ESNEK PROGRAM YAZMA, HATA YAKALAMA

Programlama yaparken, yazmış olduğumuz programı kullanacak olan kullanıcıların tamamen bilgisiz olduklarını varsayarak ve olası her türlü kullanım hatasını göz önünde tutarak kod yazmak, programımızı esnek ve kullanım hatalarına karşı dirençli kılacaktır. Nesne yönelimli tasarım ve programlama yaklaşımı kullanıcı ile etkileşim içerisinde olan ve nispeten büyük ölçekli programlarda tercih edilen bir yaklaşım olduğundan, "esnek program yazma ve hata yakalama" konusunun da bu ders kapsamında önemli bir yere sahip olduğu söylenebilir. Aşağıdaki örneği inceleyelim:

```
using System;

public class Sinif1
{
    static int Bol(int bolunen, int bolen)
    {
        return bolunen / bolen;
    }
    static void Main()
    {
        Console.WriteLine("1. ISLEMIN SONUCU : {0}", Bol(3, 2));
        Console.WriteLine("2. ISLEMIN SONUCU : {0}", Bol(2, 3));
        Console.WriteLine("3. ISLEMIN SONUCU : {0}", Bol(3, 0));
        // 3. islemde calisma ani hatasi olacaktir: sayi/0

        Console.ReadLine();
    }
}
```

Yukarıdaki programda, argüman olarak aldığı **int** tipindeki iki tamsayıdan birincisinin ikincisi ile bölümünden elde edilen sonucu döndüren **Bol** fonksiyonu tanımlanmıştır (Tamsayı bölmesi yapıldığı için 3/2 işleminin sonucu 1, 2/3 işleminin sonucu ise 0 olacaktır.). Ancak **Main** fonksiyonu içerisinde **Bol** fonksiyonunun 3. çağrılışında fonksiyonun 2. argümanı **0** olarak verilmiştir. Matematikte bir sayının sıfırla bölünmesi "belirsizlik" anlamı taşıdığından, yukarıdaki program bu fonksiyon çağrısının yapıldığı anda çalışma anı hatası verecektir. Aşağıdaki programda ise bu sorun giderilmiştir:

```
using System;

public class Sinif1
{
    static int Bol(int bolunen, int bolen)
    {
        if (bolen != 0) // Basit bir hata kontrolu
            return bolunen / bolen;
        else
        {
            Console.WriteLine("Bolen sayi sifir olamaz!"); // uyari
            return bolunen / (bolen - 1);
        }
    }
```

Çıktı:

```
1. ISLEMIN SONUCU : 1
2. ISLEMIN SONUCU : 0
Bolen sayi sifir olamaz!
3. ISLEMIN SONUCU : -3
```

```
    static void Main()
    {
        Console.WriteLine("1. ISLEMIN SONUCU : {0}", Bol(3, 2));
        Console.WriteLine("2. ISLEMIN SONUCU : {0}", Bol(2, 3));
        Console.WriteLine("3. ISLEMIN SONUCU : {0}", Bol(3, 0));
        Console.ReadLine();
    }
}
```

(devam)

Yukarıda verilen program incelendiğinde, **Bol** fonksiyonu içerisinde şartlı yapılar yardımı ile basit bir hata kontrolü yapıldığı görülecektir. Burada, sadece **bolen** argümanının 0 değerini alması durumunda bir çalışma anı hatası neydana geleceği, diğer durumlarda fonksiyonun sorunsuz çalışacağı öngörülmüş ve **bolen** argümanının 0 değerini alması durumunda fonksiyon ekrana "Bolen sayi 0 olamaz!" yazdırmakta ve sonuç olarak bölünen sayının bölen sayının 1 eksiğine oranını döndürmektedir. **Bu örnekte, çalışma anı hatasına neyin yol açabileceği kolayca anlaşıldığı için basit bir hata kontrolü ile bu sorunun giderildiği söylenebilir.** Aşağıdaki örneği inceleyelim:

```
using System;

public class Sinif1
{
    static int IslemYap(int sayi1, int sayi2)
    {
        int a, b, c, d;

        a = sayi1 + sayi2;
        b = sayi1 - sayi2;
        c = a + b - 2;
        d = a - c - b;
        b = c / d;
        a = (b + 4) / (c - 4);
        c = d / (sayi1 - a);

        return c;
    }

    static void Main()
    {
        Console.WriteLine("1. ISLEMIN SONUCU : {0}", IslemYap(7, 5));
        Console.WriteLine("2. ISLEMIN SONUCU : {0}", IslemYap(2, 6));
        Console.WriteLine("3. ISLEMIN SONUCU : {0}", IslemYap(4, 3));
        // 3. islemde calisma ani hatasi olacaktir: sayi/0
        Console.ReadLine();
    }
}
```

