Reiter István

C#

2009, 0.91 verzió

Tartalomjegyzék

1. Bevezető

Napjainkban egyre nagyobb teret nyer a .NET Framework és egyik fő nyelve a C#.
Ez a jegyzet abból a célból született, hogy megismertesse az olvasóval ezt a nagyszerű technológiát.
A jegyzet a C# 2.0 és 3.0 verziójával is foglalkozik, ha nincs külön feltüntetve akkor az adott nyelvi elem gond nélkül használható a korábbi verzióban.
Néhány fejezet feltételez olyan tudást amely alapját egy későbbi fejezet képezi, ezért ne essen kétségbe a kedves olvasó, ha valamit nem ért, egyszerűen olvasson tovább és térjen vissza a kérdéses fejezethez, ha rátalált a válaszra.
Bármilyen kérést, javaslatot illetve hibajavítást szívesen várok a reiteristvan@gmail.com email címre.

1.1 A jegyzet jelölései

Forráskód: szürke alapon, kerettel

Megjegyzés: fehér alap, kerettel

1.2 Jogi feltételek

A mindenkori legfrissebb változat letölthető a következő oldalakról:

- http://people.inf.elte.hu/reiter_i/sharp.html
- http://devportal.hu/groups/fejlesztk/media/p/1122.aspx

A következő frissítés időpontja: 2009. Április 12.

2. Microsoft .NET Framework

A kilencvenes évek közepén a Sun MicroSystems kiadta a Java platform első nyilvános változatát. Az addigi programnyelvek/platformok különböző okokból nem tudták felvenni a Java –val a versenyt, így számtalan fejlesztő döntött úgy, hogy a kényelmesebb és sokoldalúbb Java –t választja.
Részben a piac visszaszerzésének érdekében a Microsoft a kilencvenes évek végén elindította a *Next Generation Windows Services* fedőnevű projektet, amelyből aztán megszületett a .NET.

2.1 A .NET platform

Maga a .NET platform a Microsoft, a Hewlett Packard, az Intel és mások közreműködésével megfogalmazott CLI (Common Language Infrastructure) egy implementációja. A CLI egy szabályrendszer, amely maga is több részre oszlik:
A CTS (Common Type System) az adatok kezelését, a memóriában való megjelenést, az egymással való interakciót, stb. írja le.
A CLS (Common Language Specification) a CLI kompatibilis nyelvekkel kapcsolatos elvárásokat tartalmazza.
A VES (Virtual Execution System) a futási környezetet specifikálja, nevezik CLR -nek (Common Language Runtime) is.
A .NET nem egy programozási nyelv, hanem egy környezet. Gyakorlatilag bármilyen programozási nyelvnek lehet .NET implementációja. Jelenleg kb. 50 nyelvnek létezik hivatalosan .NET megfelelője, nem beszélve a számtalan hobbifejlesztésről.

2.1.1 MSIL/CIL

A “hagyományos” programnyelveken – mint pl. a C++ - megírt programok ún. natív kódra fordulnak le, vagyis a processzor számára – kis túlzással – azonnal értelmezhetőek. Ezeknek a nyelveknek az előnye a hátránya is egyben. Bár gyorsak, de rengeteg hibalehetőség rejtőzik a felügyelet nélküli (unmanaged) végrehajtásban.
A .NET (akárcsak a Java) más úton jár, a fordító egy köztes nyelvre (Intermediate Language) fordítja le a forráskódot. Ez a nyelv a .NET világában az MSIL illetve a szabványosítás után CIL (**M**icro**s**oft/**C**ommon IL) – különbség csak az elnevezésben van.

2.1.2 Fordítás/futtatás

A natív programok ún. gépi kódra fordulnak le, míg a .NET forráskódokból egy CIL nyelvű futtatható állomány keletkezik. Ez a kód a feltelepített .NET Framework –nek szóló utasításokat tartalmaz. Amikor futtatjuk ezeket az állományokat először az ún. JIT (just–in–time) fordító veszi kezelésbe és lefordítja őket gépi kódra, amit a processzor már képes kezelni.
Amikor “először” lefordítjuk a programunkat akkor egy ún. *Assembly* (vagy szerelvény) keletkezik. Ez tartalmazza a felhasznált illetve megvalósított típusok adatait (ez az ún. *Metadata*) amelyeket a futtató környezet fel tud használni a futtatáshoz (az osztályok neve, metódusai, stb…). Egy Assembly egy vagy több file –ból is állhat.

2.1.3 BCL

A .NET Framework telepítésével a számítógépre kerül – többek közt – a BCL (Base Class Library), ami az alapvető feladatok (file olvasás/ írás, adatbázis kezelés, adatszerkezetek, stb…) elvégzéséhez szükséges eszközöket tartalmazza. Az összes többi könyvtár (ADO.NET, WCF, stb…) ezekre a könyvtárakra épül.

2.2 A C# programozási nyelv

A C# (ejtsd: Szí-sárp) a Visual Basic mellett a .NET fő programozási nyelve. 1999 –ben Anders Hejlsberg vezetésével kezdték meg a fejlesztését.
A C# tisztán objektumorientált, típusbiztos, általános felhasználású nyelv. A tervezésénél a lehető legnagyobb produktivitás elérését tartották szem előtt.
A nyelv elméletileg platformfüggetlen (létezik Linux és Mac fordító is), de napjainkban a legnagyobb hatékonyságot a Microsoft implementációja biztosítja.

2.3 Alternatív megoldások

A Microsoft .NET Framework jelen pillanatban csak és kizárólag Microsoft Windows operációs rendszerek alatt elérhető. Ugyanakkor a szabványosítás után a CLI specifikáció nyilvános és bárki számára elérhető lett, ezen ismeretek birtokában pedig több független csapat vagy cég is létrehozta a saját CLI implementációját, bár eddig még nem sikerült teljes mértékben reprodukálni az eredetit. Ezt a célt nehezíti, hogy a Microsoft időközben számos a specifikációban nem szereplő változtatást végzett a keretrendszeren.

