlar. Bu açıkça dönüştürme işlemide aşağıdaki örnek satırlarda görüldüğü gibi olur.

```
IMusteri m=(IMusteri)spt1;
ISiparis s=(ISiparis)spt1;
```

İşte şimdi istediğimiz metodu, bu değişken isimleri ile birlikte aşağıdaki örnek satırlarda olduğu gibi çağırabiliriz.

```
m.Baslat();
s.Baslat();
```



Şekil 2. Programın Çalışmasının Sonucu.

Gördüğümüz gibi arayüzleri yazmak ve sınıflara çoklu kalıtımı amacıyla uygulamak son

derece kolay. Böylece geldik bir makalemizin daha sonuna. İlerleyen makalelerimizde arayüzleri incelemeye devam edeceğiz. Hepinize mutlu günler dilerim.

Arayüzler'de is ve as Anahtar Sözcüklerinin Kullanımı

Bugünkü makalemizde, arayüzlerde **is** ve **as** anahtar kelimelerinin kullanımını inceleyeceğiz. Bir sınıfa arayüz(ler) uyguladığımızda, bu arayüzlerde tanımlanmış metodları çağırmak için çoğunlukla tercih edilen bir teknik vardır. O da, bu sınıfa ait nesne örneğini, çalıştırılacak metodun tanımlandığı arayüz tipine dönüştürmek ve bu şekilde çağırmaktır. Bu teknik, her şeyden önce, program kodlarının okunabilirliğini ve anlaşılabilirliğini arttırmaktadır. Öyleki, bir isim uzayında yer alan çok sayıda arayüzün ve sınıfın yer aldığı uygulamalarda be tekniği uygulayarak, hangi arayüze ait metodun çalıştırıldığı daha kolay bir şekilde gözlemlenebilmektedir. Diğer yandan bu teknik, aynı metod tanımlamalarına sahip arayüzler için de kullanılır ki bunu arayüzlere çoklu kalıtımın nasıl uygulandığını gösteren bir önceki makalemizde işlemiştik.

Bu teknik ile ilgili olarak, dikkat edilmesi gereken bir noktada vardır. Bir sınıfa ait nesne örneğini, bu sınıfa **uygulamadığımız** bir arayüze ait herhangi bir metodu çalıştırmak için, ilgili arayüz tipine dönüştürdüğümüzde **InvalidCastException** istisnasını alırız. Bu noktayı daha iyi vurgulayabilmek için aşağıdaki örneğimizi göz önüne alalım. Bu örnekte iki arayüz yer almakta olup, tanımladığımız sınıf, bu arayüzlerden sadece bir tanesini uygulamıştır. Ana sınıfımızda, bu sınıfa ait nesne örneği, uygulanmamış arayüz tipine dönüştürülmüş ve bu arayüzdeki bir metod çağırılmak istenmiştir.

```
using System;
namespace Interface3
{
    public interface IKullanilmayan
    {
        void Yaz();
        void Bul();
    }
    public interface IKullanilan
    {
        void ArayuzAdi();
    }
    public class ASinifi:IKullanilan
    {
        public void ArayuzAdi()
        {
            Console.WriteLine("Arayüz adl:IKullanilan");
        }
    }
    class Class1
    {
        static void Main(string[] args)
        {
            ASinifi a=new ASinifi();
            IKullanilan Kul=(IKullanilan)a;
            Kul.ArayuzAdi();
            IKullanilmayan anKul=(IKullanilmayan)a;
```

