X.3. Szálak szinkronizálása

Amíg a szálak végrehajtásánál adatokkal, egyéb függvényhívással kapcsolatos feladataink nincsenek, a szálak egymástól nem zavartatva rendben elvégzik feladataikat. Ez az ideális helyzet viszont ritkán fordul elő.

Tekintsük azt a példát, mikor egy osztály az adatmentés feladatát végzi. (Ezt a példánkban egyszerűen a képernyőre írással valósítjuk meg.)

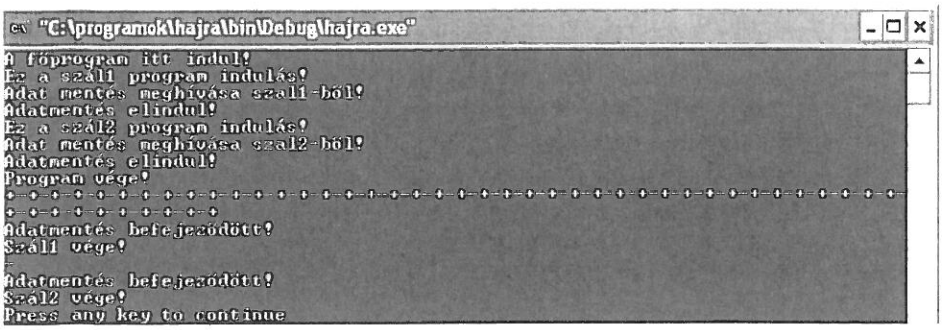
Definiáljunk két szálat, amelyek természetesen a programosztály adatmentését használják. Az előző forráskódot kicsit átalakítva az alábbi programot kapjuk:

Példa:

```
using System;
using System.Threading;

class adatok
{
     public void mentes(string s)
     {
          Console.WriteLine("Adatmentés elindul!");
          for(int i=0;i<50;i++)
          {
               Thread.Sleep(1);
               Console.Write(s);
          }
          Console.WriteLine("");
          Console.WriteLine("Adatmentés befejeződött!");

     }
}

class program
{
     public static adatok a=new adatok();
     // adatmentésmező a programban

     // szál1 definiálás
     public static void szal1()
     {
          Console.WriteLine("Ez a szál1 program indulás!");
          // egy másodpercet várunk
          Console.WriteLine("Adatmentés meghívása szál1-ből!");
          a.mentes("+");
          Console.WriteLine("Szál1 vége!");
     }
```

```
        // szál1 definiálás
        public static void szal2()
        {
            Console.WriteLine("Ez a szál2 programindulás!");
            // egy másodpercet várunk
            Console.WriteLine("Adatmentés meghívása szal2-ből!");
            a.mentes("-");
            Console.WriteLine("Szál2 vége!");
        }

        public static void Main()
        {
            Console.WriteLine("A főprogram itt indul!");
            ThreadStart szalmutato1=new ThreadStart(program.szal1);
            ThreadStart szalmutato2=new ThreadStart(program.szal2);
            // létrehoztuk a fonal függvénymutatót, ami a
            //
            Thread fonal1=new Thread(szalmutato1);
            Thread fonal2=new Thread(szalmutato2);
            fonal1.Start();
            fonal2.Start();
            //
            //
            Console.WriteLine("Program vége!");
        }
}
```

A programot futtatva az alábbi eredményt kapjuk:



13. ábra

Ebből a futási e
azaz ugyanannak a f
(Csak azért kerü
a párhuzamos függv
A + és a – karak
végrehajtása nem j
esetre, amikor az
archiválási céllal, v
„sorrendje", és ez a
többszálú végrehajt
megszakításmentes
nizált végrehajtását
Ha egy adat m
végrehajtási szálna
kínálkozik. Csak a
tudjuk, és nem is c
Automatikus s
kor egy egész osz
megtennünk:

1. A *Synchr*
tumokkal
el ezt az o
2. Az osztály

Példa:

```
[Syncroniz
class szir
{
    int i
    publi

    {

    }
...
}
```

Az automat
elérhetik. Ezt o
manuális szinkr

Ebből a futási eredményből az látszik, hogy a *fonal1* és a *fonal2* mentése, azaz ugyanannak a függvénynek a végrehajtása párhuzamosan történik!

(Csak azért került egy kis várakozás a ciklusba, hogy szemléletesebb legyen a párhuzamos függvényfutás, a + és – karakterek váltott kiírása.)

A + és a – karakterek váltakozó kiírása, azaz a két fonal törzsének váltakozó végrehajtása nem jár különösebb gonddal. De gondoljunk például egy olyan esetre, amikor az adatmentő függvény soros portra írja a mentési adatokat archiválási céllal, vagy vezérel valamilyen eszközt. Ekkor lényeges az adatok „sorrendje", és ez a fajta futási eredmény nem kielégítő. Tehát alapvető igény a többszálú végrehajtás esetén, hogy biztosítani tudjuk egy függvény, egy utasítás megszakításmentes (ezt gyakran *thread safe* lehetőségnek nevezik), ún. szinkronizált végrehajtását.

Ha egy adat módosítását, egy függvény hívását egy időpontban csak egy végrehajtási szálnak szeretnénk engedélyezni, akkor erre több lehetőségünk is kínálkozik. Csak a leggyakrabban használt lehetőségeket említjük, hiszen nem tudjuk, és nem is célunk az összes lehetőség referenciaszerű felsorolása.

Automatikus szinkronizációnak nevezi a szakirodalom azt a lehetőséget, mikor egy egész osztályra „beállítjuk" ezt a szolgáltatást. Ehhez két dolgot kell megtennünk:

1. A *Synchronization*() attribútummal kell jelölni az osztályt. Az attribútumokkal a következő fejezet foglalkozik, így most egyszerűen fogadjuk el ezt az osztályra, függvényre, változóra állítható információt.
2. Az osztályt a *ContextBoundObject* osztályból kell származtatni.

Példa:

```
[Syncronization()]
class szinkronosztály: ContextBoundObject
{
    int i=5;                // egy időbencsak egy szál fér hozzá
    public void Novel()     // ezt a függvényt is csak egy szál
                            // tudja egyszerre végrehajtani
    {
        i++;
    }
...
}
```

Az automatikusan szinkronizált osztályoknál a statikus mezőket többen is elérhetik. Ezt orvosolja a függvények, blokkok szinkronizációja, amit gyakran manuális szinkronizációnak is neveznek.

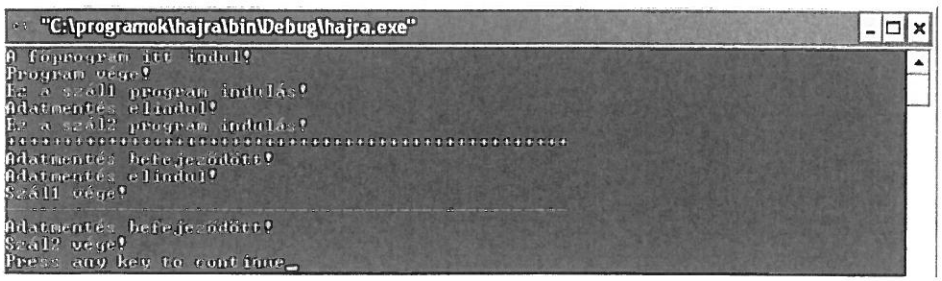
Egy függvény szinkronizált végrehajtását a legegyszerűbben a *MethodImplAttribute* attribútum beállításával érhetjük el. Ezt mind példány, mind pedig statikus függvényre alkalmazhatjuk.

```
[MethodImplAttribute(MethodImplOptions.Syncronized)]
public void safe_fv()
{
    ...
}
```

Egy függvény szinkronizált végrehajtására kínál egy másik megoldást a *Monitor* osztály *enter* és *exit* függvénye. Amíg egy monitor a belépéssel véd egy végrehajtási blokkot, addig egy másik szál azt nem tudja meghívni. Ekkor a következőképpen módosul az adatmentés függvénye:

```
public void mentes(string s)
{
    Monitor.Enter(this);

        Console.WriteLine("Adatmentés elindul!");
        for(int i=0;i<50;i++)
        {
            Thread.Sleep(1);
            Console.Write(s);
        }
        Console.WriteLine("");
        Console.WriteLine("Adatmentés befejeződött!");
    Monitor.Exit(this);
}
```

Az eredményben azt láthatjuk, hogy az a blokk, amit a *Monitor* véd, megszakítás nélkül fut le.



14. ábra

Hasonló eredményt kaphatunk, ha a C# nyelv *lock* utasításával védjük a kívánt blokkot. Ha a fenti függvényt a *lock* nyelvi utasítással védjük, azt a fordító a következő alakra alakítja:

```
Monitor.Enter(this);
try
{
    utasítások;
}
finally
{
    Monitor.Exit(this);
}
```

A teljesség igénye nélkül a *Monitor* osztály két függvényéről még szót kell ejtenünk. Az egyik a *Wait*, ami a nevéből sejthetően leállítja a szál végrehajtását, és a paraméterobjektum zárását befejezi.

```
Monitor.Wait(objektum);
```

A függvény használathoz a *Pulse* függvény is hozzátartozik, ami az objektumra várakozó szálat továbbengedi.

```
Monitor.Pulse(objektum);
```

Hasonló szolgáltatást ad a Windows API *mutex* (*mutual exclusive*, kölcsönös kizárás) lehetőségének megfelelő *Mutex* osztály. Lényeges különbség a monitorhoz képest, hogy a kizárólagos végrehajtáshoz megadhatjuk azt is, hogy ez mennyi ideig álljon fenn. (*WaitOne* függvény)

```
class adatok
{   Mutex m=new Mutex();
    public void mentes(string s)
    {
        m.WaitOne();
            Console.WriteLine("Adatmentés elindul!");
            for(int i=0;i<50;i++)
            {
                Thread.Sleep(1);
                Console.Write(s);
            }
            Console.WriteLine("");
            Console.WriteLine("Adatmentés befejeződött!");
        m.Close();
    }
}
```

Még manuálisabb adat vagy utasítás szinkronizációt biztosít a *ReadWriterLock*, valamint az *Interlocked* osztály is. Ezek és a további szinkronizációs lehetőségek ismertetése túlmutat e könyv keretein.

Ha könyvtári szolgáltatásokat veszünk igénybe, például MSDN Help, az egyes hívások, osztályok mellett olvashatjuk, hogy *thread safe*-e vagy nem a használata, attól függően, hogy a szinkronizált hozzáférés biztosított-e vagy sem.

A szálakkal kapcsolatban befejezésül meg kell említeni a *[STAThread]* illetve az *[MTAThread]* attribútumot, ami azt mondja meg, hogy *Single* (egy) vagy *Multi* (több) szálat tartalmazó alkalmazásként definiáljuk a programunkat COM komponensként való együttműködésnél. Alapértelmezés a [STAThread] használata, ami például a Windows alkalmazások „Drag and Drop" jellemzők használata esetén követelmény is.

X.4. Feladatok

1. Mit értünk párhuzamos programvégrehajtás alatt?
2. Hogyan tudunk szálat definiálni?
3. Mit jelent a *lock* utasítás? Milyen könyvtári szolgáltatás felel meg ennek az utasításnak?
4. Készítsen programot, amely két süket ember beszélgetését szimulálja! Mindketten egy-egy fájlban tárolják a mondókájukat!
5. Használjunk szinkronizálást, módosítsuk az előző feladatot úgy, hogy a mondatvégeknél lehet a másik szereplőnek átvenni a beszéd fonalát!

XI. Attri

Képzeljük el,
függvénymezőnek
azaz a típusunk h
pus mellé bizonyo
Példaként néz
is előforduló esete

Első példa:
Egy program
között lehet néh
szükségünk lehet
korábbi Window
tudunk ebben az
innen be is olvas
is, a visszaolvas
jelenleg kik azo
válasz! Minden
adataink! Haszn
a mezőket, és ez

Második pél
Ez a példa
életszerű. Defin
léteznek – az er
normális haszn
életkor, végzett
mációra, hogy
donságok közé
tényleges műk
rendesen a dolg

Az ilyen pl
attribútum.
Az attribú
osztályokhoz,
Ezek az inform

izációt biztosít a
Ezek és a további
retein.
iul MSDN Help, az
l *safe*-e vagy nem a
ztosított-e vagy sem.
teni a *[STAThread]*
g, hogy *Single* (egy)
iljuk a programunkat
ezés a [STAThread]
and Drop" jellemzők

tatás felel meg ennek

élgetését szimulálja!
t!

feladatot úgy, hogy a
a beszéd fonalát!