2.3.1 SSCLI

Az SSCLI (Shared Source Common Language Infrastructure) vagy korábbi nevén Rotor a Microsoft által fejlesztett nyílt forrású, keresztplatformos változata a .NET Frameworknek (tehát nem az eredeti lebutított változata). Az SSCLI Windows, FreeBSD és Mac OSX rendszereken fut.
Az SSCLI –t kimondottan tanulási célra készítette a Microsoft, ezért a liszensze engedélyez mindenfajta módosítást, egyedül a piaci értékesítést tiltja meg.
Ez a rendszer nem szolgáltatja az eredeti keretrendszer teljes funkcionalitását, jelen pillanatban valamivel a .NET 2.0 mögött jár.
Az SSCLI project jelen pillanatban megszűnni – vagy legalábbis időlegesen leállni – látszik. Ettől függetlenül a forráskód és a hozzá tartozó dokumentációk rendelkezésre állnak, letölthetőek a következő webhelyről:

- http://www.microsoft.com/downloads/details.aspx?FamilyId=8C09FD61-3F26-4555-AE17-3121B4F51D4D&displaylang=en

2.3.2 Mono

A Mono projekt szülőatyja Miguel de Icaza 2000 –ben kezdte meg a fejlesztést és egy évvel később mutatta be ez első kezdetleges C# fordítót. A Ximian (amelyet Icaza és Nat Friedman alapított) felkarolta az ötletet és 2001 júliusában hivatalosan

is elkezdődött a Mono fejlesztése. 2003 –ban a Novell felvásárolta a Ximian –t, az 1.0 verzió már Novell termékként készült el, egy évvel később.
A legutolsó verzió (2.0.1) amely 2008 októberében látott napvilágot. Ez a változat már a teljes .NET 2.0 –át magába foglalja, illetve a C# 3.0 képességeit is részlegesen támogatja (pl. van LINQ to XML illetve LINQ to Objects). Ugyanez a verzió tartalmazza a Novell SilverLight alternatíváját, amit MoonLight –ra kereszteltek, ennek fejlesztésében a Novell segítségére van a Microsoft is. A Mono saját fejlesztőeszközzel is rendelkezik a MonoDeveloppal.

Ezzel a változattal párhuzamosan fejlesztés alatt áll a .NET 3.0 –t is támogató rendszer, a project kódneve Olive (http://www.mono-project.com/Olive).

A Mono Windows, Linux, UNIX, BSD, Mac OSX és Solaris rendszereken elérhető. Napjainkban a Mono mutatja a legígéretesebb fejlődést, mint a Microsoft .NET jövőbeli ellenfele illetve keresztplatformos társa.
A Mono emblémája egy majmot ábrázol, a szó ugyanis spanyolul majmot jelent.
A Mono hivatalos oldala:

- http://www.mono-project.com/Main_Page

2.3.3 DotGNU

A DotGNU a GNU projekt része, amelynek célja egy ingyenes és nyílt alternatívát nyújtani a Microsoft implementáció helyett. Ez a projekt – szemben a Mono –val – nem a Microsoft BCL –el való kompatibilitást helyezi előtérbe, hanem az eredeti szabvány pontos és tökéletes implementációjának a létrehozását.
A DotGNU saját CLI megvalósításának a Portable .NET nevet adta.
A jegyzet írásának idején a projekt leállni látszik.
A DotGNU hivatalos oldala:

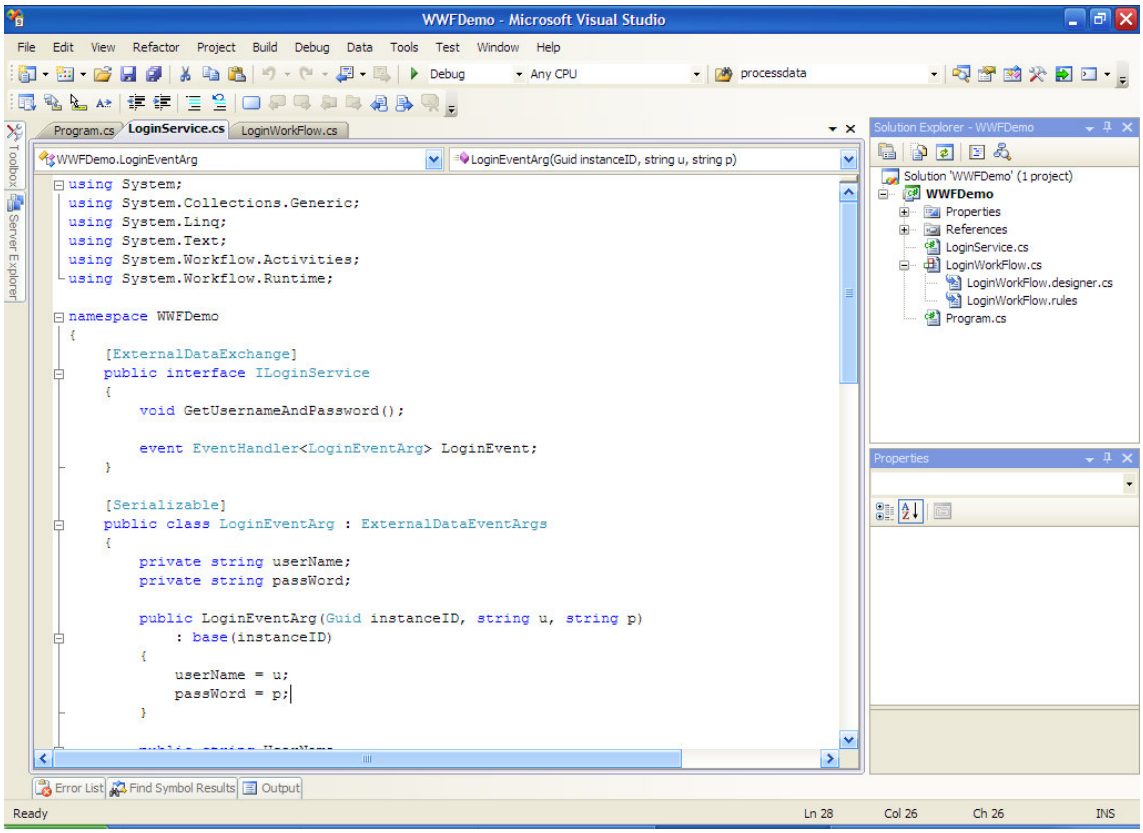
- http://www.gnu.org/software/dotgnu/

3. Fejlesztői környezetek

Ebben a fejezetben néhány IDE –t (Integrated Development Environment), azaz Integrált Fejlesztői Környezetet vizsgáluk meg. Természetesen ezek nélkül is lehetséges fejleszteni, azonban egy jó fejlesztőeszköz hiánya jelentősen meggátolja a gyors és produktív programozást. Ugyanakkor a jegyzet első felében egy hagyományos szövegszerkesztővel és a parancssorral is lehet gond nélkül haladni.

3.1 Microsoft Visual Studio

Valószínűleg a legelterjedtebb IDE a .NET programozásához. A Visual Studio termékcsalád része az Express sorozat, amely teljesen ingyenes mindenki számára, egy Express eszközzel készített program el is adható, anélkül, hogy bármiféle liszenszet vásárolnánk. Ez a termékvonal némileg kevesebb funkcionalitással rendelkezik, mint nagyobb testvére, ugyanakkor kiválóan alkalmas hobbifejlesztésre (de akár “rendes” kereskedelmi program készítésére is). A jegyzet a grafikus felületű alkalmazások készítésénél Visual Studio –t használ majd, ezért érdemes tájékozódni az egyes fejlesztőeszközök különbségeiről. A képen a Visual Studio 2008 felülete látható:

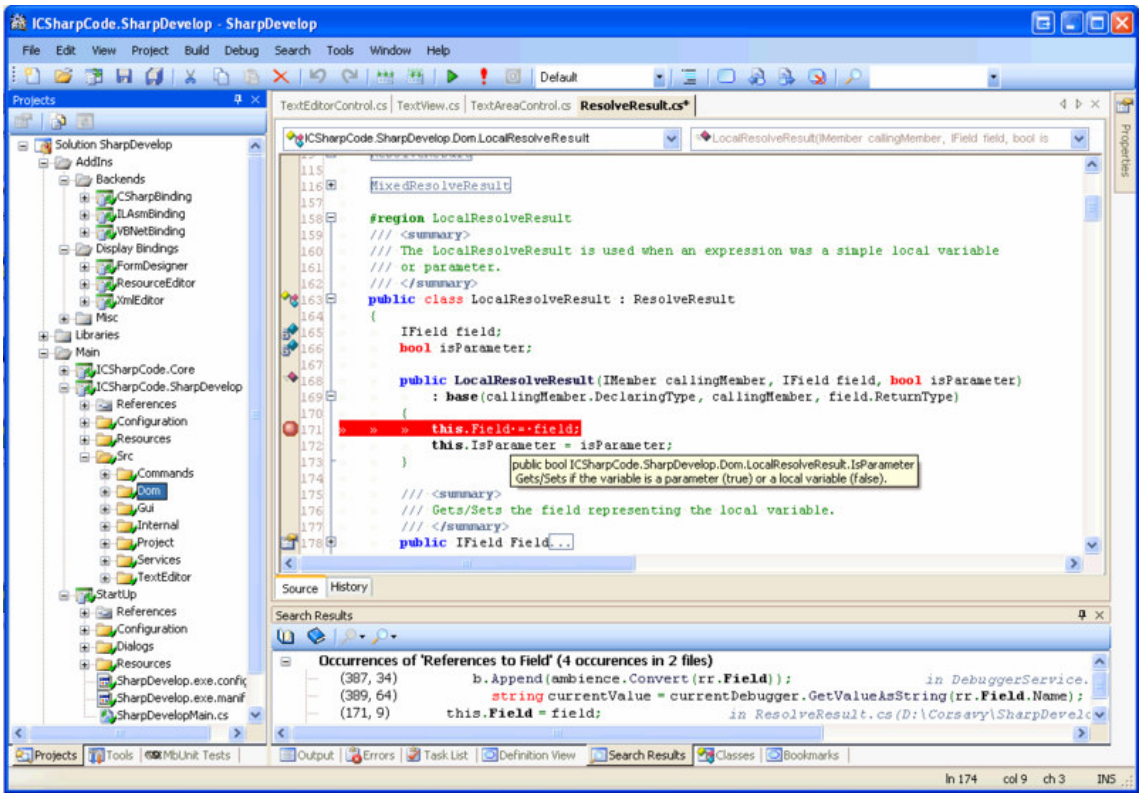


Jobb oldalon fölül a Solution Explorer látható, ez a projektünk file –jait tartalmazza. Alatta a Properties ablak, ez főként a grafikus designer ablaknál használható, az egyes vezérlőelemek tulajdonságait (név, szélesség, magasság, stb…) állíthatjuk be. Végül a legnagyobb területet a kódszerkesztő ablak foglalja el.