Yukarıdaki programda yer alan **IslemYap** fonksiyonu, **int** tipinde iki argüman almakta, bu argümanlarla alınan değerleri kullanarak çeşitli toplama, çıkarma ve bölme işlemleri sonucunda **int** tipinde bir değer elde ederek bu değeri döndürmektedir. Elbette bu fonksiyon içerisinde yer alan bölme işlemlerinin herhangi birinde de bölen sayı 0 olursa program çalışma anı hatası verecektir. Ancak böylesi bir programda, paydalardan herhangi birinin sıfır olması durumunu kestirmek bir önceki programa göre daha zor olacaktır. Bunun gibi çalışma anı hatasına açık durumlarda, çalışma anı hatasının yerini ve zamanını tahmin etmenin de güç

olduğu göz önünde bulundurulduğunda, **try&catch** bloklarından yararlanmak en mantıklı çözüm olacaktır:

```
using System;

public class Sinif1
{
    static int IslemYap(int sayi1, int sayi2)
    {
        int a, b, c, d;

        try
        { // dene
            a = sayi1 + sayi2;
            b = sayi1 - sayi2;
            c = a + b - 2;
            d = a - c - b;
            b = c / d;
            a = (b + 4) / (c - 4);
            c = d / (sayi1 - a);
        }
        catch
        { // "try" blogunda calisma ani hatasi olursa bu bloktakileri yap
            Console.WriteLine("Sifir ile bolme hatasi kotarildi!");
            c = -1;
        }

        return c;
    }

    static void Main()
    {
        Console.WriteLine("1. ISLEMIN SONUCU : {0}", IslemYap(7, 5));
        Console.WriteLine("2. ISLEMIN SONUCU : {0}", IslemYap(2, 6));
        Console.WriteLine("3. ISLEMIN SONUCU : {0}", IslemYap(4, 3));
        // 3. islemde calisma ani hatasi olacaktir: sayi/0

        Console.ReadLine();
    }
}
```

Çıktı:

```
1. ISLEMIN SONUCU : 0
2. ISLEMIN SONUCU : 2
Sifir ile bolme hatasi kotarildi!
3. ISLEMIN SONUCU : -1
```

try&catch bloğu, olası çalışma anı hatalarının kotarılması (yakalanarak telafi edilmesi) için kullanılır. Çalışma anı hatası vermesi muhtemel olan kod parçası **try** bloğu içerisine yerleştirilirken çalışma anı hatası olması durumunda devreye sokulması istenen kod parçası da **catch** bloğu içerisine yazılır. **try** bloğu içerisinde herhangi bir yerde çalışma anı hatası gerçekleşmesi durumunda **try** bloğundaki kodların çalışma anı hatasının olduğu nokta ve sonrasındakilerden hiçbiri çalıştırılmamış kabul edilerek bunların yerine **catch** bloğundaki kodlar çalıştırılır.

Yazdığımız bir programdaki tüm kodları **try&catch** blokları içerisine almak, ekstra kontrol maliyetlerinden dolayı programımızın çalışma performansını düşürecektir. Ayrıca böylesi bir yaklaşım, programın belirli bir kısmında çalışma anı hatası olması durumunda o hatayı en mantıklı biçimde telafi edebilecek kodların en doğru biçimde belirlenmesini de olanaksız kılacaktır. Bu yüzden en doğru yaklaşım, çalışma anı hatası verebilecek kod bloğunu