XI. Attribútumok

Képzeljük el, hogy definiáltunk egy új típust (osztályt), minden adatnak, függvénymezőnek elkészítettük a megfelelő kezelését, jelentését, használatát, azaz a típusunk használatra kész. Mégis, egyes mezők vagy esetleg az egész típus mellé bizonyos plusz információkat szeretnénk hozzárendelni.

Példaként nézzünk két ilyen, a mindennapi programozási feladataink során is előforduló esetet!

Első példa:

Egy program általában több típussal, adattal dolgozik. Ezek között az adatok között lehet néhány, amelyek értékére a program következő indulásakor is szükségünk lehet. Ezeket a program végén elmentjük a registry-be. (A registry a korábbi Windows ini állományokat váltja fel, programhoz kötődő információkat tudunk ebben az operációs rendszerbeli adatbázisban tárolni.) Természetesen innen be is olvashatjuk ezeket az adatokat. A programunk elvégzi ezt a mentést is, a visszaolvasást is. Ha viszont felteszi valaki a kérdést a programban, hogy jelenleg kik azok, akiket a registry-ben is tárolunk, akkor erre nincs általános válasz! Minden programba bele kell kódolni azt, hogy melyek a mentett adataink! Hasznos lenne, ha nem kellene ezt megtenni, csak jelölhetnénk ezeket a mezőket, és ezt a jelölést később kikereshetnénk.

Második példa:

Ez a példa talán kicsit távolabb áll az informatikától, de nem kevésbé életszerű. Definiáljuk – vagy nem is kell definiálnunk, mert egyébként is léteznek – az emberek különböző csoportjait! Ebbe a definícióba értsük bele a normális használathoz szükséges összes tulajdonságot (nem, foglalkozás, életkor, végzettség, ...)! Emellett szükségünk lehet mondjuk egy olyan információra, hogy például az illető fradidrukker-e! Természetesen az alaptulajdonságok közé is felvehetnénk ezt a jellemzőt, de mivel ennek az adatnak a típus tényleges működéséhez semmi köze nincs – nem fradidrukker is végezheti rendesen a dolgát –, ezért ez nem lenne szerencsés.

Az ilyen plusz információk elhelyezésére nyújt lehetőséget a C# nyelvben az attribútum.

Az attribútumok olyan nyelvi elemek, melyekkel fordítási időben osztályokhoz, függvényekhez vagy adatmezőkhöz információkat rendelhetünk. Ezek az információk aztán futási időben lekérdezhetők.

XI.1. Attribútumok definiálása

Mivel a nyelvben gyakorlatilag minden osztály, így ha egy új attribútumot szeretnénk definiálni, akkor valójában egy új osztályt kell definiálnunk. Annyi megkötés van csak, hogy ezen attribútumot leíró osztálynak az *Attribute* bázisosztályból kell származnia.

Ezek alapján a *fradi_szurkoló* attribútum definiálása a következőképpen történhet:

Példa:

```
class fradi_szurkoló:Attribute
{
    ...
}
```

Egy attribútum, ahogyan egy kivétel is, már a nevével információt hordoz. Természetesen lehetőségünk van ehhez az osztályhoz is adatmezőket definiálni, ha úgy ítéljük meg, hogy a típusnév, mint attribútum még kevés információval bír. Ekkor – mint egy rendes osztályhoz – adatmezőket tudunk definiálni. Az adattípusokra annyi megkötés van, hogy felhasználói osztálytípus nem lehet. Ellenben lehetnek a rendszerbeli alaptípusok *(bool, int, float, char, string, double, long short, object, Type, publikus enum)*. Adatok inicializálására konstruktort definiálhatunk.

Egy attribútum osztályban kétféle adatot, paramétert különböztethetünk meg. Pozicionális és nevesített paramétert. Pozicionális paraméter a teljesen normális konstruktorparaméter. Nevesített paraméternek nevezzük a publikus elérésű adat- vagy tulajdonságmezőket. A tulajdonságmezőnek rendelkezni kell mind a *get*, mind a set taggal. A nevesített paramétereknek úgy adhatunk értéket, mintha normál pozicionális konstruktorparaméter lenne, csak a konstruktorban nincs semmilyen jelölésre szükség. A konstrukció kicsit hasonlít a C++ nyelvben használatos alapértelmezett *(default)* paraméterek használatához.

Ezek után nézzük meg a *fradi_szurkoló* attribútumunkat két adattal kiegészítve. Az egyik legyen az az információ, hogy hány éve áll fenn ez a viszony, míg a másik az, hogy az illető törzsszurkoló-e?

Példa:

```
class fradi_szurkoló:Attribute
{
    private int év;          // hány éve szurkoló
    private bool törzsgárda;   // törzsszurkoló-e
    public fradi_szurkoló(int e)
    {
        év=e;     // konstruktor beállítja a normal paramétert
    }
```

```
        public bool Törzsgárda
        {
            get
            {
                return törzsgárda;
            }
            set
            {
                törzsgárda=value;
            }
        }
    }
```

Ekkor a *Törzsgárda* tulajdonságmezőt nevesített paraméternek hívjuk. Az elnevezést az mutatja, hogy erre a tulajdonságra a konstruktorhívás kifejezésében tudunk hivatkozni, mintha ez is konstruktorparaméter lenne, pedig nem is az!

Példa:

```
[fradi_szurkoló(5,Törzsgárda=true)]
```

XI.2. Attribútumok használata

Az eddig megismert attribútumjellemzők definiálását, használatát, majd a beillesztett információ visszanyerését nézzük meg egy példán keresztül! Mielőtt ezt tennénk, meg kell jegyezni, hogy az információvisszanyerési lehetőségek a típusinformáció lekérdezés lehetőségéhez illeszkednek. Ennek bázisosztálya a *Type*. Ezt a könyvtári szolgáltatást az irodalom gyakran reflekciónak nevezi. Ezeket a lehetőségeket csak olyan mértékben nézzük, amennyire a példaprogram megkívánja.

A típusinformáció szolgáltatás a *Reflection* névtérben található, tehát a használatához a

```
using System.Reflection;
```

utasítást kell a program elejére beírni.

A típusinformáció visszanyeréséhez jellemzően a *Type* osztály *GetType()* statikus függvényét kell meghívni, ami egy paramétert vár tőlünk, azt az osztálytípust, amit éppen fel szeretnénk dolgozni.

```
Type t=Type.GetType("program");
```

Ekkor a programosztályomat szeretném a *t* nevű típusleíró objektumon keresztül elemezni. A kapott *t* objektumra megkérdezhetem például, hogy osztály-e:

```
if (t.IsClass()) Console.WriteLine("Igen");
```

További lehetőségekhez a *Type* osztály online dokumentációjánál nincs jobb forrás.

A típushoz tartozó attribútumok listáját a *GetCustomAttributes()* függvényhívás adja meg. Ez eredményül egy *Attribute* vektort ad. Jellemző módon, ezen egy *foreach* ciklussal lépdelünk végig:

```
foreach (Attribute at in t.GetCustomAttributes())
{
    // feldolgozzuk az at attribútumot...
}
```

Ez a feldolgozás osztályokra (pl. program, stb.) vonatkozó attribútumokra megfelelő. Előfordulhat viszont olyan is, mikor függvényekhez vagy adatmezőkhöz rendelünk attribútumot. Ekkor először az adott típus függvényeit vagy az adatmezőit kell megkapnunk, majd ezen információk attribútumait kell lekérdeznünk.

Egy típus függvényeit a *GetMethods()* hívás szolgáltatja, eredményül egy *MethodInfo* típust ad.

```
foreach(MethodInfo m in t.GetMethods())
{
    foreach (Attribute at in m.GetCustomAttributes())
    {
    // feldolgozzuk az m függvény at attribútumát...
    }
}
```

Egy típus adatmezőit a *GetFields()* adja, eredménye *FieldInfo* típusú.

```
foreach(FieldInfo f in t.GetFields())
{
    foreach (Attribute at in f.GetCustomAttributes())
    {
    // feldolgozzuk az f adatmező at attribútumát...
    }
}
```

Ha egy attribútumosztályt definiálunk, írjuk az osztály neve után az *Attribute* szót, ahogyan azt gyakran tanácsként is megfogalmazzák a szakirodalomban,

hiszen így könnyen meg lehet különböztetni a rendes osztályoktól. Természetesen ebben az esetben is elhagyhatjuk az *attribute* szócskát a használat során, mert azt a fordító odaérti.

Tehát ekkor a *fradi_szurkoló* attribútumosztályt a következőképpen definiáljuk:

```
class fradi_szurkolóAttribute:Attribute
{
    ...
}
```

Ezen definíció esetében is használhatjuk a következő alakot:

```
[fradi_szurkoló]
...
```

Természetesen ekkor a teljes nevű hivatkozás is helyes:

```
[fradi_szurkolóAttribute]
...
```

Ezek után megnézzük a *fradi_szurkoló* attribútumunk definiálását és használatát egy kerek példán keresztül. A példa nem használja az iménti *Attribute* kiegészítéses definíciót.

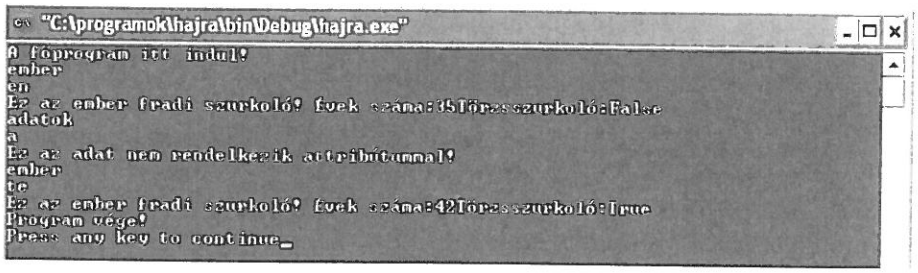
Példa:

```
using System;
using System.Reflection;

class fradi_szurkoló:Attribute
{
    private int ev;          // hány éve szurkoló
    private bool torzsszurkolo;     // vajon törzsszurkoló-e
    public fradi_szurkoló(int e)    // az e rendes, pozicionális
    {                               // paraméter
        ev=e;
    }
    public override string ToString()
    {
        string s="Ez az ember fradiszurkoló!
                  Évek száma:"+ev+"Törzsszurkoló:"+torzsszurkolo;
        return s;
    }
    // az alábbi Torzsszurkoló tulajdonság nevesített
    // fradi_szurkoló paraméter, mert publikus és van get és set
```

```
        // része, amivel az adott torzsszurkolo logikai mezőt állíthatjuk
        public bool Törzsszurkoló
        {
            get
            {
                return torzsszurkolo;
            }
            set
            {
                torzsszurkolo=value;
            }
        }
}


class ember
{
    string nev;
    public ember(string n)
    {
        nev=n;
    }
}


class adatok{} //nem fontos, hogy érdemi információt tartalmazzon

class program
{
    // attribútum beállítása
    [fradi_szurkoló(35)]
    public ember en= new ember("Zoli");
    // a normál konstruktort hívtuk meg, azok a mezők, amiket nem
    // inicializál, alapértelmés szerint kerülnek beállításra
    // 0, ha szám, false ha bool, illetve null, ha referencia
    // Egy attribútum csak a következő adat vagy függvény vagy
    // osztályra hat!!!
    // Így a következő adatok típusú változónak nincs köze
    // a fradi_szurkoló attribútumhoz
    //
    public adatok a=new adatok();
    //
    [fradi_szurkoló(42,Törzsszurkoló=true)]
    public ember te= new ember("Pali");
    // Pali már törzsszurkoló
    public static void Main()
    {
```

```
            program p=new program();
            Console.WriteLine("A főprogram itt indul!");
            Type t=Type.GetType("program");
            foreach( FieldInfo mezo in t.GetFields())
            {
                Console.WriteLine(mezo.FieldType);
                // mezőtípus kiírása
                Console.WriteLine(mezo.Name);
                // mezőnév kírása
                if ((mezo.GetCustomAttributes(true)).Length>0)
                {
                    foreach (Attribute at in mezo.GetCustomAttributes(true))
                    {
                        // megnézzük fradiszurkoló-e
                        fradi_szurkoló f=at as fradi_szurkoló;
                        if (f!=null) // igen ez a mező fradiszurkoló
                            Console.WriteLine(at.ToString());
                    }
                }
                else
                    Console.WriteLine("Ez az adat nem rendelkezik
                                                    attribútummal!");
            }
            Console.WriteLine("Program vége!");
        }
    }
```

Eredményül az alábbi kép jelenik meg:



15. ábra

XI.3. Könyvtári attribútumok

Három könyvtári attribútum áll rendelkezésünkre:

- System.AttributeUsageAttribute: Segítségével megmondhatjuk, hogy a definiálandó új attribútumunkat milyen típusra engedjük használni. Ha nem állítjuk be, alapértelmezés szerint mindenre lehet használni.