A Visual Studio tartalmazza az ún. IntelliSense rendszert, amely automatikusan felajánlja az elkezdett metódusok/változók/osztályok/stb. nevének kiegészítését.
Új projekt létrehozásához Kattintsunk a File menüre, válasszuk ki a New menüpontot azon belül pedig a Projekt –et (vagy használjuk a Ctrl+Shift+N billentyűkombinációt). Ha nem az lenne megnyitva, akkor kattintsunk a bal oldali listában a C# elemre. Grafikus felületű alkalmazás készítéséhez válasszuk ki a Windows Application vagy a WPF Application sablont. Konzolalkalmazás készítéséhez (a jegyzet első részében főként ilyet írunk majd) a Console Application sablonra lesz szükségünk.
Mielőtt lefordítjuk a programunkat, kiválaszthatjuk, hogy Debug vagy Release módban fordítsunk, előbbi tartalmaz néhány plusz információt a hibakeresés elősegítéséhez (így egy kicsit lassabb), utóbbi a már kész programot jelenti.
Lefordítani a programot a Build menüpont Build Solution parancsával (vagy az F6 gyorsbillentyűvel) tudjuk, futtatni pedig a Debug menü Start Debugging (vagy az F5 billentyű lenyomásával illetve a kis zöld háromszögre való kattintással) parancsával. Ez utóbbi automatikusan lefordítja a projektet, ha a legutolsó fordítás után megváltozott valamelyik file.

3.2 SharpDevelop

A SharpDevelop egy nyílt forráskódú ingyenes alternatíva Windows operációs rendszer alatti .NET fejlesztéshez.



A legfrisebb verziót idén februárban adták ki, ez már támogatja a C# 3.0 verzióját, valamint a Visual Basic.NET, Boo, IronPython és F# programozási nyelveket is.

A SharpDevelop hivatalos oldala:

- http://www.sharpdevelop.com/OpenSource/SD/

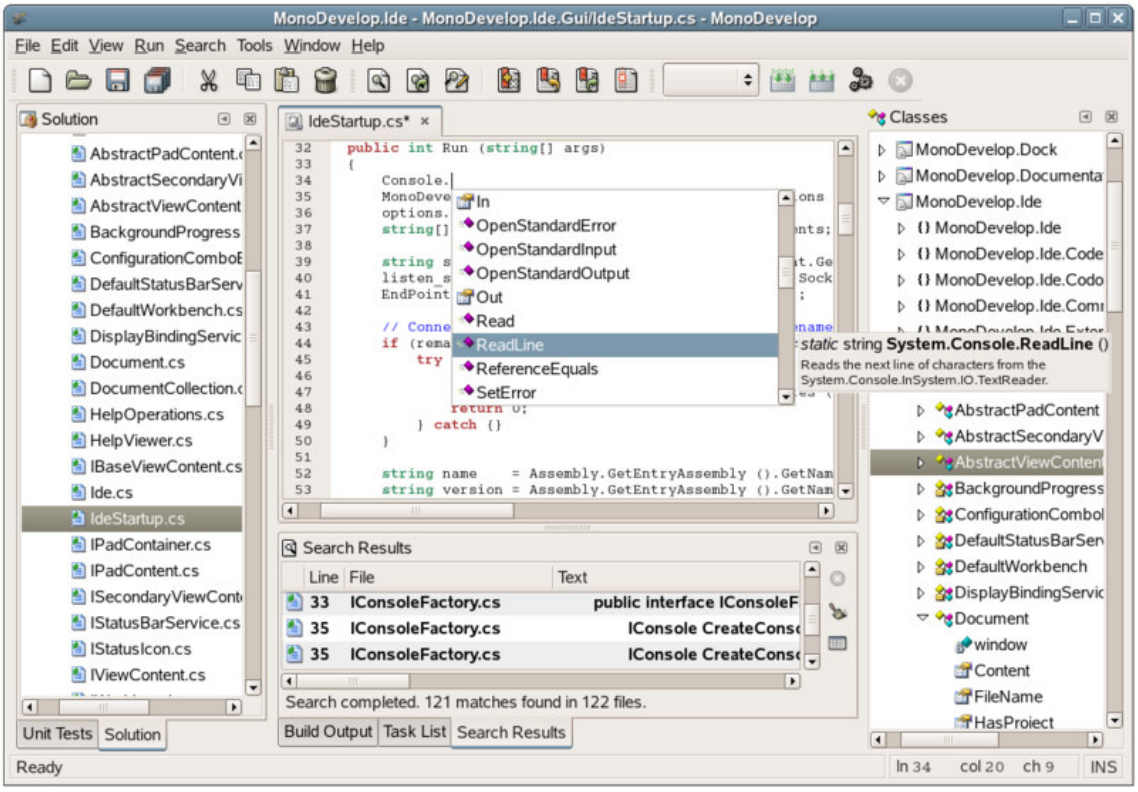
Érdekesség, hogy a fejlesztők egy könyvet is írtak a fejlesztés menetéről és ez a könyv ingyenesen letölthető. Eredetileg a Wrox kiadó oldaláról lehetett letölteni, de annak megszűnése után átkerült az Apress kiadóhoz. Sajnos úgy tűnik, hogy az Apress nem kívánja nyílvánosságra hozni a könyvet, de természetesen az interneten bármit meg lehet találni, így a következő webhelyről ez is letölthető:

- http://www.mediafire.com/?thnognidwyj

A könyv letöltése nem számít illegális cselekménynek!

3.3 MonoDevelop

A MonoDevelop elsősorban a Mono –hoz készült nyílt forráskódú és ingyenes fejlesztőeszköz. Eredetileg Linux oprációs rendszer alá szánták, de van már Windows, Mac OSX, OpenSolaris és FreeBSD alatt futó változata is.



A MonoDevelop a következő programozási nyelveket támogatja: C#, Java, Visual Basic.NET, Nemerle, Boo, CIL, C, C++.

A MD eredetileg a SharpDevelop –ból származott el, de a mai változatok már kevéssé hasonlítanak az ősre. A MonoDevelop –ot akárcsak a Mono –t a Novell (is) fejleszti.

A MonoDevelop hivatalos oldala:

- http://www.monodevelop.com/

A MonoDevelop egyetlen nagy hátulütővel rendelkezik: Windows operációs rendszer alá nem létezik hozzá telepítő szoftver, a felhasználónak magának kell lefordítania a forráskódot.

4. “Hello C#!”