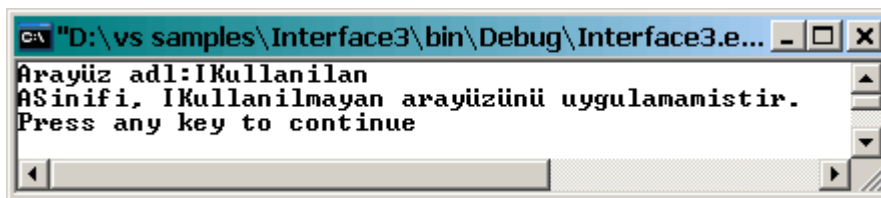
```
        anKul.Yaz();
    }
}
```

Bu örneği derlediğinizde herhangi bir derleyici hatası ile karşılaşmazsınız. Ancak çalışma zamanında **"System.InvalidCastException: Specified Cast Is Invalid"** çalışma zamanı hatasını alırız. İşte bu sorunu **is** veya **as** anahtar sözcüklerinin kullanıldığı iki farklı teknikten birisi ile çözebiliriz. **Is** ve **as** bu sorunun çözümünde aynı amaca hizmet etmekle beraber aralarında önemli iki fark vardır. **Is** anahtar kelimesi aşağıdaki formasyonda kullanılır.

nesne **is** tip

Is anahtar kelimesi nesne ile tipi karşılaştırır. Yani belirtilen nesne ile, bir sınıfı veya arayüzü kıyaslarlar. Bu söz dizimi bir if karşılaştırmasında kullanılır ve eğer nesnenin üretildiği sınıf, belirtilen tip'teki arayüzden uygulanmışsa bu koşullu ifade true değerini döndürecek. Aksi durumda false değerini döndürür. Şimdi bu tekniği yukarıdaki örneğimize uygulayalım. Yapmamız gereken değişiklik **Main** metodunda yer almaktadır.

```
static void Main(string[] args)
{
    ASinifi a=new ASinifi();
    IKullanilan Kul=(IKullanilan)a;
    Kul.ArayuzAdi();
    if(a is IKullanilmayan)
    {
        IKullanilmayan anKul=(IKullanilmayan)a;
        anKul.Yaz();
    }
    else
    {
        Console.WriteLine("ASinifi, IKullanilmayan arayüzünü uygulamamıştır.");
    }
}
```



Şekil 1: **is** Anahtar Kelimesinin Kullanımı.

If koşullu ifadesinde, **a** isimli nesneyi oluşturduğumuz **ASinifi** sınıfına, **IKullanilmayan** arayüzünü uygulayıp uyguladığımızı kontrol etmekteyiz. Sonuç false değerini döndürecek. Nitekim, **ASinifi** sınıfına, **IKullanilmayan** arayüzünü uygulamadık.

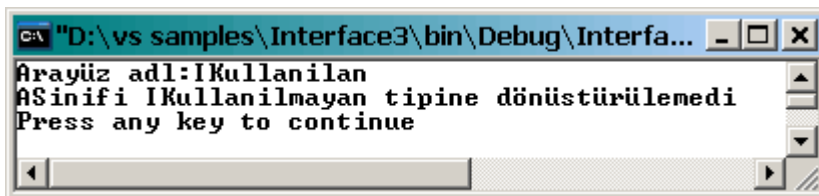
Is anahtar sözcüğü arayüzler dışında sınıflar içinde kullanabiliriz. Bununla birlikte **is** anahtar sözcüğü kullanıldığında, program kodu Intermediate Language (IL) (Ortak Dile) çevrildiğinde, yapılan denetlemenin iki kere tekrar edildiğini görürüz. Bu verimliliği düşürücü bir etkidir. İşte **is** yerine **as** anahtar sözcüğünü tercih etmemizin nedenlerinden biriside budur. Diğer taraftan **is** ve **as** teknikleri, döndürdükleri değerler bakımından da farklılık gösterir. **Is** anahtar kelimesi, bool tipinde true veya false değerlerini döndürür. **As** anahtar kelimesi ise, bir nesneyi, bu nesne sınıfına uygulanmış bir arayüz tipine dönüştürür. Eğer nesne sınıfı, belirtilen arayüzü uygulamamışsa, dönüştürme işlemi yine de yapılır, fakat dönüştürülmenin aktarıldığı değişken null değerine sahip olur. **As** anahtar kelimesinin formu aşağıdaki gibidir.