  Az *AttributeTargets* felsorolás tartalmazza a választási lehetőségeinket:

```
[AttributeUsage(AttributeTargets.Class)]

osztályattribútum:Attribute
{

}
```

  Ekkor az osztályattribútumok csak osztályok jelölésére használhatók. Előfordulhat, hogy egy attribútumot többször is hozzá szeretnénk rendelni egy adathoz, akkor ennek az osztálynak az *AllowMultiple* nevesített paraméterét igazra kell állítani.

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
```

- System.ConditionalAttribute: Egy szöveg a paramétere. Csak függvényhez kapcsolhatjuk, és az a függvény, amihez kapcsoltuk csak akkor hajtódik végre, ha a paraméterül adott szöveg definiált.

  Példa:

```
...
[Conditional("alma")]
void almafuggveny()
{
    Console.WriteLine("Alma volt!");
}

#define alma
almafuggveny();          // hívás rendben

#undefine alma
almafuggveny();          // hívás elmarad
```

A feltételes végrehajtású függvények kötelezően *void* visszatérésűek, és nem lehetnek virtuálisak, vagy override jelzővel ellátottak! Ezeket a kiértékeléseket az „előfordító" nézi végig.

- System.ObsoleteAttribute: egy üzenet(szöveg) a paramétere, ha egy függvény elavult, és javasolt a mellőzése, akkor használhatjuk.

XI.4. Feladatok

1. Mi az attribútum, hogy tudjuk használni?
2. Hogyan definiálhatunk saját attribútumot?
3. Mi a különbség a rendes és nevesített attribútum paraméter között?
4. Készítsünk *Hisz_e_a_mesében* attribútumot! Definiáljuk az *ember* osztályt, majd alkalmazzuk az attribútumot egyes 'példányaira'!

   Módosítsuk az előző attribútumot úgy, hogy meg tudjuk adni azt az időt amilyen régóta hisz valaki a mesében!

XII. Szabványos adatmentés

Az természetes, ahogy korábban már láttuk is, hogy a rendszerkönyvtárak bőséges támogatást nyújtanak az adatok fájlba mentéséhez illetve visszaolvasásához. Az alkalmazások tekintetében ez teljesen természetes igény. Az alapértelmezett lehetőségek mellett általában a környezetek olyan szabványos eszközzel is rendelkeznek, amelyek minden rendszertípusra rendelkezésre állnak, illetve a felhasználói típusokra „egyszerűen" implementálhatók. Ezzel egy olyan szolgáltatást kapnak az alkalmazások, amelyekkel szabványos ki- és bemeneti adatmozgatást végezhetnek minden erre felkészített típusra.

Ennek a gyakorlati következménye az, hogy az így felkészített rendszerben a programozónak nem kell semmilyen extra mentési stratégián gondolkodnia, hanem egyszerűen csak azokat az objektumokat kell kiválasztania, amelyek értékeit menteni akarja.

Természetesen ez a szolgáltatás is megtalálható majd minden mai környezetben. Az első klasszikus megvalósítása ennek a szolgáltatásnak talán a Visual C++ környezetében jelent meg a Microsoft Foundation Classes (MFC) szolgáltatásaként, ahol a dokumentumosztály (az adatok helye) rendelkezésre bocsát egy *Serialize* függvényt, amely ezt a szabványos alkalmazás adatmentést biztosítja. Az MFC-ben minden könyvtári típus a *Serialize* függvényben használható, míg a felhasználói típusok egy egyszerű lépéssorozat eredményeként képessé tehetők a szerializációra.

Talán innen eredeztethető a névadás is, az irodalomban ezt a szolgáltatást szerializáció *(Serialization)*, szabványos mentés névvel találhatjuk meg.

XII.1. Könyvtári típusok szabványos mentése

A szerializációs szolgáltatás a C# nyelvben, ahogy a standard input-output is, a rendszer könyvtári szolgáltatásának része, és ez a *System.Runtime.-Serialization* névtérben található.

Ha könyvtári típusok mentéséről van szó, akkor közvetlenül ennek a névtérnek a használatára nincs is szükségünk.

Mielőtt a közvetlen lehetőségről beszélnénk, a ki- és bemeneti adatfolyam formázásáról kell néhány szót ejtenünk.

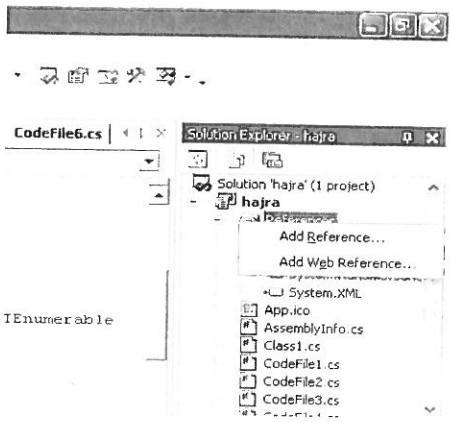
A jelenlegi rendszerkönyvtár kétféle típusú adatfolyam formázását támogatja. Bináris, illetve Soap, XML alapú (szöveges) formázást. Ezek az adatformázó osztályok végzik a valódi adatfolyam elkészítését. A bináris adatfolyam bináris eredményfájlt, míg a Soap XML formátumú szöveges állományt hoz

létre. Ha ez a kétféle lehetőség nem elég, akkor egyéni formázó osztályok definiálhatók.

A formázó osztályok a *System.Runtime.Serialization.Formatters.Binary* és a *System.Runtime.Serialization.Formatters.Soap* névterekben találhatók.

A szerializáció folyamata az alábbi lépésekből áll:

1. Ki kell jelölni egy *FileStream* objektumot, ami a program kapcsolatát jelenti az állománnyal.
2. Definiálni kell egy formázó objektumot, ez jelenleg bináris vagy szöveges lehet. A szöveges formázó objektum létrehozásához a *System.Runtime.Serialization.Formatters.soap.dll* referenciát a projekthez kell csatolni *(Project nézet \ References \ jobb egérgomb \ Add reference)*.



16. ábra

3. A formázó objektum *Serialize* függvényével mentjük az adatokat.
4. Bezárjuk a fájlkapcsolatot.

Ezek után nézzük a lehetőséget bemutató példaprogramot!

Példa:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;
```

```
class serprog
{
    // program adatai
    int i=2;
    double d=3.5;
    string s="fradi";

    public static void Main()
    {
        serprog p=new serprog();
        //1.filestream létrehozása
        FileStream fb=File.Create("ser.bin");
        // 2.Binaryformatter készítés
        FileStream fs=File.Create("ser.txt");
        BinaryFormatter b=new BinaryFormatter();
        SoapFormatter so=new SoapFormatter();
        //3.bináris mentés
        b.Serialize(fb,p.i);
        b.Serialize(fb,p.d);
        b.Serialize(fb,p.s);
        // 3.Soap mentés
        so.Serialize(fs,p.i);
        so.Serialize(fs,p.d);
        so.Serialize(fs,p.s);

        //4. fájlzárás
        fb.Close();
        fs.Close();
    }
}
```

A program magyarázatául csak annyit, hogy miután lefuttatjuk, a projekt *debug* könyvtárában (ebbe a könyvtárba kerül a lefordított program) létrejön a *ser.bin*, illetve a *ser.txt* állomány. Ha ezt megnézzük, láthatjuk, hogy míg az előbbi bináris, az utóbbi XML formátumú.

Visszaolvasás hasonló módon, csak a *formatter* objektum *Deserialize* függvényhívással végezhető el.

XII.2. Saját típusok szabványos mentése

Az előző példa során láttuk, mit jelent az alaptípusok mentési lehetősége. A valós alkalmazásaink során azonban biztosan készíteni kell a fő programosztályon kívül egyéb felhasználói osztályokat is.

Ebben az esetben a mentési folyamat visszavezeti az adatok mentését a rendszertípusok mentésére. Ahhoz, hogy ezt megtegye nekünk, egyetlen dolgot kell tennünk, a saját típusunkat a menthető, *Serializable* attribútummal kell jelölnünk.

Módosítsuk a példaprogramunkat úgy, hogy definiálunk egy *ember* típust, amit még használnia kell a programunknak. A módosított forrásszöveg a következőképpen néz ki:

Példa:

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;

[Serializable]
// az attribútum hatására a formázó lekezeli a típus szerializációját
class ember
{
     string nev;
     int kor;
     public ember(string n, int k)
     {
          nev=n; kor=k;
     }
}
class serprog
{
     // program adatai
     int i=2;
     double d=3.5;
     string s="fradi";
     // a saját típus használata
     ember e=new ember("Zoli", 34);
     public static void Main()
     {
          serprog p=new serprog();
          //fájlstream létrehozása
          FileStream fb=File.Create("ser.bin");
          // Binaryformatter készítés
          FileStream fs=File.Create("ser.txt");
          BinaryFormatter b=new BinaryFormatter();
          SoapFormatter so=new SoapFormatter();
          //Bináris mentés
          b.Serialize(fb,p.i);
          b.Serialize(fb,p.d);
```

```
            b.Serialize(fb,p.s);
            b.Serialize(fb,p.e);
            //Soap mentés
            so.Serialize(fs,p.i);
            so.Serialize(fs,p.d);
            so.Serialize(fs,p.s);
            so.Serialize(fs,p.e);

            //fájl lezárása
            fb.Close();
            fs.Close();
        }
    }
```

A gyakorlati munka során előfordulhat – például az osztály belső szerkezete nem engedi meg –, hogy nem bízhatjuk rá a formázóra a típusunk végigelemzését és az elemek mentését. Ebben az esetben a valódi szerializációs szolgáltatást végző függvényeket nekünk kell az osztályunkba implementálni.

Azon túl, hogy a *[Serializable]* attribútumot definiálni kell, az osztályunknak még implementálnia kell az *ISerializable* interface-t is. Ebben a deszerializáció számára egy speciális konstruktort és egy *GetObjectData* függvényt kell definiálni.

Az interface és a függvények hosszabb elemzése nélkül nézzük meg az ennek megfelelően módosított *ember* osztályunkat. A program kódja változatlan, ezért azt nem listázzuk ide.

Példa:

```
...
[Serializable]
// az attribútum hatására a formázó lekezeli a típus szerializációját
class ember:ISerializable
// az interface két plusz függvénydefiníciót ír elő
{
    string nev;
    int kor;
    public ember(string n, int k)
    {
        nev=n; kor=k;
    }
    internal ember(SerializationInfo si, StreamingContext st)
    {
        // elemek visszaállítása
        nev=si.GetString("nev");
        kor=si.GetInt32("kor");
    }
```

```
        public void GetObjectData(SerializationInfo si, StreamingContext st)
        {
            // a szerializálás adatait beállítja a SerializationInfo si
            // objektumba
            si.AddValue("nev",nev);
            si.AddValue("kor",kor);
            // típusinformáció beállítása
            Type t=this.GetType();
            si.AddValue("Típusinfo",t);
        }
    }
```

XII.3. Feladatok

1. Mit nevezünk szabványos mentésnek?
2. Milyen hasonló megvalósításokat ismer más fejlesztő rendszerben?
3. Milyen szabványos mentéseket támogat a fejlesztő rendszer?
4. A másodfokú egyenlet (VIII.3. fejezet 4. feladat) adatait mentsük el bináris formában!
5. Definiáljunk egy *szurkoló* osztályt *(név, kedvenc_csapat)*, és biztosítsuk ennek a típusnak szabványos menthetőségét!

XIII. Könyvtárak, távoli és web könyvtárak

Az alkalmazások készítésének egyik lényeges eleme a fejlesztők rendelkezésére álló könyvtári szolgáltatások mennyisége, minősége, illetve a használt keretrendszernek az a tulajdonsága, hogy mennyire ad lehetőséget osztott alkalmazás készítésére.

Egy alkalmazás kiszolgáló függvényei vagy a helyi gépen helyezkednek el, vagy valamelyik távoli kiszolgáló gépen.

XIII.1. Helyi könyvtárak használata

XIII.1.1. Rendszerkönyvtári elemek használata

A helyi fejlesztőrendszer könyvtárairól nyugodtan kijelenthetjük, hogy azok használata nélkül, ahogy a legelső részben is láttuk, még a legegyszerűbb program sem készíthető el. (*System* névtér, *Console* osztály, *WriteLine* függvényhívás.)

A .NET keretrendszer egyik, talán leggyakrabban használt könyvtára a *System.Collections* névtér. A névtér sok hasznos osztálydefiníciót tartalmaz, melyek közül az alábbiak a legfontosabbak:

- ArrayList: a méretét dinamikusan növelni képes objektumvektor. Az *IList* interface-t implementálja. Az *IList* három jellemző tulajdonságot ír elő, ezek:

IsFixedSize:	megvizsgája, fix méretű-e a vektor,
IsReadOnly:	olvasható-e a vektor,
Item:	a vektor adott indexű elemét adja, az ArrayList objek tum indexereként használható.

- HashTable: olyan vektoriális adatsorozat, ahol egy kulcsindexhez tartozik egy érték. A kulcs „hash" értékéhez rendeli az adatot. Gyakran asszociatív vektornak hívják ezt a szerkezetet.

- Queue: a sor adatszerkezet megvalósítása. Az az elem megy elsőnek ki a sorból, amely elsőnek beérkezett. *(First in, first out.)*
- SortedList: rendezett lista. Hasonló szerkezet, mint a *HashTable*, melyek elemei a kulcs alapján sorba vannak rendezve. Kulcs és index alapján is az elemekhez lehet férni.
- Stack: verem adatszerkezet. Az utoljára betett elemet tudjuk mindig kivenni. *(Last in, first out)*

Példaként tekintsük a következő *ArrayList* és *Queue* használatát bemutató mintaprogramot:

Példa:

```
using System;
using System.Collections;
public class mintacol
{
    public static void Main()
    {
        // Új ArrayList objektum létrehozása.
        ArrayList elemek = new ArrayList();
        elemek.Add( "Hajrá" );
        // alapértelmezésben a méret növelhető (IsFixedSize==false)
        elemek.Add( "Fradi" );
        elemek.Add(25);
        for (int i=0;i<elemek.Count;i++)
            Console.WriteLine(elemek[i]);

        // Új sor (Queue) objektum létrehozása.
        Queue sor = new Queue();
        // adat sorba írása
        sor.Enqueue( "mézes" );
        sor.Enqueue( "maci" );
        sor.Enqueue( 3.1415 );
        // mind az Arraylist, mind a sor implementálja az IEnumerable
        // interface-t, így a foreach használható
        foreach(object o in sor)
            Console.WriteLine(o);

        //  ArrayList objektumhoz hozzáadjuk a sort.
        elemek.AddRange( sor );
        // elem kivétele a sorból
        Console.WriteLine(sor.Dequeue());
        Console.WriteLine(sor.Dequeue());
        Console.WriteLine(sor.Dequeue());
```

```
            // Kiírjuk az elemeket
            Console.WriteLine( "A bővített ArrayList a következő
                                        elemeket tartalmazza:" );
            PrintValues( elemek );
        }

        // az IEnumerable alapján végiglépdelünk az elemeken
        public static void PrintValues( IEnumerable adatok)
        {
            IEnumerator adat = adatok.GetEnumerator();
            while ( adat.MoveNext() )
                Console.WriteLine( adat.Current );
        }
    }
```

XIII.1.2. Saját könyvtári elemek használata

A keretrendszer közös nyelvi specifikációjának köszönhetően tetszőleges nyelvi könyvtárat használhatunk azonos módon, tetszőleges alkalmazásban. A példában VB függvényt definiálunk, amit egy másik alkalmazás használ:

```
Public Class Demo
    Shared Function legjobb_csapat() As String
        legjobb_csapat = "Fradi!"
    End Function
End Class
```

A fenti forráskódot tetszőleges szövegszerkesztőbe beírva, az alábbi parancssor segítségével lefordítható. (A keretrendszer mindegyik fordítója indítható parancssorból.)

```
vbc /target: library /out:DllDemo Demo.vb
```

Természetesen új projektet készítve, az osztálykönyvtár típust választva a Visual Studio.NET környezetbe is beírhatjuk ezt a pár sort, majd a *Build* menüpont segítségével lefordíthatjuk.

Az elkészült *DllDemo.dll* állomány egy tipikus könyvtári állomány lesz, amit bármely másik alkalmazás használhat, pontosan úgy, ahogy a rendszerkönyvtárakat. Készítsünk most egy C# nyelvű alkalmazást, amely szeretné használni a *DllDemo* könyvtári szolgáltatást.

A használatához az alkalmazáshoz kell csatolni *(add reference)* ezt a könyvtárállományt, majd a *using DllDemo* utasítással a *DllDemo* névtér elérhetőségét is biztosítani kell.

Példa:

```
using System;
using DllDemo

public class
{
    public s
    {
        //
        //
        Der
        //
        //
        Co
        Co
    }
}
```

XIII.2. Távoli

A távoli könyv
megérdemelnének,
közé is beletartozik
Egy alkalmaz
nevezzük. Az előz
A független
lehetőségét a .N
Valójában hasonl
COM (Componen
Az alapprobl
valahol meglévő
nincs jelen, visz
másik gép önálló
Így a legfont
melyek természe
legfontosabb kér
Ennek a kon

- Kommuni
- Az adatok

Példa:

```
using System;
using DllDemo;

public class minta
{
    public static void Main()
    {
        // Új demo objektum létrehozása.
        // ezt készítettük Visual Basic nyelvben
        Demo d = new Demo();
        //kíváncsiak vagyunk arra, hogy ki a legjobb csapat
        // meghívjuk a dll függvényt
        Console.WriteLine(d.legjobb_csapat());
        Console.WriteLine("Program vége");
    }
}
```

XIII.2. Távoli könyvtárhívás

A távoli könyvtárhívás szerkezete, lehetőségei önmagában egy teljes könyvet is megérdemelnének, de a jelen tárgyalási menetbe, a könyvtári szolgáltatások típusai közé is beletartozik, ezért egy rövid bevezetést meg kell említeni erről a területről.

Egy alkalmazás operációs rendszerbeli környezetét *application domain*nek nevezzük. Az előző DLL készítési lehetőség egy *application domain*t alkot.

A független alkalmazások közti kommunikációt, távoli alkalmazáshívás lehetőségét a .NET környezetben *REMOTING* névvel említi az irodalom. Valójában hasonló lehetőségről van szó, amit a korábbi fejlesztési környezetek, *COM (Component Object Model)* néven említenek.

Az alapprobléma valójában ugyanaz, mint a DLL könyvtár esetében, egy valahol meglévő szolgáltatást szeretnénk igénybe venni. Ez DLL formában most nincs jelen, viszont az a típusú objektum, aminek ez a szolgáltatása van, egy másik gép önálló alkalmazásaként van jelen.

Így a legfontosabb kérdés az, hogy önálló alkalmazási környezetek között, melyek természetesen vagy azonos számítógépen helyezkednek el, vagy nem, a legfontosabb kérdés az, hogyan tudunk információt átadni.

Ennek a kommunikációnak legfontosabb elemei a következők:

- Kommunikációs csatorna kialakítása, regisztrálása.
- Az adatok szabványos formázása a kommunikációs csatornába írás előtt.

- Átmeneti, proxy objektum létrehozása, mely az adatcserét elvégzi a távoli objektum (pontosabban annak proxy objektuma) és a helyi alkalmazás között.
- Távoli objektum aktiválása, élettartama

Kommunikációs csatornát *Tcp* vagy *Http* alapon alakíthatunk ki. Használat előtt regisztrálni kell egy csatornát, ami egy kommunikációs porthoz kötött. Egy alkalmazás nem használhat más alkalmazás által lefoglalt csatornát. A kétféle kommunikáció használata között a legfontosabb különbség az adatok továbbításában van. A *Http* protokoll a SOAP szabványt használja az adatok továbbítására XML formában. A *Tcp* csatorna bináris formában továbbítja az adatokat.

A proxy objektum reprezentálja a távoli objektumot, továbbítja a hívásokat, visszaadja a hívási eredményt. A távoli objektumok a szerver oldalon automatikusan, vagy kliens oldali aktiválással jöhetnek létre. Az automatikus objektumok esetében megkülönböztetünk egy kérést kiszolgáló objektumot *(Single call)* vagy több kérést kiszolgáló *(Singleton)* objektumot. Meg kell természetesen jegyezni, hogy a szolgáltatást, a programot magát el kell indítani, hiszen a klienshívás hatására csak az alkalmazás egy kiszolgáló típusa jön automatikusan létre! Ezt vagy úgy érjük el, hogy az alkalmazásunkat futtatjuk egy konzolsori parancs kiadásával, vagy mint regisztrált szervízt az operációs rendszer futtatja!

Mielőtt egy konkrét példát néznénk, beszélni kell a paraméter átadás lehetőségéről. Egy alkalmazáson belül a keretrendszer biztosítja az adatok paraméterkénti átadását, átvételét. Esetünkben nem egy alkalmazásról van szó, hanem egy kliensről és egy (vagy több) szerverről, melyek különböző környezetben (*app. domain*ben) futnak. Emiatt az adatok átadása sem lehet ugyanaz, mint egy alkalmazáson belül. A különböző alkalmazási környezetben futó programok közötti adatcsere folyamatát *Marshaling* kifejezéssel illet az irodalom, utalva arra, hogy ez a fajta alkalmazási határon átvezető adatforgalom mást jelent, mint egy alkalmazási környezet esetében.

Egy adatot három kategóriába sorolhatunk a .NET keretrendszerben, a távoli adatátadás *(Marshaling)* szempontjából:

1. Érték szerint átadott adatok. Ezen típusok a szabványos szerializációt (mentés) támogató objektumok. *(Marshal-by-value).* Ekkor a típus a *[Serializable]* attribútummal jelölt. A környezet alaptípusai (*int*, stb.) menthetőek, így érték szerint átadható adatok.
2. Referencia szerint átadott adatok. Ezek a típusok kötelezően a *MarshalByRefObject* típusból származnak.

3. Alkalmazásdomainek között nem átadható típusok. Ebbe a kategóriába esik minden olyan típus, amelyik nem *MarshalByRefObject* utód, vagy rendelkezik a *[Serializable]* attribútummal.

A legfontosabb tulajdonságok ismertetése után nézzünk egy egyszerű példát. Manuálisan fogjuk a szerverkiszolgálókat is futtatni az egyszerűség kedvéért.

A példánkban két szerverszolgáltatást készítünk, az egyik *http*, míg a másik *tcp* hálózati kommunikációt folytat. A forráskódot mind parancssorból, mind a környezetből tudjuk fordítani. Ez utóbbi esetben a *Project* referencia ablakban a projekthez kell adni *a System.Runtime.Remoting* névteret.

Az alábbi példa egy *bajnokszerver* típust definiál, regisztrálja magát, és más feladata nincs. Enterre a *Main* program azért várakozik, mert ha engedjük befejeződni, mivel nem rendszerrész-szolgáltatás, nem lehet elérni.

Példa:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
public class BajnokSzerver : MarshalByRefObject {

    public string foci;

    public static int Main(string [] args) {

        TcpChannel chan1 = new TcpChannel(8085);
        // 8085 tcp port lefoglalva
        ChannelServices.RegisterChannel(chan1);
        // regisztráció rendben
        RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(BajnokSzerver),
            "bajnok",     // kliens oldalon elérhető
                          // szolgáltatás neve
            WellKnownObjectMode.Singleton);

        System.Console.WriteLine("Program vége, nyomjon entert");
        System.Console.ReadLine();
        return 0;
    }

    public BajnokSzerver() {
        foci="Magyar bajnokság";
        Console.WriteLine("Remote szerver aktiválva!");
    }
```

```
        public string Ki_a_bajnok(int ev)
        {
            string nev="Nem tudom!";
            switch (ev)
            {
                case 2002: nev="Dunaferr";
                    break;
                case 2003: nev="MTK";
                    break;
                case 2004: nev="Ferencváros";
                    break;
            }
            Console.WriteLine("A kért bajnocsapat: {0} ",nev);
            return nev;
        }
}
```

A másik szerver lényegében abban különbözik, hogy *http* csatornát használ:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
public class golkiraly : MarshalByRefObject {

    public string sportag ;

    public static int Main(string [] args) {

        HttpChannel chan1 = new HttpChannel(8086);
        // http csatorna foglalása
        ChannelServices.RegisterChannel(chan1);
        // regisztráció
        RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(golkiraly),// típus nevének regisztrálása
            "golkiraly",        // kliens oldali http szolgáltatásnév
            WellKnownObjectMode.Singleton); // szervermód

        System.Console.WriteLine("Program vége!");
        System.Console.ReadLine();
        return 0;
    }

    public golkiraly() {
        sportag = "foci";
        Console.WriteLine("Gólkirály szerver aktiválva");
    }
```