belirleyerek sadece bu kısmı **try** bloğu içerisine almak ve bu kısımdaki olası bir çalışma anı hatası durumunda izlenmesi gereken en uygun yolu da **catch** bloğu içerisinde kodlamaktır. **IslemYap** fonksiyonu içerisinde çalışma anı hatası verme potansiyeli olan kodların sadece bölme işlemi yapılan kısımlar olduğu düşünülürse, bu fonksiyonu aşağıdaki gibi gerçeklemek daha uygun olacaktır:

```
...
    static int IslemYap(int sayi1, int sayi2)
    {
        int a, b, c, d;

        a = sayi1 + sayi2;
        b = sayi1 - sayi2;
        c = a + b - 2;
        d = a - c - b;
        try
        { // dene
            b = c / d;
            a = (b + 4) / (c - 4);
            c = d / (sayi1 - a);
        }
        catch
        { // "try" blogunda calisma ani hatasi olursa bu bloktakileri yap
            Console.WriteLine("Sifir ile bolme hatasi kotarildi!");
            c = -1;
        }
        return c;
    }
...
```

Çıktı:

```
1. ISLEMIN SONUCU : 0
2. ISLEMIN SONUCU : 2
Sifir ile bolme hatasi kotarildi!
3. ISLEMIN SONUCU : -1
```

Çalışma anı hatasına sebebiyet veren sıfırla bölme, dosya bulunamaması gibi durumlara "sıra dışı durum (İngilizce: exception)" denmektedir. **System** isim uzayı içerisinde, çeşitli sıra dışı durumları temsil etmek üzere tasarlanmış sınıflar yer almakta olup bu sınıfların nesneleri, sembolik olarak "sıra dışı durum" anlamını taşımaktadır. Örneğin, **System** isim uzayı içerisindeki **DivideByZeroException** sınıfı nesneleri birer sıfır ile bölme sıra dışı durumunu temsil ederler. **catch** deyiminden sonra parantez içerisinde, **try** bloğunda olması muhtemel olan sıra dışı durumun türüne uygun tipte bir sınıf referansı belirtilerek bu referans, **catch** bloğunun içerisinde **throw** deyiminden sonra kullanılabilir. Bu, **try** bloğunda yakalanan sıra dışı durumun **catch** bloğu içerisinde yeniden fırlatılmasıdır. Bu fırlatma işlemi, işletim sistemine "sıfırla bölme hatası" olduğuna dair özel bir sinyal gönderecek ve program da çalışma anı hatası verecektir. **catch** bloğunun bahsedilen biçimde güncellenmiş hali şöyledir:

```
...
        catch(DivideByZeroException dbz)
        { // "try" blogunda calisma ani hatasi olursa bu bloktakileri yap
            Console.WriteLine("Sifir ile bolme hatasi kotarildi!");
            c = -1;
            throw dbz;
        }
...
```

try bloğu içerisinde yakalanan hata ekrana yazdırılmak isteniyorsa **catch** bloğu aşağıdaki gibi güncellenebilir:

```
...
        catch (Exception e)
        { // "try" blogunda calisma ani hatasi olursa bu bloktakileri yap
            Console.WriteLine("{0}", e.ToString());
            c = -1;
        }
...
```

Eğer **try&catch** blokları ve **throw** deyimi kullanılmadan sıfırla bölme işlemi yapılsaydı bu sefer de oluşan sıra dışı durum, bizzat işletim sistemi tarafından karşılanacaktı ve işletim sistemi, Windows için konuşacak olursak, programın çıktısı olan çalıştırılabilir dosyayı (".exe" uzantılı dosyayı) çalıştıramayarak hatalı bir durumla karşılaşıldığını bildirecekti.

```
using System;

public class Sinif1
{
    static void Main()
    {
        throw new DivideByZeroException();
    }
}
```

Yukarıdaki programda olduğu gibi, **try&catch** blokları kullanmadan sıradışı durum nesnesi fırlatarak işletim sistemine sinyal göndermek de mümkündür. Bu program da çalışma anı hatası verecektir.

try&catch bloklarının kullanımı bize çalışma anı hatalarının yakalanarak programın çlışmaya devam etmesine imkân vermesinin yanı sıra programın neresinde çalışma anı hatası yapıldığının anlaşılarak koda yama yapılmasına (olası açıkların giderilmesine) da olanak sağlayacaktır:

```
using System;
using System.IO;

public class Sinif1
{
    public static void fonksiyon1(int arg1, int arg2)
    {
        int sonuc;
        try
        {
            sonuc = arg1 / arg2;
        }
        catch
        {
            Console.WriteLine("Sinif1.fonksiyon1 icerisinde beklenmedik bir durum olustu!");
        }
    }
```