sınıf nesne1=nesne2 as tip

Burada eğer nesneye belirtilen tipi temsil eden arayüz uygulanmamışsa, nesne null değerini alır. Aksi durumda nesne belirtilen tipe dönüştürülür. İşte **is** ile **as** arasındaki ikinci farkta budur. Konuyu daha iyi anlayabilmek için **as** anahtar kelimesini yukarıdaki örneğimize uygulayalım.

```
static void Main(string[] args)
{
    ASinifi a=new ASinifi();
    IKullanilan Kul=(IKullanilan)a;
    Kul.ArayuzAdi();
    IKullanilmayan anKul=a as IKullanilmayan;
    if(anKul!=null)
    {
        anKul.Yaz();
    }
    else
    {
        Console.WriteLine("ASinifi IKullanilmayan tipine dönüştürülemedi");
    }
}
```

Burada a nesnemiz ASinifi sınıfının örneğidir. **As** ile bu örneği IKullanilmayan arayüzü tipinden anKul değişkenine aktarmaya çalışıyoruz. İşte bu noktada, ASinifi, IKullanilmayan arayüzünü uygulamadığı için, anKul değişkeni null değerini alacaktır. Sonra if koşullu ifadesi ile, anKul 'un null olup olmadığı kontrol ediyoruz. Uygulamayı çalıştırdığımızda aşağıdaki sonucu elde ederiz.



Şekil 2: as Anahtar Kelimesinin Kullanımı

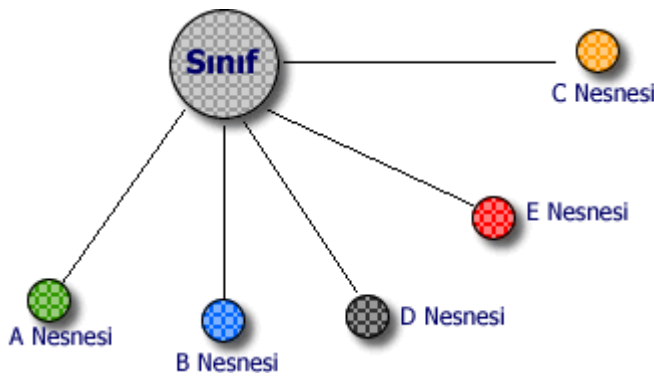
Geldik bir makalemizin daha sonuna. Bir sonraki makalemizde görüşmek dileğiyle hepimize mutlu günler dilerim.

Sınıf (Class) Kavamına Giriş

Bu makalemizde ADONET kavramı içerisinde sınıfları nasıl kullanabileceğimizi incelemeye çalışacak ve sınıf kavramına kısa bir giriş yapacağız. Nitekim C# dili tam anlamıyla nesne yönelimli bir dildir. Bu dil içerisinde sınıf kavramının önemli bir yeri vardır. Bu kavramı iyi anlamak, her türlü teknikte, sınıfların avantajlarından yararlanmanızı ve kendinize özgü nesnelere sahip olabilmenizi sağlar. Zaten .net teknolojisinde yer alan her nesne, mutlaka sınıflardan türetilmektedir.

Çevremize baktığımız zaman, çok çeşitli canlılar görürüz. Örneğin çiçekler. Dünya üzerinde kaç tür(cins) çiçek olduğunu bileniniz var mı ? Ama biz bir çiçek gördüğümüzde ona çoğunlukla "Çiçek" diye hitap ederiz özellikle adını bilmiyorsak. Sonra ise bu çiçeğin renginden, yapraklarının şeklinden, ait olduğu türden, adından bahsederiz. Çiçek tüm bu çiçekler için temel bir sınıf olarak kabul edilebilir. Dünya üzerindeki tüm çiçekler için ortak nitelikleri vardır. Her çiçeğin bir renginin(renklerinin) olması gibi. İşte nesne yönelimli programlama kavramında bahsedilen ve her şeyin temelini oluşturan sınıf kavramı bu benzetme ile tamamen aynıdır. Çiçek bir sınıf olarak algılanırken, sokakta gördüğümüz her çiçek bu sınıfın ortak özelliklerine sahip birer nesne olarak nitelendirilebilir. Ancak tabiki çiçekler arasında da türler mevcuttur. Bu türler ise, Çiçek temel sınıfından türeyen kendi belirli özellikleri dışında Çiçek sınıfının özelliklerindeki kalıtsal olarak alan başka sınıflardır. Bu yaklaşım inheritance (kalıtım) kavramı olarak ele alınır ve nesne yönelimli programlamanın temel üç ögesinden biridir. Kalıtım konusuna ve diğerlerine ilerleyen makalelerimizde değinmeye çalışacağız.