```
    public string ki_a_golkiraly(int ev) {
        Console.WriteLine("Gólkirály szolgáltatás az alábbi
                                                    sportágban: {0}",
            sportag);
        return "Sajnos még nem tudom!";
    }
}
```

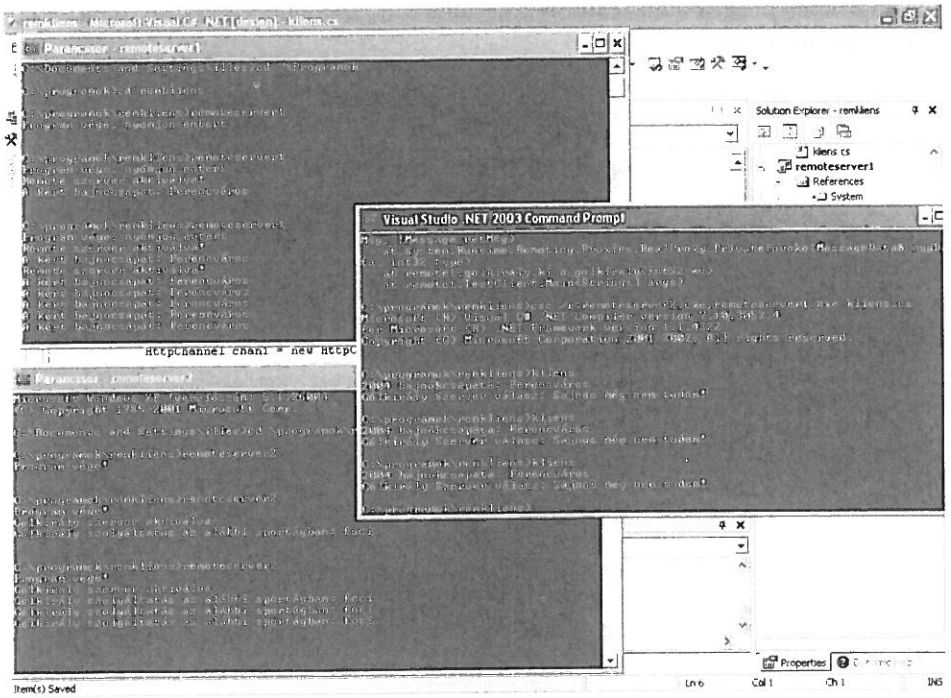
A kliens programunk, amelyik mindkét kiszolgálóprogramot használja a következő formájú:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels.Http;
using System.IO;
public class kliens
{
    public static int Main(string [] args) {
        HttpChannel chan1 = new HttpChannel();
        // kliens csatorna portot nem adunk meg
        ChannelServices.RegisterChannel(chan1);
        golkiraly g =(golkiraly)Activator.GetObject(
            typeof(golkiraly),
            "http://localhost:8086/golkiraly");
        TcpChannel chan2 = new TcpChannel();
        ChannelServices.RegisterChannel(chan2);
        BajnokSzerver b = (BajnokSzerver)Activator.GetObject(
            typeof(BajnokSzerver),
            "tcp://localhost:8085/bajnok");
        try {
            string bajnok = b.Ki_a_bajnok(2004);
            Console.WriteLine("2004 bajnokcsapata: {0}",bajnok );
            string golkiraly= g.ki_a_golkiraly(2004);
            Console.WriteLine(
            "Gólkirály Szerver válasz: {0}",golkiraly );
            }

        catch (Exception ioExcep) {
            Console.WriteLine("Remote IO Error" +
                "\nException:\n" + ioExcep.ToString());
            return 1;
        }
        return 0;
    }
}
```

A kliens fordítása parancssorból kényelmesebb:

```
csc /r:remoteserver1.exe,remoteserver2.exe kliens.cs
```

Indítsuk el egy-egy ablakban először a szerver-, majd a kliensprogramot, ahogy a következő képen is látszik.



17. ábra

Természetesen a szerverablakban látható eredmény a szemléletességet szolgálja, a valós alkalmazások (mivel nem is futnak önálló ablakban) nem írogatnak semmit a képernyőre.

A korábbi feladathoz hasonló kliens-szerver alkalmazáskészítési lehetőséget is biztosít a keretrendszer, úgynevezett *WebRequest*, *WebResponse* modellt használva, vagy a klasszikus *TCP*, *UDP* protokollok használatával (*TcpListener*, *TcpClient*, *UdpClient*). Az osztályok a *System.Net* névtérben találhatók. A *System.Net* összes szolgáltatása a *System.Net.Sockets* névtér szolgáltatásaira épül. A *System.Net.Sockets* névtér a WinSock32 API megvalósítása. Ezen hálózati alkalmazások készítésének lehetősége túlmutat e könyv keretein, így nem is részletezzük azokat.

XIII.3. Webkö

A webkönyvtár
távoli könyvtárhív
megvalósításához h
rálását és biztosítja
Ekkor azon a
IIS webszervernek
Ebben az esetl
szerveroldali alkal
Web Service névv
például a klasszikus
Szerveralkalm
mazást. Ebben az
ellátott függvényel
Nézzük ezek a
lósított forrását:

Példa:

```
using Syst
using Syst
using Syst
using Syst
using Syst
using Syst
using Syst

namespace
{
    /// <
    /// S
    /// <
    publi
    {
```

XIII.3. Webkönyvtárak használata

A webkönyvtárak használata *(Web Services, webszolgáltatások)* valójában a távoli könyvtárhívás *http* alapú alkalmazásának webkiszolgálón keresztüli megvalósításához hasonlít, a webszerver tölti be a kiszolgáló rendszerbe integrálását és biztosítja a távoli elérhetőséget.

Ekkor azon a kiszolgálón, ahol webkönyvtárat akarunk elhelyezni, ott MS IIS webszervernek kell futni.

Ebben az esetben is legalább két alkalmazás készítéséről beszélhetünk, a szerveroldali alkalmazás készítéséhez egy külön sablont találunk (ASP.*NET Web Service* névvel), míg a kliensalkalmazás tetszőleges C# alkalmazás lehet, például a klasszikus konzol.

Szerveralkalmazás készítéséhez kezdjünk új *ASP.NET Web Service* alkalmazást. Ebben az alkalmazásban egyszerűen a *[Webmethod]* attribútummal ellátott függvények érhetők el a külső kliensek számára

Nézzük ezek alapján a bajnokra vonatkozó példánk webkönyvtárral megvalósított forrását:

Példa:

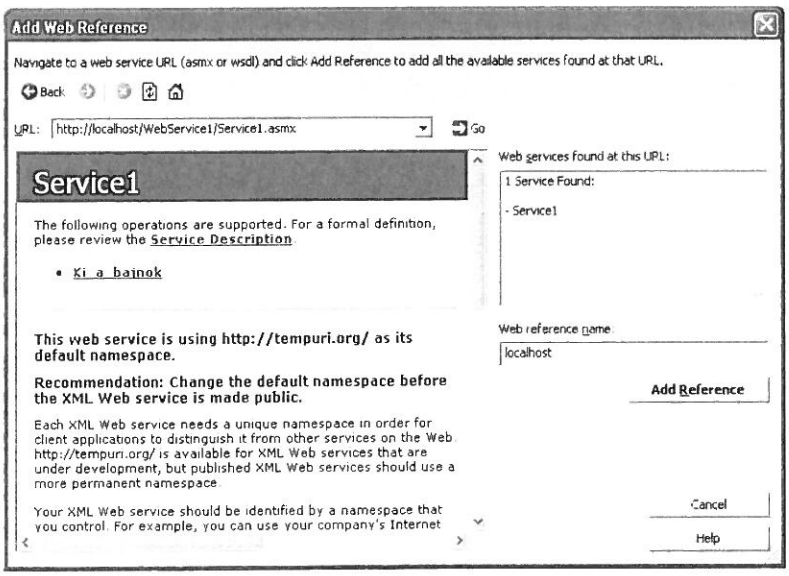
```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace WebService1
{
    /// <summary>
    /// Summary description for Service1.
    /// </summary>
    public class Service1 : System.Web.Services.WebService
    {
        public Service1()
        {
            InitializeComponent();
        }

        //Component Designer generated code

        [WebMethod]
        public string Ki_a_bajnok(int ev)
        {
```

```
                string nev="Nem tudom!";
                switch (ev)
                {
                    case 2002: nev="Dunaferr";
                        break;
                    case 2003: nev="MTK";
                        break;
                    case 2004: nev="Ferencváros";
                        break;
                }
                return nev;
            }
        }
    }
```

A forráskód *service1.asmx.cs* néven található. Fordítás után az IIS kiszolgáló *webservice1* virtuális könyvtárába kerülő *service1.asmx* állományra hivatkozva tudjuk futtatni a feladatot. A fenti kiszolgáló használatához webreferenciát kell a készítendő projekthez adni, ahol meg kell adni a Web Service címét a következő módon:

http://localhost/webservice1/service1.asmx

Ezt legegyszerűbben az *Add Web Reference* menüpontban tehetjük meg:



18. ábra

A használatát az alábbi forráskód mutatja:

```
...
using System;
namespace webhasznal

{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //
            // a webreferencia neve localhost
            //
            localhost.Service1 s=new localhost.Service1();

            Console.WriteLine(s.Ki_a_bajnok(2004));
        }
    }
}
```

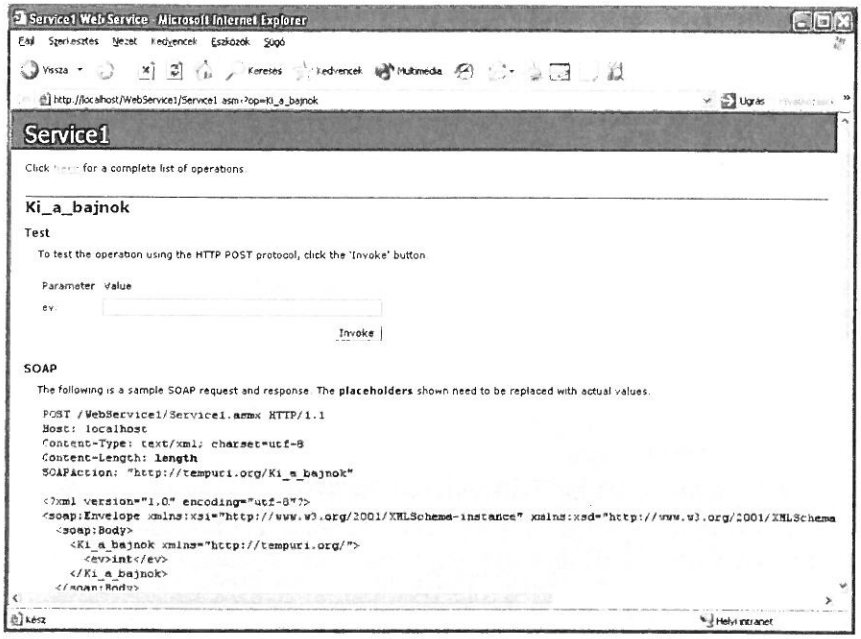
A futási eredmény megadja a várt csapatnevet. Természetesen a *using* névtér használattal az *s* objektumdefiníciót egyszerűbben írhatjuk.

```
using webhasznal.localhost;
// projekt neve után jön a webreferencia neve
...
Service1 s=new Service1();
...
// A használatban már nincs változás
```

A webkiszolgáló szolgáltatását, nemcsak egy kliens programból, hanem a legáltalánosabban használt webes kliens programunkból, például az Internet Explorerből is kipróbálhatjuk (19. ábra).

A szolgáltatásokat általánosan leíró *nyelv (Web Service Description Language, WSDL)* természetesen XML formában adja meg, ezt az alábbi kérés-sel tudjuk megnézni:

http://localhost/webservice1/service1.asmx?WSDL



19. ábra

A webszolgáltatásnak ezt a használatát gyakran szinkronhívásnak nevezzük. Ugyanis ebben az esetben a *Ki_a_bajnok* hívása az eredmény visszaérkezéséig nem tér vissza. Ez webes kiszolgálás esetén gyakran hosszabb időt is igénybe vehet. Sőt az sem kizárt, hogy adott időn belül vissza sem tér.

Ha figyelmesen megnézzük a projekt könyvtárunkat, akkor ebben megjelent egy *Web References* könyvtár is. Ebben aztán annyi könyvtárt találunk, ahány webreferenciát csatoltunk a projektünkhöz. Esetünkben találunk egy localhost könyvtárat (*localhost* a neve a http://localhost/webservice1 URL által megadott webkönyvtárnak), ebben egy *References.cs* állományt. Ha ezt megnézzük, látjuk, hogy nem csak a megírt függvényünk van leírva benne, hanem egy *BeginKi_a_bajnok* és egy *EndKi_a_bajnok* hívás is.

Ezek a függvények adnak lehetőséget arra, hogy egy ilyen webkiszolgálón elhelyezett szolgáltatást ne csak a saját nevével, úgynevezett szinkronhívással tudjunk elérni, hanem aszinkron módon is.