Çıktı:

```
Sinif1.fonksiyon2 icerisinde beklenmedik bir durum olustu!
```

(devam)

```
    public static void fonksiyon2()
    {
        string temp;
        try
        {
            StreamReader sr = new StreamReader("C:/olmayan.txt");
            temp = sr.ReadToEnd();
        }
        catch
        {
            Console.WriteLine("Sinif1.fonksiyon2 icerisinde beklenmedik bir durum olustu!");
        }
    }
}

public class Sinif2
{
    static void Main()
    {
        Sinif1.fonksiyon1(5, 6);
        Sinif1.fonksiyon2();
        Console.ReadLine();
    }
}
```

Çalışma anı hatasına yol açabilecek işlemlerden birisi de tip dönüşümleridir. Aşağıdaki programı inceleyelim:

```
using System;
using System.IO;

public class Sinif1
{
    static void ToplaYaz()
    {
        int toplam = 0;
        // girilen degerler tamsayiya donusturulebillir degilse
        // calisma ani hatasi ile karsilasilacaktir
        Console.Write("Lutfen 1. sayiyi giriniz : ");
        toplam += Convert.ToInt32(Console.ReadLine());
        Console.Write("Lutfen 2. sayiyi giriniz : ");
        toplam += Convert.ToInt32(Console.ReadLine());

        Console.WriteLine("\nTOPLAM : {0}", toplam);
    }
    static void Main()
    {
        ToplaYaz();
        Console.ReadLine();
    }
}
```

Yukarıdaki programda, kullanıcıdan konsol üzerinden aldığı değerleri tamsayıya dönüştürerek bunların toplamını ekrana yazdıran **ToplaYaz** fonksiyonu görülmektedir. **Console.ReadLine** fonksiyonu ile konsoldan alınan her değerin tipi normalde **string**' dir. **string** tipindeki bir verinin **int** tipine dönüştürülebilmesi için, tamsayı formatında olması gerekir. Yukarıdaki programda eğer kullanıcı tamsayı formatında olmayan bir değer girerse, **string** tipinden **int** tipine dönüşüm gerçekleşmeyecek, çalışma anı hatası ile karşılaşılacaktır. Programın çalıştırılmasına dair iki farklı örnek:

Durum 1:

Çalışma anı hatası yok.

```
Lutfen 1. sayiyi giriniz : 3
Lutfen 2. sayiyi giriniz : -7

TOPLAM : -4
```

Durum 2:

```
Lutfen 1. sayiyi giriniz : 6
Lutfen 2. sayiyi giriniz : 8t
```

Çalışma anı hatası var.

Böylesi bir programda çalışma anı hatasının önüne geçmek için **ToplaYaz** fonksiyonu aşağıdaki gibi güncellenmelidir:

```
static void ToplaYaz()
{
    int toplam = 0;

    try
    {
        Console.Write("Lutfen 1. sayiyi giriniz : ");
        toplam += Convert.ToInt32(Console.ReadLine());
        Console.Write("Lutfen 2. sayiyi giriniz : ");
        toplam += Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("\nTOPLAM : {0}", toplam);
    }
    catch
    {
        Console.WriteLine("Girilen degerler tamsayi formatina uygun olmalidir!");
    }
}
```

Burada, kullanıcının hatalı değer girmesi durumunda ekrana toplam sonucu yazılması mümkün olmayacağı için `Console.WriteLine("\nTOPLAM : {0}", toplam);` ifadesi de **try** bloğu içerisine koyulmuştur. Programın güncel halinin çalıştırılmasına dair iki farklı örnek aşağıdaki gibidir:

Durum 1:

```
Lutfen 1. sayiyi giriniz : 3
Lutfen 2. sayiyi giriniz : -7

TOPLAM : -4
```

Çalışma anı hatası yok.