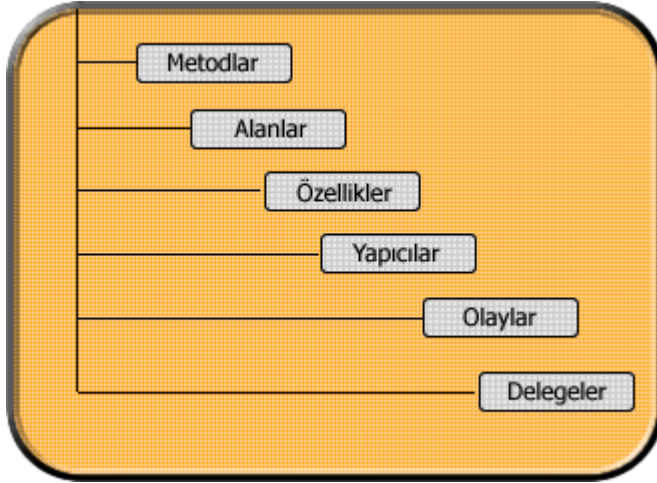
Bugün yapacağımız bir sınıfın temel yapı taşlarına kısaca değinmek ve kendimize ait işimize yarayabilecek bir sınıf tasarlamak. Çiçek sınıfından gerçek C# ortamına geçtiğimizde, her şeyin bir nesne olduğunu görürüz. Ancak her nesne temel olarak Object sınıfından türemektedir. Yani her şeyin üstünde bir sınıf kavramı vardır. Sınıflar, bir takım üyelere sahiptir. Bu üyeler, bu sınıftan örneklenirilen nesneler için farklı değerlere sahip olurlar. Yani bir sınıf varken, bu sınıftan örneklenirilmiş n sayıda nesne oluşturabiliriz. Kaldiki bu nesnelerin her biri, tanımlandığı sınıf için ayrı ayrı özelliklere sahip olabilirler.



Şekil 1. Sınıf (Class) ve Nesne (Object) Kavramı

Bir sınıf kendisinden oluşturulacak nesneler için bir takım üyeler içermelidir. Bu üyeler, alanlar (fields), metodlar (methods), yapıcılar (constructor), özellikler (properties), olaylar(events), delegeler (delegates) vb... dır. Alanlar verileri sınıf içerisinde tutmak amacıyla kullanılırlar. Bir takım işlevleri veya fonksiyonellikleri gerçekleştirmek için metodları kullanırız. Çoğunlukla sınıf içinde yer alan alanların veya özelliklerin ilk değerlerin atanması gibi hazırlık işlemlerinde ise yapıcılar kullanırız. Özellikler kapsülleme

dediğimiz Encapsulating kavramının bir parçasıdır. Çoğunlukla, sınıf içersinden tanımladığımız alanlara, dışarıdan doğrudan erişilmesini istemeyiz. Bunun yerine bu alanlara erişen özellikleri kullanırız. İşte bu sınıf içindeki verileri dış dünyadan soyutlamaktır yani kapsüllemektir. Bir sınıfın genel hatları ile içereceği üyeleri aşağıdaki şekilde de görebilirsiniz.



Şekil 2 . Bir sınıfın üyeleri.