A *Begin*-nel kezdődő függvény mindig azonnal visszatér, eredményül egy *IAsyncResult* objektumot kapunk. Ezen az objektumon keresztül kérdezhetjük meg, hogy befejeződött-e a végrehajtás. Esetünkben a második paraméter és a harmadik is a *null*, jelezve azt, hogy az aszinkron hívás eredményét az *IAsyncResult* objektumon keresztül akarjuk lekérdezni. Egyébként a második paraméter egy delegált *(callback)* függvény, ami által adott függvény kerül végrehajtásra akkor, amikor megérkezik az eredmény. A harmadik paraméter

egy kérés állapotot jelző objektum, amiben a *callback* függvény meg tudja nézni az eredményparamétereket.

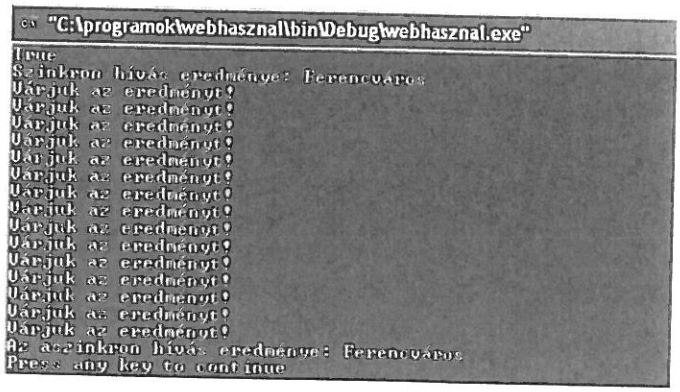
Az alábbi példa nem használja ezeket a paramétereket, hanem az *IAsyncResult IsCompleted* tulajdonságát figyelve várakozik az eredményre.

Példa:

```
static void Main(string[] args)
{
    //
    // TODO: Add code to start application here
    //
    localhost.Service1 s=new localhost.Service1();
    Console.WriteLine("Szinkronhívás eredménye:
                                        {0}",s.Ki_a_bajnok(2004));
    IAsyncResult e=s.BeginKi_a_bajnok(2004,null,null);
    // elindítottuk a függvényhívást, majd addig
    //várakozunk, amig az eredmény meg nem érkezik
    while(e.IsCompleted!=true)
        Console.WriteLine("Várjuk az eredményt!");
    // megjött az eredmény
    // kiolvassuk azt
    string eredmeny=s.EndKi_a_bajnok(e);
    Console.WriteLine("Az aszinkronhívás eredménye:
                                                {0}",eredmeny);
}
```

Eseményvezérelt környezetben, grafikus alkalmazás készítésekor ez a módszer kézenfekvő lehet.

A program futásának eredménye jól illusztrálja az aszinkron végrehajtás jellegzetességét, amit az alábbi futási kép is jól szemléltet:



20. ábra

XIII.4. Feladatok

1. Milyen rendszer könyvtári típusokat ismer?
2. Mi a különbség a rendszer könyvtári és a web könyvtár szolgáltatás között?
3. Mit nevezünk szinkron, illetve aszinkron könyvtári hívásnak?
4. Írjon programot, amely egy listában tárolja a kifizetett telefonszámlák összegét! Valósítsa meg a beolvasást, kiírást függvényként!
5. Készítsen WebService-t, amely egy adott névről eldönti, hogy olimpiai bajnok neve-e!

XIV. Az előfeldolgozó

XIV.1. Szimbólumdefiníció használata

Az előfeldolgozónak vagy előfordítónak szóló utasítások mindig a # karakterrel kezdődnek.

Az előfordítónak szóló makró definiálási lehetőségének a formája:

#define név

Példa:

```
#define menu_h          // menu_h szimbólum definiálva
```

A definíciók hatása az adott modul végéig tart.

Egy azonosító definiálásának gyakori formája a következő:

Példa:

```
#define ALMA
```

Ekkor nem definiálunk helyettesítési értéket, így ez csak annyit mond meg az előfordítónak, hogy ettől kezdve az ALMA szó legyen ismert. Ez gyakran használt mód a feltételes fordítás azonosítóinak definiálására.

Ha már nincs szükségünk egy korábban definiált azonosítóra, és meg szeretnénk kérni az előfordítót, hogy felejtse azt el, akkor a következő előfordítónak ezt az utasítást adhatjuk:

#undef név

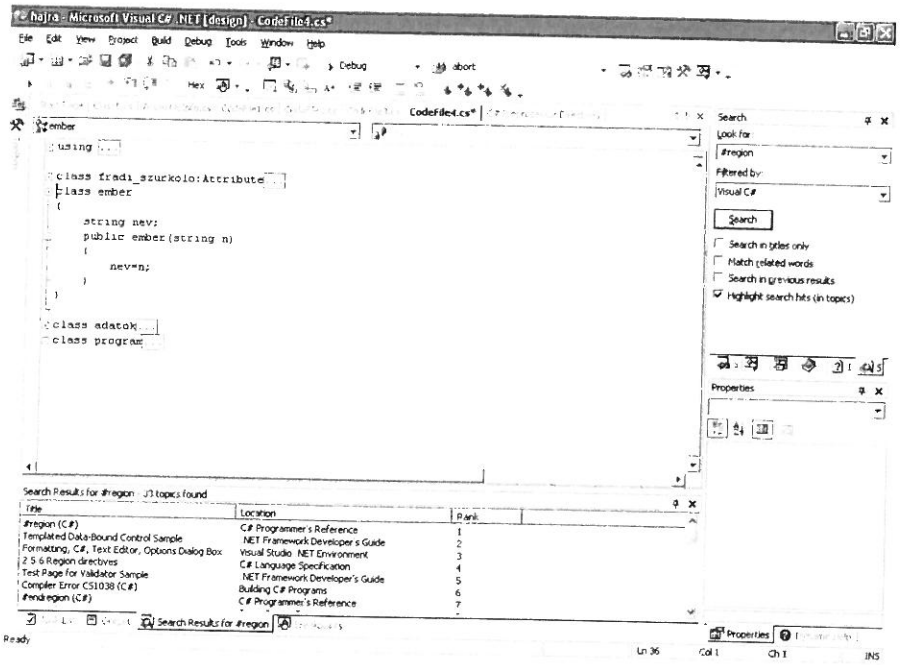
Például az imént definiált *ALMA* azonosító esetében:

Példa:

```
#undef ALMA
```

XIV.2. Területi jelölés

A fejlesztőrendszer szövegszerkesztője nyújtja alapértelmezésben azt a kényelmi szolgáltatást, hogy egy függvény vagy osztálydefiníció törzsét a szövegszerkesztő bal oldalán található + vagy – gombokkal elrejthetjük vagy megtekinthetjük.



21. ábra

A fenti képen az *ember* osztály tartalmát látjuk, míg a többi egységét (adatok, program...) nem, azokat csak jelöli a szövegszerkesztő, hogy léteznek, de pillanatnyilag nem érdekesek. Ez a segítség nagyobb állományok szerkesztésénél hasznos, hiszen a szerkesztőablakban jobban a lényegre tudunk figyelni.

Ezt a szolgáltatást egészíti ki a

```
#region alma
    osztályok, függvények helye
#endregion alma
```

régió, területdefiníció. Ekkor az *alma* „blokkban", régióban definiált osztályok, függvények egyszerre húzhatók össze vagy nyithatók ki.

XIV.3. Feltételes fordítás

#if konstans kif	Igaz-e konstans kif.
#ifdef azonosító	Van-e ilyen makró
#ifndef azonosító	Nincs-e ilyen makró

Mindhárom alakot követheti a *#else* direktíva, majd kötelezően zárja:

#endif
#line sorszám

A fordító úgy viselkedik, mintha a következő sor a sorszámmal megadott sor lenne.

#line default	eredeti sorszám visszaállítása

XIV.4. Hibaüzenet

Ha már az előfordító felfedez valamilyen hibát, akkor hibaüzenetet ad, és befejezi az előfordítást.

A vezérlődirektíva formája:

#error hibaszöveg

Példa:

```
#ifndef c#
#error Sajnos nem a megfelelő fordítót használja!
#endif
```

XIV.5. Feladatok

1. Mik az előfordító jellemző szolgáltatásai?
2. Hogy definiálhatunk és szüntethetünk meg egy azonosítót?
3. Mit jelent a régió definíció?
4. Az eddig elkészített programjait módosítsa régió definíciókkal! Figyelje meg, hogy mennyivel áttekinthetőbbé vált a programjának a kódja!
5. Definiálja úgy a másodfokú egyenletet megoldó függvényt, hogy csak akkor fordítsuk le, ha „szükséges"!

XV. Nem felügyelt kód használata

A C# nyelvű program fordítása egy felügyelt eredményprogramot ad, amin a felügyeletet a .NET keretrendszer biztosítja. Szükség lehet azonban arra, hogy a rendelkezésre álló nem felügyelt, rendes Win32 API könyvtárak szolgáltatásait elérjük, és az ezekkel kapcsolatos adatainkat tudjuk használni, a könyvtárhoz hasonló nem felügyelt kódrészletet tudjunk írni.

XV.1. Nem felügyelt könyvtár elérése

Talán leggyakrabban ez az igény merül fel, hiszen ha megvan egy jól megírt szolgáltatás a korábbi Windows API könyvtárban, akkor azok használata mindenképpen kifizetődőbb, mint helyettük megírni azok menedzselt változatát.

Erre ad lehetőséget a *DllImport* attribútum használata. Egy könyvtári függvény használatához három lépést kell megtenni:

1. A *DllImport* attribútumnak meg kell mondani a *Dll* állomány nevét, amit használni akarunk.
2. A könyvtárban lévő függvényt a fordító számára deklarálni kell, ebben az extern kulcsszó segít. Ezeket a függvényeket egyúttal statikusnak is kell jelölni.
3. A *System.Runtime.InteropServices* névteret kell használni.

Ezek után nézzük meg példaként a *MessageBox* API függvény használatát.

Példa:

```
using System;
using System.Runtime.InteropServices;
class dllhasznál
{
    [DllImport("user32.dll")]
    static extern int MessageBox(int hwnd,string msg,
                                 string caption, int type);
    public static void Main()
    {
        MessageBox(0,"Hajrá Fradi !","Ez az ablakfelirat!",0);
    }
}
```

A program futtatása után az alábbi rendszer-üzenetablak jelenik meg:

22. ábra

XV.2. Mutatók használata

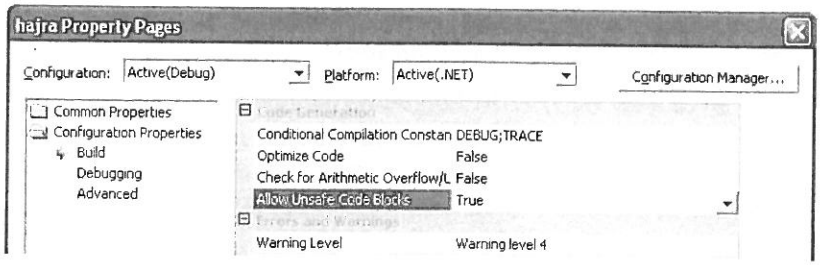
Ahogy korábban is volt szó róla, a keretrendszer a C++ jellegű mutatók használatát nem támogatja. Előfordulhat viszont az, hogy külső erőforrások eléréséhez, illetve azok adatai miatt szükség lehet nem menedzselt, nem biztonságos környezet engedélyezésére. Ebben a környezetben aztán a C++ nyelvben használt mutatófogalom használható.

A nyelv egy függvényt vagy egy utasításblokkot tud nem biztonságossá, nem felügyelt kódrészletté nyilvánítani az *unsafe* kulcsszó használatával.

Emellett egy menedzselt adatot *fixed* jelzővel tudunk ellátni, ha azt akarjuk, hogy a GC által felügyelt területből egy típushoz (menedzselt típus) nem biztonságos mutató hozzáférést kapjunk. Ez természetesen óvatos használatot kíván, hiszen könnyen előfordulhat, hogy az objektumunk a Garbage Collection eredményeként már régen nincs, mikor mi még mindig a mutatójával bűvészkednénk!

A mutatókat csak *unsafe* blokkban használhatjuk.

Ahhoz, hogy a fordító engedélyezze az unsafe blokkot, a projekttulajdonságok között be kell állítani az *„Allow Unsafe Code Blocks"* opciót igazra, ahogy az a következő *Tulajdonság* ablakban is látszik.



23. ábra

Ezt a beállítást parancssori környezetben a *csc* fordítónak az */unsafe* kapcsolója használatával érhetjük el.