A híres-hírhedt “Hello World!” program elsőként Dennis Ritchie és Brian Kerningham “A C programozási nyelv” című könyvében jelent meg, és azóta szinte hagyomány, hogy egy programozási nyelv bevezetőjeként ezt a programot mutatják be. Semmi mást nem csinál, mint kiírja a képernyőre az üdvözletet. Mi itt most nem a világot, hanem a C# nyelvet üdvözöljük, ezért ennek megfelelően módosítsuk a forráskódot:

```
using System;

class HelloWorld
{
    static public void Main()
    {
        Console.WriteLine("Hello C#!");
        Console.ReadKey();
    }
}
```

Mielőtt lefordítjuk tegyünk pár lépést a parancssorból való fordítás elősegítésére. Ahhoz, hogy így is le tudjunk fordítani egy forrásfile –t, vagy meg kell adnunk a fordítóprogram teljes elérési útját (ez a mi esetünkben elég hosszú) vagy a fordítóprogram könyvtárát fel kell venni a **PATH** környezeti változóba.

Ez utóbbihoz Vezérlőpult/Rendszer -> Speciális fül/Környezeti változók. A rendszerváltozók listájából keressük ki a Path –t és kattintsunk a Szerkesztés gombra. Most nyissuk meg a Sajátgépet, C meghajtó, Windows mappa, azon belül Microsoft.NET/Framework. Nyissuk meg vagy a v2.0… vagy a v3.5.. kezdetű mappát (attól függően, hogy C# 2.0 vagy 3.0 kell). Másoljuk ki a címsorból ezt a szép hosszú elérést. Vissza a Path –hoz. A változó értéke sorban navigáljunk el a végére, írjunk egy pontosvesszőt (**;**) és illesszük be az elérési utat. Mindent OK –zunk le és kész. Ha van megnyitva konzol vagy PowerShell azt indítsuk újra, utána írjuk be, hogy *csc*. Azt kell látnunk, hogy **Microsoft ® Visual C# 2008 Compiler Version 3.5**…
Itt az évszám és verzió változhat, ez a C# 3.0 üzenete.

Most már fordíthatunk a

```
csc filenev.cs
```

paranccsal (természetesen a szöveges file kiterjesztése „.txt”, így nevezzük át mivel a C# forráskódot tartalmazó file -ok kiterjesztése „.cs”).

Az első sor megmondja a fordítónak, hogy használja a *System* névteret. Ezután létrehozunk egy osztályt, mivel a C# teljesen objektumorientált ezért utasítást csakis osztályon belül adhatunk. A “HelloWorld” osztályon belül definiálunk egy *Main* nevű statikus függvényt, ami a programunk belépési pontja lesz. Minden egyes C# program a Main függvénnyel kezdődik, ezt mindenképpen létre kell hoznunk. Végül meghívjuk a *Console* osztályban lévő *WriteLine(…)* és *ReadKey(…)* függvényeket. Előbbi kiírja a képernyőre a paraméterét, utóbbi vár egy billentyűleütést.

Ebben a bekezdésben szerepel néhány (sok) kifejezés ami ismeretlen lehet, a jegyzet későbbi fejezeteiben mindenre fény derül majd.

4.1 A C# szintaktikája

Amikor egy programozási nyelv szintaktikájáról beszélünk, akkor azokra a szabályokra gondolunk, amelyek megszabják a forráskód felépítését. Ez azért fontos, mert az egyes fordítóprogramok az ezekkel a szabályokkal létrehozott kódot tudják értelmezni.
A C# C-stílusú szintaxissal rendelkezik (azaz a C programozási nyelv szintaxisához hasonlít), ez három fontos kitételt von maga után:

- Az egyes utasítások végén pontosvessző (**;**) áll
- A kis- és nagybetűk különböző jelentőséggel bírnak, azaz a "*program*" és "*Program*" azonosítók különböznek. Ha a fenti kódban *Console.WriteLine(…)* helyett *console.writeline(…)* –t írnánk a program nem fordulna le.
- A program egységeit (osztályok, metódusok, stb.) ún. blokkokkal jelöljük ki, a kapcsos zárójelek (**{**, **}**) segítségével.

4.1.1 Kulcsszavak

Szinte minden programnyelv definiál kulcsszavakat, amelyek speciális jelentőséggel bírnak a fordító számára. Ezeket az azonosítókat a saját meghatározott jelentésükön kívűl nem lehet másra használni, ellenkező esetben a fordító hibát jelez. Vegyünk például egy változót, aminek az "*int*" nevet akarjuk adni. Az "*int*" egy beépített típus neve is, azaz kulcsszó, ezért nem fog lefordulni a program:

```
int int; //hiba
```

A legtöbb fejlesztőeszköz megszínezi a kulcsszavakat (is), ezért könnyű elkerülni a fenti hibát.
A C# 77 darab kulcsszót ismer:

abstract	**default**	**foreach**	**object**	**sizeof**	**unsafe**
as	**delegate**	**goto**	**operator**	**stackalloc**	**ushort**
base	**do**	**if**	**out**	**static**	**using**
bool	**double**	**implicit**	**override**	**string**	**virtual**
break	**else**	**in**	**params**	**struct**	**volatile**
byte	**enum**	**int**	**private**	**switch**	**void**
case	**event**	**interface**	**protected**	**this**	**while**
catch	**explicit**	**internal**	**public**	**throw**	
char	**extern**	**is**	**readonly**	**true**	
checked	**false**	**lock**	**ref**	**try**	
class	**finally**	**long**	**return**	**typeof**	
const	**fixed**	**namespace**	**sbyte**	**uint**	
continue	**float**	**new**	**sealed**	**ulong**	
decimal	**for**	**null**	**short**	**unchecked**	

Ezeken kívűl létezik még 23 darab azonosító, amelyeket a nyelv nem tart fenn speciális használatra, de különleges jelentéssel bírnak. Amennyiben lehetséges kerüljük a használatukat "hagyományos" változók, metódusok, osztályok létrehozásánál:

add	**equals**	**group**	**let**	**remove**	**var**
ascending	**from**	**in**	**on**	**select**	**where**
by	**get**	**into**	**orderby**	**set**	**yield**
descending	**global**	**join**	**partial**	**value**	

Néhányuk a környezettől függően más-más jelentéssel is bírhat, a megfelelő fejezet bővebb információt ad majd ezekről az esetekről.