Durum 2:

```
Lutfen 1. sayiyi giriniz : 6
Lutfen 2. sayiyi giriniz : 8t
Girilen degerler tamsayi formatina uygun olmalidir!
```

Çalışma anı hatası yok.

try bloğu kullanıldıktan sonra bu bloğun altına mutlaka bir **catch** bloğu eklenmelidir.

try bloğu içerisinde çalışma anı hatası meydana gelene kadar yapılan işlemler çalışma anı hatasından etkilenmezler. Çalışma anı hatasına sebebiyet veren kod ve bu koddan sonra yazılan dğer kodlar ise çalıştırılmazlar. Program akışı, çalışma anı hatası ile karşılaşıldığı andan itibaren **catch** bloğu içerisinden devam edecektir:

```
using System;

public class Sinif1
{
    static void IslemYap()
    {
        int a = 0, b = 1, c = 2, d = 3;

        Console.WriteLine("[birinci konum] --> c : {0}", c);

        try
        {
            Console.WriteLine("[ikinci konum]  --> c : {0}", c);
            c = 7;
            Console.WriteLine("[ucuncu konum]  --> c : {0}", c);
            d = b / a; // calisma ani hatasi
            Console.WriteLine("[dorduncuuncu konum] --> c : {0}", c);
        }
        catch
        {
            Console.WriteLine("[besinci konum] --> c : {0}", c);
        }
        Console.WriteLine("[altinci konum] --> c : {0}", c);
    }

    static void Main()
    {
        IslemYap();
        Console.ReadLine();
    }
}
```

Çıktı:

```
[birinci konum] --> c : 2
[ikinci konum]  --> c : 2
[ucuncu konum]  --> c : 7
[besinci konum] --> c : 7
[altinci konum] --> c : 7
```

Ekstra Bilgiler

Programı Uyutmak

Yazmış olduğumuz programı uyutmak (dondurmak, işlem yaptırmadan bekletmek) için **System.Threading** alt isim uzayı içerisindeki **Thread** sınıfının **statik** erişimli **Sleep** fonksiyonu kullanılabilir. Bu fonksiyon, milisaniye cinsinden uyunması gereken süreyi argüman olarak (**int** tipinde) alır ve bu süre kadar programı askıda bekletir. Aşağıdaki örnekte, basit bir **C#** programının 7 saniyeliğine uyutulması gerçeklenmiştir:

```
using System;
using System.IO;

public class Sinif1
{
    static void Main()
    {
        Console.WriteLine("Program 7 saniye uyuyacak...");
        System.Threading.Thread.Sleep(7000);
        Console.WriteLine("Program uyandi !");
        Console.ReadLine();
    }
}
```

Sleep fonksiyonundan yararlanılarak gerçeklenen diğer bir uygulama da aşağıda verilmiştir:

```
using System;
using System.IO;

public class Sinif1
{
    static void Main()
    {
        for (int i = 80; i > 0; i--)
        {
            Console.Write("*");
            System.Threading.Thread.Sleep(i * 15);
        }
        Console.ReadLine();
    }
}
```

Haricî Bir Çalıştırılabilir Dosyayı Çalıştırmak (Windows Altında)

Windows altında haricî bir çalıştırılabilir dosyayı (".exe" uzantılı dosyayı) çalıştırmak, ya da kendisi doğrudan çalıştırılabilir olmayan bir dosya aracılığıyla çalıştırılabilir başka bir dosyayı çalıştırmak (bir ".txt" dosyasına tıklayarak bu dosyayı işlemek üzere "notepad.exe" programını harekete geçirmek gibi) için, **System.Diagnostics** alt isim uzayında bulunan **Process** sınıfının **Start** fonksiyonundan yararlanmak gerekir. ".txt" uzantılı ve çalıştırılabilir olmayan bir dosya aracılığı ile "notepad.exe" gibi çalıştırılabilir bir dosyayı harekete geçirmek, ilgili metin dosyasının adresini çalıştırılabilir dosyaya argüman olarak vermek anlamını taşımaktadır. Aşağıda, "**C:/Program Files/Notepad++/**" dizini altında "**notepad++.exe**" isminde çalıştırılabilir bir dosya bulunan ve "**C:/**" dizininde "**metin.txt**" isminde çalıştırılamayan bir dosya bulunan, "notepad" programının yüklü olduğu, Windows işletim sistemine sahip bir bilgisayar için hazırlanmış bir örnek yer almaktadır (Siz de bu örneği kendi bilgisayarınıza özgü biçimde düzenleyebilirsiniz.):

```
using System;
using System.IO;

public class Sinif1
{
    static void Main()
    {
        System.Diagnostics.Process.Start("C:/Program Files/Notepad++/notepad++.exe");
        System.Diagnostics.Process.Start("C:/metin.txt");
    }
}
```