Ezek után nézzünk egy klasszikus C++ nyelvszerű maximumérték meghatározást. Az alábbi példában a *max* függvény egy egész vektor legnagyobb értékét határozza meg.

Példa:

```
using System;
class unmanaged
{
    unsafe int max(int* v, int db)
    {
        int i=0;
        int m=v[i++];
        while(i<db)
        {
            if (m<v[i]) m=v[i];
            i++;
        }
        return m;
    }
    int[] s=new int[10]{1,2,3,4,5,0,21,11,1,15};
    unsafe public static void Main()
    {
        int h=0;
        unmanaged u=new unmanaged();
        fixed (int* m= &u.s[0])
        {
            h=u.max(m,10);
        }
        Console.WriteLine(h);
    }
}
```

XV.3. Feladatok

1. Mit jelent a nem felügyelt kód *(unsafe)*?
2. Hogyan tudunk Win32 API függvényt meghívni?
3. Mi a *fixed* változó?
4. Hogyan használhatunk mutatókat egy C# programban?
5. Készítsen unsafe függvényt, amelyik a paraméter vektort nagyság szerint sorbarendezi!

'unsafe kapcso-

ıérték meghatá-
ıagyobb értékét

ırt nagyság

XVI. Grafikus alkalmazások alapjai

A mai grafikus felhasználói felületeken az egyik leginkább kedvelt vagy elvárt alkalmazáskészítési lehetőség a grafikus programok készítése. Emellett az is elmondható, hogy egy program futtatását nem biztos, hogy a helyi gépen szeretnénk végezni. Az internet jelenlegi elterjedését figyelembe véve, egyre gyakrabban merül fel az az igény, hogy az alkalmazást bárki elérhesse egy szabványos internetböngésző program (Internet Explorer, Netscape, Opera stb.) segítségével.

Miután megismertük a korábbi fejezetekben a C# nyelvi és legfontosabb keretrendszeri szolgáltatásait, befejezésképpen nézzünk meg egy-egy példát Windows alapú és „Webes" alapú alkalmazások készítésére.

XVI.1. Windows alkalmazások alapjai

Ahogy eddig is láttuk, a legfontosabb alkalmazási típusok készítéséhez a fejlesztőkörnyezet kész sablont bocsát a fejlesztők rendelkezésére. Így van ez ebben az esetben is.

Új alkalmazás *(project)* készítésekor a *„Windows Application"* sablont választva kapjuk a kicsit preparált forráskódot. Ez az állomány *form1.cs* névre hallgat, és valójában a programot adó *Main* függvény törzse ki van töltve:

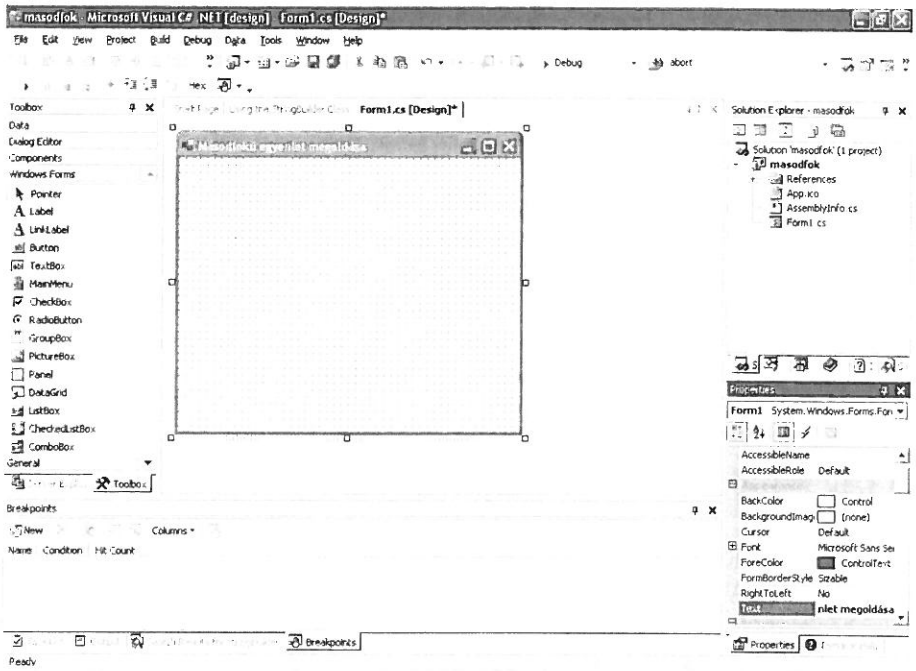
```
static void Main()
{
    Application.Run(new Form1());
}
```

Ez a kódsor azt jelenti, hogy a form utódosztályunk *(Form1)* által képviselt grafikus felület illeszkedjen az operációs rendszer felügyeletébe, és az ablak jelenjen meg. Ez az ablak először természetesen üres, a fő feladat éppen az, hogy megfelelő tartalommal lássuk el, ezáltal elkészítve a kívánt programot.

A program elkészítésében a legnagyobb segítséget a grafikus könyvtári elemek, a *Windows Forms* névtér objektumai (form ablak, címke, nyomógomb stb.) adják. Ezt a lehetőséghalmazt a *Toolbox* ablak mutatja, amit a „rajzszög" segítségével gyakran a képernyőre helyezünk.

Készítsünk a legjellemzőbb tulajdonságok bemutatására egy másodfokú egyenletet megoldó programot. Az új projekt nevének adjuk meg a *masodfok* nevet, válasszuk ki a *Windows Application* sablont, majd a kapott felületre rajzszögezzük ki a *Toolbox* ablakot.

Az így kapott képernyő a következőképpen néz ki:



24. ábra

A forrásállományt megnézve azt láthatjuk, hogy ebben az állapotában a programunk valójában egy form (ablak) objektumból áll *(new Form1())*. Ezt a *Tulajdonságok (Properties)* ablak lenyíló mezőjéből is megállapíthatjuk, hiszen nem tudunk másik objektumot kiválasztani.

A *Tulajdonságok (Properties)* ablakban tervezéskor állíthatjuk be a kiválasztott komponensünk legjellemzőbb tulajdonságait, kezdő adatait. Ezek a tulajdonságok futás közben is hasonló módon megváltoztathatók.

Az egyszerű adatok mellett ez az ablak ad lehetőséget egyes objektumok eseménykezelő paraméterének beállítására. Ez azért lényeges, mert ebben a környezetben a programok valódi tevékenységét ezek a függvények végzik. Így valójában programkészítés címén kicsit egyszerűsítve nincs másról szó, mint hogy olyan grafikus elemekkel építsük fel a programablakot, amelyek eseménykezelői éppen a kívánt feladatot oldják meg.

Ezek után nézzük a célul kitűzött egyszerű feladatot, a másodfokú egyenlet megoldását, amin keresztül a legjellemzőbb lépéseket szemléltetni tudjuk.

Mielőtt nekifognánk a megoldásnak, le kell szögeznünk, hogy a karakteres felületen használt beolvasási és kírási lehetőségek nem használhatóak. Erre a

célra a *Toolbo*
kiírásra a címk
Ezen elem
tét, ahol a cím
paraméterek b
A formra
jeleníteni kívá
értékét pedig t
Ezek alapj

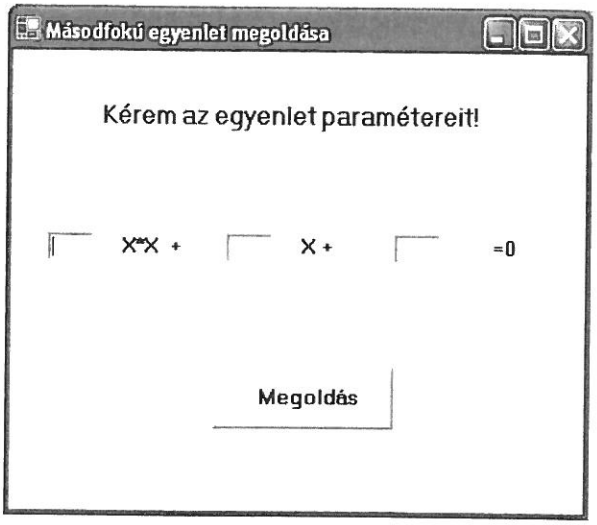
Azt term
megfelel.
A fenti g
lasztómezőjé
változó névv
szer automati
a programter
ezen változó
tehetjük meg
A három
neveket adta
ablakot repre

célra a *Toolbox* ablak elemeit tudjuk használni. A leggyakrabban használt elem kiírásra a címke *(Label)*, míg beolvasásra a szövegdoboz *(TextBox)*.

Ezen elemek segítségével alakítsuk ezután ki a program felhasználói felületét, ahol a címkékkel információt írunk ki, míg a szöveges beolvasó elemek a paraméterek beolvasását biztosítják.

A formra tegyünk címkéket, és a címke *Text* tulajdonság mezőjébe a megjeleníteni kívánt szöveget írjuk bele. A szövegmezők alapértelmezett *Text* mező értékét pedig töröljük ki.

Ezek alapján egy kevés munkával az alábbi felület alakítható ki:



25. ábra

Azt természetesen nem állítom, hogy ez a legszebb kialakítás, de a célnak megfelel.

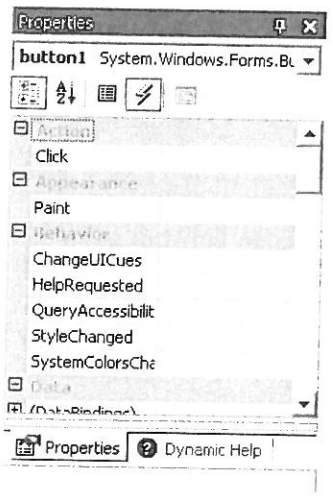
A fenti grafikus tervezés után láthatjuk a *Tulajdonság* ablak objektum kiválasztómezőjében, hogy minden egyes önálló vezérlő *(label, textbox)* egy-egy változó névvel jelenik meg. Ezeket a neveket (*label1*, *label2*, stb.) a keretrendszer automatikusan adja, és ha szükségünk van ezek későbbi használatára, akkor a programtervezési szempontok figyelembevételével adjunk 'beszédes nevet' ezen változóknak. Ezt a *Tulajdonság* ablak *Name* mezőjének módosításával tehetjük meg.

A három együttható beolvasását biztosító textmezőnek rendre az *a*, *b*, *c* neveket adtam, míg a megoldást megjelenítő címkének a *megoldas* nevet. Az ablakot reprezentáló C# nyelvi forráskód az alábbiak szerint módosul.

```
namespace masodfok
{
    /// <summary>
    /// Summary description for Form1.
    /// </summary>
    public class Form1 : System.Windows.Forms.Form
    {
        private System.Windows.Forms.Label label1;
        private System.Windows.Forms.Label label2;
        private System.Windows.Forms.Label label3;
        private System.Windows.Forms.Label label4;
        private System.Windows.Forms.Button button1;
        private System.Windows.Forms.TextBox a;
        private System.Windows.Forms.TextBox b;
        private System.Windows.Forms.TextBox c;
        private System.Windows.Forms.Label megoldas;
        ...
```

Ezeket a bejegyzéseket a keretrendszer automatikusan elvégzi. Azonban meg kell jegyezni, hogy a vezérlőink elnevezését csak ebben a kódrészben végzi el a keretrendszer, így ha utólag nevezünk át vezérlőket, akkor a programkódbeli változásokról magunknak kell gondoskodnunk.

A program tényleges megoldását az egyetlen nyomógomb eseménykezelő függvénye fogja elvégezni. A vezérlőelem eseményeit az alábbi *Tulajdonság* ablakban láthatjuk.



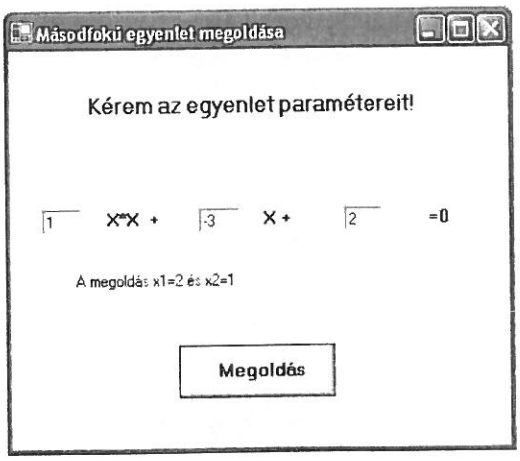
26. ábra

Kettőt kattintva a nyomógombra, a keretrendszer beállítja a nyomógomb *Click* eseménykezelőjét, a forráskódba beírja ennek a függvénynek a keretét, és számunkra nem marad más hátra, mint a valódi programkódot a függvény törzsébe beírni. A feladat ismertsége megengedi, hogy különösebb magyarázat nélkül lássuk az eseménykezelő függvény törzsét:

```
private void button1_Click(object sender, System.EventArgs e)
{
    double a1=Convert.ToDouble(a.Text);
    double b1=Convert.ToDouble(b.Text);
    double c1=Convert.ToDouble(c.Text);
    double m1,m2;
    double d=b1*b1-4*a1*c1;          // diszkrimináns
    if (d<0)
        megoldas.Text="Nincs megoldás, a diszkrimináns negatív.";
    else
    {
        m1=(-b1+Math.Sqrt(d))/2*a1;
        m2=(-b1-Math.Sqrt(d))/2*a1;
        megoldas.Text=String.Format("A megoldás x1={0} és
                                                    x2={1}",m1,m2);
    }
}
```

A programot futtatva, miután beírjuk a megfelelő együtthatókat, az alábbi formában kapjuk meg az eredményt.