Sınıflar ile ilgili bu kısa bilgilerden sonra derseniz sınıf kavramını daha iyi anlamamızı sağlayacak basit bir örnek geliştirelim. Sınıflar ve üyeleri ile ilgili diğer kavramları kodlar içerisinde yer alan yorum satırlarında açıklamaya devam edeceğiz. Bu örnek çalışmamızda, Sql Suncusuna bağlanırken, bağlantı işlemlerini kolaylaştıracak birtakım üyeler sağlayan bir sınıf geliştirmeye çalışacağız. Kodları yazdıkça bunu çok daha iyi anlayacaksınız. İşte bu uygulama için geliştirdiğimiz, veri isimli sınıfımızın kodları.

```
using System;
using System.Data.SqlClient;
namespace Veriler /* Sınıfımız Veriler isimli isim uzayında yer alıyor. Çoğu zaman aynı
isme sahip sınıflara sahip olabiliriz. İşte bu gibi durumlarda isim uzayları bu sınıfların
birbirinden farklı olduğunu anlamamıza yardımcı olurlar.*/
{
    public class Veri /* Sınıfımızın adı Veri */
    {
        /* İzleyen satırlarda alan tanımlamalarının yapıldığını görmekteyiz. Bu alanlar
        private olarak tanımlanmıştır. Yani sadece bu sınıf içersinden erişilebilir ve değerleri
        değiştirilebilir. Bu alanları tanımladığımız özelliklerin değerlerini tutmak amacıyla
        tanımlıyoruz. Amacımız bu değerlere sınıf dışından doğrudan erişilmesini engellemek.*/
        private string SunucuAdi;
        private string VeritabaniAdi;
        private string Kullanici;
        private string Parola;
        private SqlConnection Kon; /* Burada SqlConnection tipinden bir değişken
        tanımladık. */
        private bool BaglantiDurumu; /* Sql sunucumuza olan bağlantının açık olup
        olmadığına bakacağız.*/
        private string HataDurumu; /* Sql sunucusuna bağlanırken hata olup olmadığına
```

bakacağız.*/

/* Aşağıda sunucu adında bir özellik tanımladık. Herbir özellik, get veya set bloklarından en az birini içermek zorundadır. */

public string sunucu /* **public** tipteki üyelere sınıf içinden, sınıf dışından veya türetilmiş sınıflardan yani kısaca her yerden erişilebilmektedir.*/

```
{  
    get  
    {
```

return SunucuAdi; /* **Get** ile, sunucu isimli özelliğe bu sınıfın bir örneğinden erişildiğinde okunacak değerin alınabilmesi sağlanır . Bu değer bizim **private** olarak tanımladığımız SunucuAdi değişkeninin değeridir. */

```
    }  
    set  
    {
```

SunucuAdi=**value**; /* **Set** bloğunda ise, bu özelliğe, bu sınıfın bir örneğinden değer atamak istediğimizde yani özelliğin gösterdiği **private** SunucuAdi alanının değerini değiştirmek için kullanırız. Özelliğe sınıf örneğinden atanan değer, **value** olarak taşınmakta ve SunucuAdi alanına aktarılmaktadır.*/

```
    }  
}
```

```
public string veritabani  
{
```

```
    get  
    {
```

return VeritabaniAdi;

```
    }  
    set  
    {
```

VeritabaniAdi=**value**;

```
    }  
}
```

public string kullanıcı /* Bu özellik sadece **set** bloğuna sahip olduğu için sadece değer atanabilir ama içeriği görüntülenemez. Yani kullanıcı özelliğini bir sınıf örneğinde, Kullanici **private** alanının değerini öğrenmek için kullanamayız.*/

```
{  
    set  
    {
```

Kullanici=**value**;

```
    }  
}
```

```
public string parola  
{
```

```
    set  
    {
```