4.1.2 Megjegyzések

A forráskódba megjegyzéseket tehetünk. Ezzel egyrészt üzeneteket hagyhatunk (pl. egy metódus leírása) magunknak, vagy a többi fejlesztőnek, másrészt a kommentek segítségével dokumentációt tudunk generálni, ami szintén az első célt szolgálja csak éppen élvezhetőbb formában.
Megjegyzéseket a következőképpen hagyhatunk:

```
using System;

class HelloWorld
{
    static public void Main()
    {
        Console.WriteLine("Hello C#"); // Ez egy egysoros komment
        Console.ReadKey();
        /*
         Ez egy többsoros komment
        */
    }
}
```

Az egysoros komment a saját sora legvégéig tart, míg a többsoros a két "/*" –on belül érvényes. Utóbiakat nem lehet egymásba ágyazni:

```
/*
   /* */
*/
```

Ez a "kód" nem fordul le.
A kommenteket a fordító nem veszi figyelembe, tulajdonképpen a fordítóprogram első lépése, hogy a forráskódból eltávolít minden megjegyzést.

4.2 Névterek

A .NET Framework osztálykönyvtárai szerény becslés szerint is legalább tízezer nevet, azonosítót tartalmaznak. Ilyen nagyságrenddel elkerülhetetlen, hogy a nevek ismétlődjenek. Ekkor egyrészt nehéz eligazodni köztük, másrészt a fordító is

megzavarodhat. Ennek a problémának a kiküszöbölésére hozták létre a névterek fogalmát. Egy névtér tulajdonképpen egy virtuális doboz, amelyben a logikailag összefüggő osztályok, metódusok, stb… vannak. Nyílván könyebb megtalálni az adatbáziskezeléshez szükséges osztályokat, ha valamilyen kifejező nevű névtértben vannak (*System.Data*).

Névteret magunk is definiálhatunk, a *namespace* kulcsszóval:

```
namespace MyNameSpace
{

}
```

Ezután a névterre vagy a program elején a *using* –al, vagy az azonosító elé írt teljes eléréssel hivatkozhatunk:

```
using MyNameSpace;

//vagy

MyNameSpace.Valami
```

A jegyzet első felében főleg a *System* névteret fogjuk használni, ahol más is kell ott jelezni fogom.

5. Változók

Amikor programot írunk, akkor szükség lehet tárolókra, ahová az adatainkat ideiglenesen eltároljuk. Ezeket a tárolókat változóknak nevezzük.
A változók a memória egy(vagy több) cellájára hivatkozó leírók. Egy változót a következő módon hozhatunk létre C# nyelven:

Típus változónév;

A változónév első karaktere csak betű vagy alulvonás jel (_) lehet, a többi karakter szám is. Lehetőleg kerüljük az ékezetes karakterek használatát.

5.1 Típusok

A C# erősen (statikusan) típusos nyelv, ami azt jelenti, hogy minden egyes változó típusának ismertnek kell lennie fordítási időben. A típus határozza meg, hogy egy változó milyen értékeket tartalmazhat illetve mekkora helyet foglal a memóriában.
A következő táblázat a C# beépített típusait tartalmazza, mellettük ott a .NET megfelelőjük, a méretük és egy rövid leírás:

C# típus	**.NET típus**	**Méret (byte)**	**Leírás**
byte	System.Byte	1	Előjel nélküli 8 bites egész szám (0..255)
char	System.Char	1	Egy Unicode karakter
bool	System.Boolean	1	Logikai típus, értéke igaz(1) vagy hamis(0)
sbyte	System.SByte	1	Előjeles 8 bites egész szám (-128..127)
short	System.Int16	2	Előjeles 16 bites egész szám (-32768..32767
ushort	System.Uint16	2	Előjel nélküli 16 bites egész szám (0..65535)
int	System.Int32	4	Előjeles 32 bites egész szám (–2147483647.. 2147483647).
uint	System.Uint32	4	Előjel nélüli 32 bites egész szám (0..4294967295)
float	System.Single	4	Egyszeres pontosságú lebegőpontos szám
double	System.Double	8	Kétszeres pontosság lebegőpontos szám
decimal	System.Decimal	8	Fix pontosságú 28+1 jegyű szám
long	System.Int64	8	Előjeles 64 bites egész szám
ulong	System.Uint64	8	Előjel nélküli 64 bites egész szám
string	System.String	NA	Unicode karakterek szekvenciája
object	System.Object	NA	Minden más típus őse

A forráskódban teljesen mindegy, hogy a “rendes” vagy a .NET néven hivatkozunk egy típusra.

Alakítsuk át a “Hello C#” programot úgy, hogy a kiírandó szöveget egy változóba tesszük:

```
using System;

class HelloWorld
{
    static public void Main()
    {
        //string típusu változó deklarációja, benne a kiírandó szöveg
        string message = "Hello C#";
        Console.WriteLine(message);
        Console.ReadKey();
    }
}
```

A C# 3.0 lehetővé teszi, hogy egy metódus hatókörében deklarált változó típusának meghatározását a fordítóra bízzuk. Ezt az akciót a *var* szóval kivitelezhetjük. Ez természetesen nem jelenti azt, hogy úgy használhatjuk a nyelvet, mint egy típustalan környezetet, abban a pillanatban, hogy értéket rendeltünk a változóhoz (ezt azonnal meg kell tennünk), az úgy fog viselkedni mint az ekvivalens típus. A ilyen változók típusa nem változtatható meg, de a megfelelő típuskonverziók végrehajthatóak.

```
int x = 10; // int típusú változó
var z = 10; // int típusú változó
z = "string"; // fordítási hiba
var w; //fordítási hiba
```

Néhány speciális esettől eltekintve a *var* használata nem ajánlott, mivel nehezen olvashatóvá teszi a forráskódot. A két leggyakoribb felhasználási területe a névtelen típusok és a lekérdezés-kifejezések.