(Természetesen további finomítás ráférne erre a programra, de ennek elvégzését a kedves Olvasóra bízom.)



27. ábra

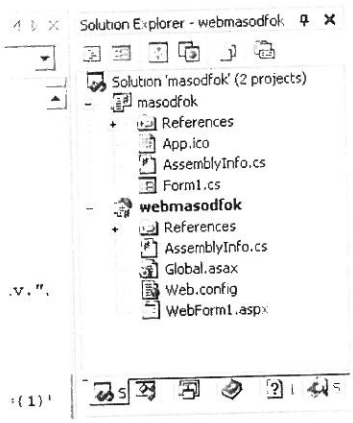
XVI.2. Webes alkalmazások alapjai

Napjainkban az internetes elérhetőség, rendelkezésre állás már-már szinte alapkövetelmény. Ennek a felhasználói igénynek a kielégítésére a keretrendszer lehetőséget ad Web alapú alkalmazások készítésére.

Az ilyen jellegű alkalmazás készítésének alapfeltétele az, hogy egy web kiszolgáló eléréséhez megfelelő jogosultsággal rendelkezzünk. Ez a gyakorlatban az alábbiakat jelenti:

- Azon a gépen ahol fejleszteni szeretnénk, először fel kell installálni az Internet Information Service (IIS) szolgáltatást.
- Fel kell installálni a Visual Studio.NET alkalmazást.
- Ez létrehoz két felhasználói csoportot a számítógépen, *Debugger Users* és *VS Developers* névvel.
- Azokat a fejlesztőket rakjuk bele ezekbe a csoportokba, amelyektől ilyen alkalmazások fejlesztését várjuk. (A *Debugger Users* csoportba minden fejlesztőt bele kell rakni, különben az operációs rendszer a nyomkövetési módban fordított állományt nem engedi a keretrendszerből futtatni!)

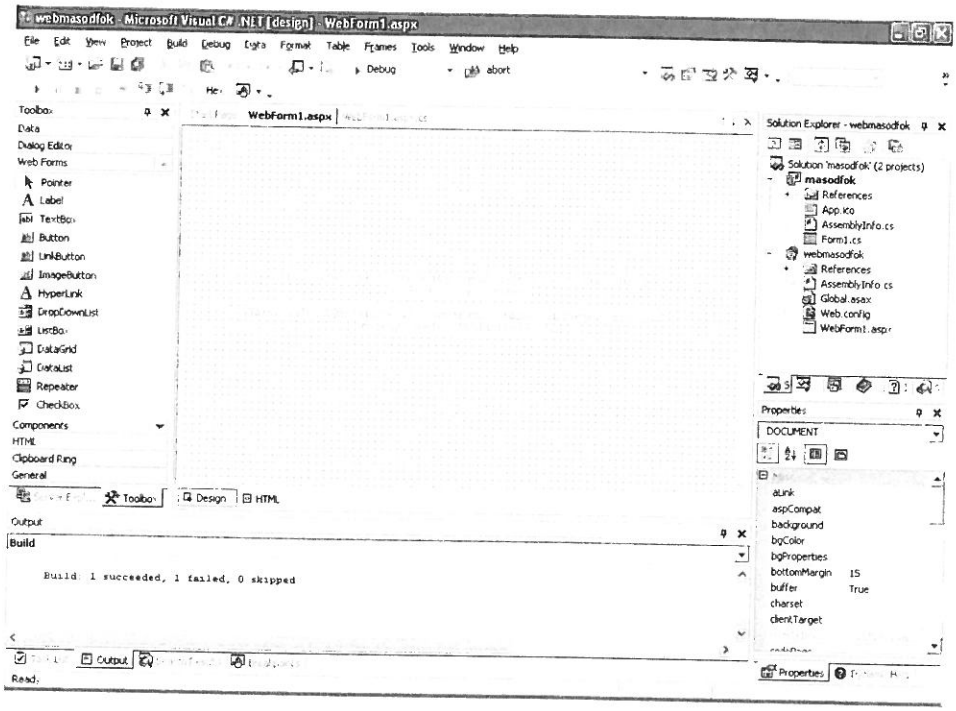
Ha a fenti feltételek megvannak, akkor *ASP.NET Web Application* sablont választva készíthetünk webes alkalmazást. Ezek alapján készítsünk egy *webmasodfok* projektet a fejlesztőkörnyezetben, ahogyan az a következő képen látszik, az előző *masodfok* projekt mellett elhelyezve. (A keretrendszer *Solution* fogalma több projektet enged egy keretbe foglalni, hiszen gyakran előfordul, hogy egy feladatmegoldást nem egy projekttel célszerű megadni. Példánk esetében nincs erről szó.)



28. ábra

Miután megadtuk a nevet, elkészül a következő üres weboldal. A létrehozott *Webform1.aspx* az üres HTML oldal (ezért is van HTML nézete) és a hátterében meghúzódó programfájl *(Webform1.aspx.cs)*. Látható, hogy a *Toolbox* ablakban a korábbi *Windows Forms* felirat helyett *Web Forms* olvasható, mutatva, hogy ezek a vezérlők web alapú alkalmazásokhoz használhatóak. A *Tulajdonság* ablak egyetlen objektuma a *DOCUMENT* objektum lesz, ami természetesen magának a HTML dokumentumnak felel meg. Ezek tulajdonságértékeit módosíthatjuk, például a *Title* mező értékét, megadva ezzel a weboldal címkéjét, vagy a *bgColor* paraméterrel beállítva a kívánt háttérszínt.

A kapott képernyő a következő alakú lesz:



29. ábra

A *Windows Form* megoldáshoz hasonlóan készítsük el a másodfokú egyenlet megoldását ebben a környezetben is. Ehhez első lépésként alakítsuk ki a programunk felületét az előző példához hasonlóan, majd a nyomógomb eseménykezelő függvényét definiáljuk.

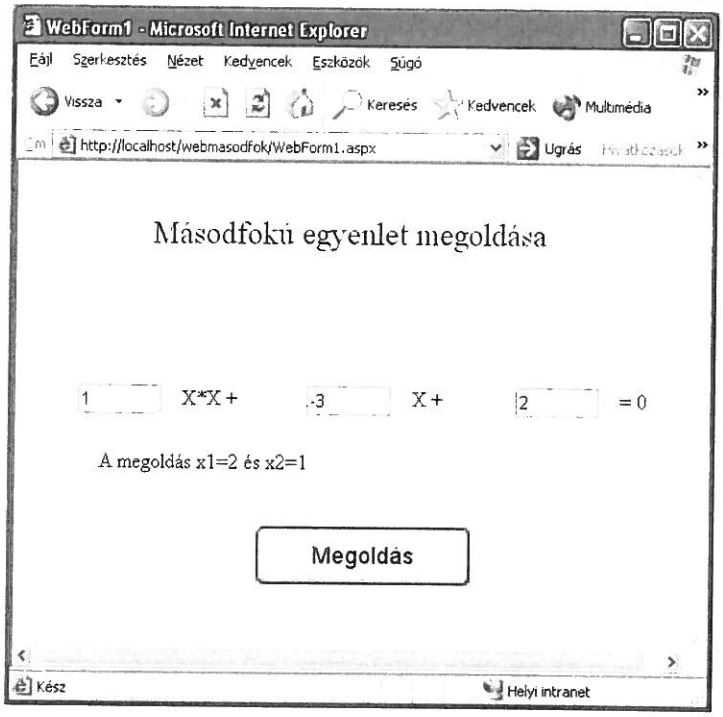
A kapott forráskód *(Webform1.aspx.cs)* a következő lesz:

```
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;

namespace webmasodfok
{
    /// <summary>
    /// Summary description for WebForm1.
    /// </summary>
    public class WebForm1 : System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.Label Label1;
        protected System.Web.UI.WebControls.Label Label2;
        protected System.Web.UI.WebControls.Label Label3;
        protected System.Web.UI.WebControls.Label Label4;
        protected System.Web.UI.WebControls.TextBox a;
        protected System.Web.UI.WebControls.TextBox b;
        protected System.Web.UI.WebControls.TextBox c;
        protected System.Web.UI.WebControls.Label megoldas;
        protected System.Web.UI.WebControls.Button Button1;

        private void Page_Load(object sender, System.EventArgs e)
        {
            // Put user code to initialize the page here
        }

        #region Web Form Designer generated code
        override protected void OnInit(EventArgs e)
        {
            //
            // CODEGEN: This call is required by the ASP.NET
                                                Web Form Designer.
            //
            InitializeComponent();
            base.OnInit(e);
        }
```

```
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>

        private void InitializeComponent()
        {
            this.Button1.Click += new
                        System.EventHandler(this.Button1_Click);
            this.Load += new System.EventHandler(this.Page_Load);
        }
        #endregion

        private void Button1_Click(object sender, System.EventArgs e)
        {
            double a1=Convert.ToDouble(a.Text);
            double b1=Convert.ToDouble(b.Text);
            double c1=Convert.ToDouble(c.Text);
            double m1,m2;
            double d=b1*b1-4*a1*c1;        // diszkrimináns
            if (d<0)
                megoldas.Text="Nincs megoldás, a diszkrimináns
                                                        negatív.";
            else
            {
                m1=(-b1+Math.Sqrt(d))/2*a1;
                m2=(-b1-Math.Sqrt(d))/2*a1;
                megoldas.Text=String.Format("A megoldás x1={0}
                                            és x2={1}",m1,m2);
            }

        }
    }
}
```

A program fordítása és futtatása után a 30. ábrán látható Internet Explorer böngészőben futó alkalmazást kapjuk, vagyis a célunkat elértük.

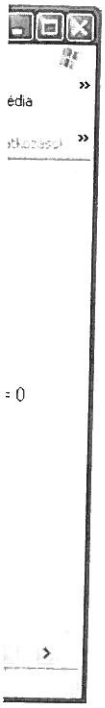
Ebben a fejezetben – kitekintésként – csak a legfontosabb környezeti beállításokról, fogalmakról tudtunk szólni. A grafikus alkalmazások lehetőségeit, a segítségünkre lévő grafikus vezérlőelemek tulajdonságait, webes, mobil és/vagy adatbázis kapcsolattal rendelkező alkalmazások készítését egy következő kötet keretében szeretnénk megmutatni.



30. ábra

XVI.3. Feladatok

1. Mi a feladata a grafikus alkalmazás *Main* függvényének?
2. Mi a különbség a Windows és az ASP.NET alkalmazás között?
3. Milyen feltételeknek kell teljesülni ahhoz, hogy ASP.NET alkalmazást tudjunk készíteni?
4. Készítsen alkalmazást, amely egy téglatest 3 oldalát beolvasva meghatározza a térfogatát és felszínét! (Használjon címkéket és szövegmezőket!)
5. Készítse el az előző feladatot webes alkalmazásként is.



Irodalomjegyzék

1. Brian W. Kernigham, Dennis M. Ritchie : A C programozási nyelv
   Műszaki Könyvkiadó, Budapest 1985, 1988

2. Bjarne Stroustrup: C++ programming language
   AT&T Bell Lab, 1986

3. Tom Archer : Inside C#
   MS Press, 2001

4. Microsoft C# language specifications
   MS Press, 2001

5. John Sharp, Jon Jagger: Microsoft Visual C#.NET
   MS Press, 2002

6. Illés Zoltán: A C++ programozási nyelv
   ELTE IK, Mikrológia jegyzet, 1995-...

7. David S.Platt: Bemutatkozik a Microsoft.NET
   Szak Kiadó, 2001

8. Illés Zoltán: A C# programozási nyelv és környezete
   Informatika a felsőoktatásban 2002, Debrecen

9. David Chappell: Understanding .NET
   Addison-Wesley, 2002