Parola=**value**;

```
    }  
}
```

public SqlConnection con /* Buradaki özellik SqlConnection nesne türündendir ve sadece okunabilir bir özelliktir. Nitekim sadece **get** bloğuna sahiptir. */

```

{
    get
    {
        return Kon;
    }
}
public bool baglantiDurumu
{
    get
    {
        return BaglantiDurumu;
    }
    set /* Burada set bloğunda başka kodlar da ekledik. Kullanıcımız bu sınıf
örneği ile bir Sql bağlantısı yarattıktan sonra eğer bu bağlantıyı açmak isterse
baglantiDurumu özelliğine true değerini göndermesi yeterli olacaktır. Eğer false değeri
gönderirse bağlantı kapatılır. Bu işlemleri gerçekleştirmek için ise BaglantiAc ve
BaglantiKapat isimli sadece bu sınıfa özel olan private metodlarımızı kullanıyoruz.*/
    {
        BaglantiDurumu=value;
        if(value==true)
        {
            BaglantiAc();
        }
        else
        {
            BaglantiKapat();
        }
    }
}
public string hataDurumu
{
    get
    {
        return HataDurumu;
    }
}

```

public Veri() /* Her sınıf mutlaka hiç bir parametresi olmayan ve yandaki satırda görüldüğü gibi, sınıf adı ile aynı isme sahip bir metod içerir. Bu metod sınıfın yapıcı metodudur. Yani Constructor metodudur. Bir yapıcı metod içersinde çoğunlukla, sınıf içinde kullanılan alanlara başlangıç değerleri atanır veya ilk atamalar yapılır. Eğer siz bir yapıcı metod tanımlamaz iseniz, derleyici aynen bu metod gibi boş bir yapıcı oluşturacak ve sayısal alanlara 0, mantıksal alanlara false ve string alanlara null başlangıç değerlerini atayacaktır.*/

```

{
}

```

/* Burada biz bu sınıfın yapıcı metodunu aşırı yüklüyoruz. Bu sınıftan bir nesneyi izleyen yapılandırıcı ile oluşturabiliriz. Bu durumda yapıcı metod içerdiği dört parametreyi alıcaktır. Metodun amacı ise belirtilen değerlere göre bir Sql bağlantısı yaratmaktır.*/

```

public Veri(string sunucuAdi,string veritabaniAdi,string kullanıcıAdi,string sifre)
{
    SunucuAdi=sunucuAdi;
    VeritabaniAdi=veritabaniAdi;
    Kullanici=kullanıcıAdi;
    Parola=sifre;
    Baglan();
}
/* Burada bir metod tanımladık. Bu metod ile bir Sql bağlantısı oluşturuyoruz.
Eğer bir metod geriye herhangi bir değer göndermiyecek ise yani vb.net teki
fonksiyonlar gibi çalışmayacak ise void olarak tanımlanır. Ayrıca metodumuzun sadece
bu sınıf içerisinde kullanılmasını istediğimiz için private olarak tanımladık. Bu sayede bu
sınıf dışından örneğin formumuzdan ulaşamamalarını sağlamış oluyoruz.*/
private void Baglan()
{
    SqlConnection con=new SqlConnection("data source="+SunucuAdi+";initial
catalog="+VeritabaniAdi+";user id="+Kullanici+";password="+Parola);
    Kon=con;
}
/* Bu metod ile Sql sunucumuza olan bağlantıyı açıyoruz ve BaglantiDurumu
alanına true değerini aktarıyoruz.*/
private void BaglantiAc() /* Bu metod private tanımlanmıştır. Çünkü sadece bu
sınıf içerisinde çağırılabilir istiyoruz. */
{
    Kon.Open();
    try
    {
        BaglantiDurumu=true;
        HataDurumu="Baglanti sağlandı";
    }
    catch(Exception h)
    {
        HataDurumu="Baglanti Sağlanamadı. "+h.Message.ToString();
    }
}
/* Bu metod ilede Sql bağlantımızı kapatıyor ve BaglantiDurumu isimli alanımıza
false değerini aktarıyoruz.*/
private void BaglantiKapat()
{
    Kon.Close();
    BaglantiDurumu=false;
}
}
}