5.2 Lokális változók

Egy blokkon belül deklarált változó lokális lesz a blokkra nézve, vagyis a program többi részéből nem látható (úgy is mondhatjuk, hogy a változó hatóköre a blokkra terjed ki). A fenti példában a *message* egy lokális változó, ha egy másik függvényből vagy osztályból próbálnánk meg elérni, akkor a program nem fordulna le.

5.3 Referencia- és értéktípusok

A .NET minden típusa a *System.Object* nevű típusból származik, és ezen belül szétoszlik érték- és referenciatípusokra. A kettő közti különbség leginkább a memóriában való elhelyezkedésben jelenik meg.
A CLR két helyre tud adatokat pakolni, az egyik a verem (*stack*) a másik a halom (*heap*). A verem egy ún. **LIFO** (last-in-first-out) adattár, vagyis az az elem amit utóljára berakunk az lesz a tetején, kivenni pedig csak a legfelső elemet tudjuk. A halom nem adatszerkezet, hanem a program által lefoglalt nyers memória, amit a CLR tetszés szerint használhat. Minden művelet a vermet használja, pl. ha össze akarunk adni két számot akkor a CLR lerakja mindkettőt a verembe és meghívja a

megfelelő utasítást ami kiveszi a verem legfelső két elemét összeadja őket, a végeredményt pedig visszateszi a verembe:

```
int x = 10;
int y = 11;
x + y
```

A verem:
|11**|**
|10**| --→összeadás művelet--→|**21**|**

Ez azt is jelenti egyben, hogy függetlenül attól, hogy értékről vagy referenciáról van szó, valamilyen módon mindkettőt be kell tudnunk rakni a verembe. Az értéktípusok teljes valójukban a veremben vannak, míg a referenciák a halomban jönnek létre és a verembe egy rájuk hivatkozó *referencia* kerül. De miért van ez így? Általában egy értéktípus csak egy-négy bytenyi helyet foglal el, ezért kényelmesen kezelhetjük a vermen keresztül. Ezzel szemben egy referenciatípus sokkal nagyobb szokott lenni és a memóriában való megjelenése is összetettebb, ezért hatékonyabb a halomban eltárolni. A forráskódban jól megkülönböztethető a kettő, míg referenciatípust a *new* operátor segítségével hozunk létre, addig egy értéktípusnál erre nincs feltétlenül szükség. Ez alól a szabály alól kivételt képez a *string* típus.

5.4 Boxing és unboxing

Boxing –nak (bedobozolás) azt a folyamatot nevezzük, amely megengedi egy értéktípusnak, hogy úgy viselkedjen, mint egy referenciatípus. Mivel minden típus (érték és referencia is) a *System.Object* típusból származik, ezért egy értéktípust értékül adhatunk egy *object* típusnak. Csakhogy az *object* maga is referenciatípus, ezért az értékadáskor létrejön a memóriában (a halomban, nem a veremben) egy referenciatípus karakterisztikájával rendelkező értéktípus. Ennek előnye, hogy olyan helyen is használhatunk értéktípust, ahol egyébként nem lehetne.
Vegyük a következő példát:

```
int x = 10;
Console.WriteLine("X erteke: {0}", x);
```

Elsőre semmi különös, de elárulom, hogy a *Console.WriteLine()* metódus ebben a formájában második paraméteréül egy *object* típusú változót vár. Vagyis ebben a pillanatban a CLR automatikusan bedobozolja az *x* változót.
A következő forráskód megmutatja, hogyan tudunk “kézzel” dobozolni:

```
int x = 10;
object boxObject = x; //bedobozolva

Console.WriteLine("X erteke: {0}", boxObject);
```

Most nem volt szükség a CLR –re.

Az unboxing (vagy kidobozolás) a boxing ellentéte, vagyis a bedobozolt értéktípusunkból kivarázsoljuk az eredeti értékét:

```
int x = 0;
object obj = x; //bedobozolva
int y = (int)obj; //kidobozolva
```

Az object típuson egy explicit típuskonverziót hajtottunk végre (erről hamarosan), így visszanyertük az eredeti értéket.

5.5 Konstansok

A *const* típusmódosító segítségével egy változót konstanssá tehetünk. A konstansoknak egyetlen egyszer adhatunk (és ekkor kell is adnunk) értéket, mégpedig a deklarációnál. Bármely későbbi próbálkozás fordítási hibát okoz.

```
const int x; //Hiba
const int x = 10; //Ez jó
x = 11; //Hiba
```

A konstans változóknak adott értéket/kifejezést fordítási időben ki kell tudnia értékelni a fordítónak.

```
Console.WriteLine("Adjon meg egy szamot: ");
int x = int.Parse(Console.ReadLine());
const y = x; //Ez nem jó, x nem ismert fordítási időben
```

5.6 A felsorolt típus

A felsorolt típus olyan adatszerkezet, amely meghatározott értékek névvel ellátott halmazát képviseli. Felsorolt típust az *enum* kulcsszó segítségével deklarálunk:

```
enum Animal { Cat, Dog, Tiger, Wolf };
```

Ezután így használhatjuk:

```
Animal a = Animal.Tiger;

if(a == Animal.Tiger) //Ha a egy tigris
{
    Console.WriteLine("a egy tigris...");
}
```

A felsorolás minden tagjának megfeleltethetünk egy egész értéket. Ha mást nem adunk meg, akkor az alapérelmezés szerint a számozás nullától kezdődik és deklaráció szerinti sorrendben (értsd: balról jobbra) eggyel növekszik.

```
enum Animal { Cat, Dog, Tiger, Wolf }

Animal a = Animal.Cat;
int x = (int)a; //x == 0
a = Animal.Wolf;
x = (int)a; //x == 3
```

Magunk is megadhatjuk az értékeket:

```
enum Animal { Cat = 1, Dog = 3, Tiger, Wolf }
```

Azok a nevek amelyekhez nem rendeltünk értéket explicit módon az őket megelőző név értékétől számítva kapják meg azt. Így a a fenti példában *Tiger* értéke négy lesz.