Yukarıdaki programda **Main** fonksiyonunun ilk satırında doğrudan çalıştırılabilir bir dosya olan "notepad++.exe" dosyası çalıştırılmıştır. İkinci satırda ise, normalde çalıştırılabilir olmayan "metin.txt" dosyası **Start** fonksiyonuna verilmiştir. Bu durumda da işletim sistemi, bu dosyanın uzantısından hareketle bu dosyayı birlikte açması gereken uygulamayı (uygulama: çalıştırılabilir dosya) belirleyerek bu dosyanın adresini ilgili uygulamaya argüman olarak vermektedir. Yukarıdaki program için bu durumu özetlemek gerekirse, **Main** fonksiyonunun ikinci satırında yapılan işlem sonucunda "**notepad.exe**" uygulaması *("notepad++.exe" ile karıştırılmamalıdır.)*, "**C:/**" dizini altındaki "**metin.txt**" dosyasını düzenlemek üzere kullanacak, yani bu dosyanın içeriğini kullanıcıya bir ara yüz ile gösterecektir.

Çalışmakta Olan Bir Uygulamayı Sonlandırmak (Windows Altında)

Windows' ta görev yöneticisinde gördüğümüz ve çalışmakta olan uygulamalar, **System.Diagnostic** alt isim uzayında yer alan **Process** sınıfının nesneleri kullanılarak, aşağıdaki örnek programda olduğu biçimde sonlandırılabilmektedir:

```
using System;
using System.IO;
using System.Diagnostics;

public class Sinif1
{
    static bool IslemOldur(string islemAdi)
    {
        foreach (Process proc in Process.GetProcesses())
        {
            if (proc.ProcessName.StartsWith(islemAdi))
            {
                proc.Kill();
                return true;
            }
        }
        return false;
        // Kaynak : http://www.dreamincode.net/code/snippet1543.htm
    }

    static void Main()
    {
        Console.WriteLine("[notepad++.exe] uygulamasi baslatiliyor...");
        System.Diagnostics.Process.Start("C:/Program Files/Notepad++/notepad++.exe");
        Console.WriteLine("[notepad++.exe] uygulamasi 10 saniye calisacak...");
        System.Threading.Thread.Sleep(10000);
        Console.WriteLine("[notepad++.exe] uygulamasi 10 saniye calisti, sonlandirilacak...");
        bool sonuc = IslemOldur("notepad++"); // ".exe" uzantisini yazmiyoruz

        if (sonuc)
            Console.WriteLine("[notepad++.exe] uygulamasi sonlandirildi!");
        else
            Console.WriteLine("[notepad++.exe] uygulamasi sonlandirilamadi!");

        Console.ReadLine();
    }
}
```

Çıktı:

```
[notepad++.exe] uygulamasi baslatiliyor...
[notepad++.exe] uygulamasi 10 saniye calisacak...
[notepad++.exe] uygulamasi 10 saniye calisti, sonlandirilacak...
[notepad++.exe] uygulamasi sonlandirildi!
```

Programın Belli Bir Kısmının İşlenmesi İçin Geçen Süreyi Bulma

System isim uzayı içerisindeki **DateTime** sınıfından yararlanarak, programın belli bir kısmının işlenmesi için geçen süreyi hesaplamak mümkündür:

```
using System;
using System.IO;

public class Sinif1
{
    static void Main()
    {
        DateTime basla, bitir;
        TimeSpan sure;

        int sayac, a = 1, b = 2, c = 3, d = 4, e = 5;

        basla = DateTime.Now;
        for (sayac = 0; sayac < 1000000000; sayac++)
        {
            a = b; b = c; c = d; d = e; e = a;
        }
        bitir = DateTime.Now;

        sure = bitir - basla;

        Console.WriteLine("Gecen Sure : {0}", sure);
        Console.ReadLine();
    }
}
```

Çıktı:

```
Gecen Sure : 00:00:12.7656250
```

Yukarıdaki programda, **for** döngüsünün çalışma süresi ölçülerek ekrana yazdırılmıştır (yaklaşık 12.8 saniye).