```

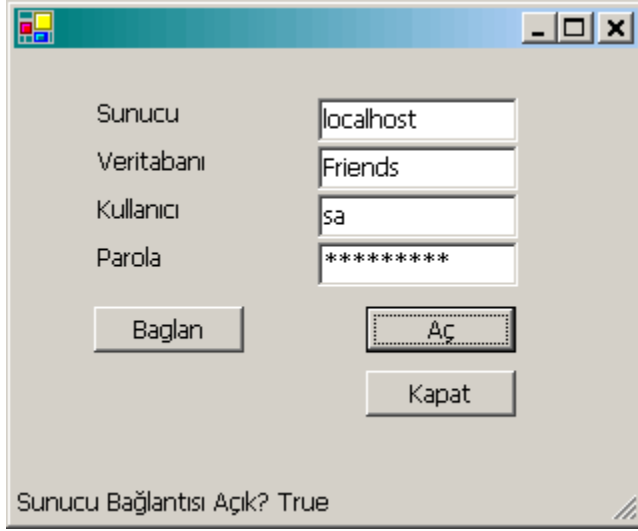
Şimdi ise sınıfımızı kullandığımız örnek uygulama formunu tasarlayalım . Bu uygulamamız aşağıdaki form ve kontrollerinden oluşuyor.

Şekil 3. Form Tasarımımız.

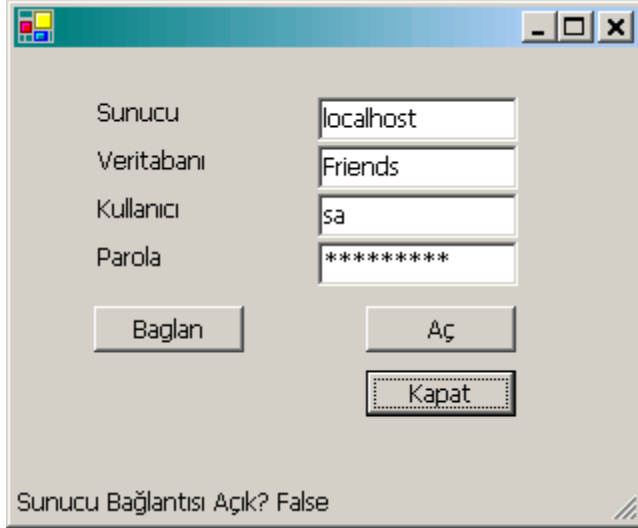
Formumuza ait kodlar ise şöyle.

```
Veriler.Veri v;  
private void btnBaglan_Click(object sender, System.EventArgs e)  
{  
    /* Bir sınıf örneği yaratmak için new anahtar kelimesini kullanırız. New anahtar  
    kelimesi bize kullanabileceğimiz tüm yapıcı metodları gösterecektir. (IntelliSense  
    özelliği). */  
    v=new Veri(txtSunucu.Text,txtVeritabanı.Text,txtKullanıcı.Text,txtParola.Text);  
    Veriler.Veri v=new Veri(  
        ▲ 2 of 2 ▼ Veri.Veri (string sunucuAdi, string veritabanıAdi, string kullanıcıAdi, string sifre)  
    )  
}  
private void btnAc_Click(object sender, System.EventArgs e)  
{  
    v.baglantiDurumu=true;  
    stbDurum.Text="Sunucu Bağlantısı Açık? "+v.baglantiDurumu.ToString();  
}  
private void btnKapat_Click(object sender, System.EventArgs e)  
{  
    v.baglantiDurumu=false;  
    stbDurum.Text="Sunucu Bağlantısı Açık? "+v.baglantiDurumu.ToString();  
}
```

Şimdi uygulamamızı bir çalıştıralım.



Şekil 5. Bağlantıyı açmamız halinde.



Şekil 6. Bağlantıyı kapatmamız halinde.

Değerli okurlarım, ben bu sınıfın geliştirilmesini size bırakıyorum. Umarım sınıf kavramı ile ilgili bilgilerimizi hatırlamış ve yeni ufuklara yelken açmaya hazır hale gelmişsinizdir. Bir sonraki makalemizde sınıflar arasında kalıtım kavramına bakıcak ve böylece nesneye dayalı programlama terminolojisinin en önemli kavramlarından birini incelemeye çalışsacağız. Hepinize mutlu günler dilerim.

Burak Selim ŞENYURT