5.7 Null típusok

A referenciatípusok az inicializálás előtt nullértéket vesznek fel, illetve mi magunk is jelölhetjük őket "beállítatlannak":

```
class RefType { }

RefType rt = null;
```

Ugyanez az értéktípusoknál már nem működik:

```
int vt = null; //ez le sem fordul
```

Ez azért van, mert a referenciatípusok rengeteg plusz információt tartalmaznak, még az inicializálás előtt is, míg az értéktípusok memóriában elfoglalt helye a deklaráció pillanatában automatikusan feltöltődik nulla értékekkel.
Ahhoz, hogy meg tudjuk állapítani, hogy egy értéktípus még nem inicializált egy speciális típust a *nullable* típust kell használnunk, amit a "rendes" típus után írt kérdőjellel (*?*) jelzünk:

```
int? i = null; //ez már működik
```

Egy nullable típusra való konverzió implicit (külön kérés nélkül) megy végbe, míg az ellenkező irányba explicit konverzióra lesz szükségünk (vagyis ezt tudatnunk kell a fordítóval):

```
int y = 10;
int? x = y; //implicit konverzió
y = (int)x; //explicit konverzió
```

6. Operátorok

Amikor programozunk utasításokat adunk a számítógépnek. Ezek az utasítások kifejezésekből állnak, a kifejezések pedig operátorokból és operandusokból illetve ezek kombinációjából jönnek létre:

i = x + y;

Ebben a példában egy utasítást adunk, mégpedig azt, hogy i –nek értékéül adjuk x és y összegét. Két kifejezés is van az utasításban:

1. x + y –> ezt az értéket jelöljük * -al
2. i = * -> i –nek értékül adjuk a * -ot

Az első esetben x és y operandusok, a '+' jel pedig az összeadás művelet operátora. Ugyanígy a második pontban i és * (vagyis x+y) az operandusok az értékadás művelet ('=') pedig az operátor.
Egy operátornak nem csak két operandusa lehet. A C# nyelv egy- (unáris) és háromoperandusu (ternáris) operátorokkal is rendelkezik.
A következő néhány fejezetben átvesszünk néhány operátort, de nem az összeset. Ennek oka, hogy bizonyos operátorok önmagukban nem hordoznak jelentést, egy speciális részterület kapcsolódik hozzájuk, ezért ezeket az operátorokat majd a megfelelő helyen ismerjük meg. (Pl. az indexelő operátor most kimarad, elsőként a tömböknél találkozhat vele az olvasó.)

6.1 Operátor precedencia

Amikor több operátor is szerepel egy kifejezésben a fordítónak muszáj valamilyen sorrendet (precedenciát) fölállítani köztük, hiszen az eredmény ettől is függ. Pl.:

```
10 * 5 + 1
```

Sorrendtől függően az eredmény lehet *51*, vagy *60*. A jó megoldás az elöbbi, az operátorok végrehajtásának sorrendjében a szorzás és osztás előnyt élvez. A legelső helyen szerepelnek pl. a zárójeles kifejezések, utolsón pedig az értékadó operátor. Ha bizonytalanok vagyunk a végrehajtás sorrendjében mindig használjunk zárójeleket, ez a végleges programra semmilyen hatással sincs. A fenti kifejezés tehát így nézzen ki:

```
(10 * 5) + 1
```

6.2 Értékadó operátor

Az egyik legáltalánosabb művelet amit elvégezhetünk az az, hogy egy változónak értéket adunk. A C# nyelvben ezt az egyenlőségjel segítségével tehetjük meg:

```
int x = 10;
```

Létrehoztunk egy *int* típusú változót, elneveztük *x* –nek és kezdőértékének *10* –et adtunk. Természetesen nem kötelező a deklarációnál (amikor tájékoztatjuk a fordítót, hogy van egy valamilyen típusú, adott nevű változó) megadni a definíciót (amikor meghatározzuk, hogy a változó milyen értéket kapjon), ezt el lehet halasztani:

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

Ettől függetlenül a legtöbb esetben ajánlott akkor értéket adni egy változónak amikor deklaráljuk.
Egy változónak nem csak konstans értéket, de egy másik változót is értékül adhatunk, de csak abban az esetben, ha a két változó azonos típusú, illetve ha létezik a megfelelő konverzió (a típuskonverziókkal egy későbbi fejezet foglalkozik).

```
int x = 10;
int y = x; //y értéke most 10
```

6.3 Matematikai operátorok

A következő példában a matematikai operátorok használatát mutatjuk meg:

```
using System;

public class Operators
{
    static public void Main()
    {
        int x = 10;
        int y = 3;

        int z = x + y; //Összeadás: z = 10 + 3
        Console.WriteLine(z); //Kiírja az eredményt: 13
        z = x - y; //Kivonás: z = 10 - 3
        Console.WriteLine(z); // 7
        z = x * y; //Szorzás: z = 10 * 3
        Console.WriteLine(z); //30
        z = x / y; //Maradék nélküli osztás: z = 10 / 3;
        Console.WriteLine(z); // 3
        z = x % y; //Maradékos osztás: z = 10 % 3
        Console.WriteLine(z); // Az osztás maradékát írja ki: 1
        Console.ReadKey(); //Vár egy billentyűleütést
    }
}
```

6.4 Relációs operátorok

A relációs operátorok segítségével egy adott értékészlet elemei közti viszonyt tudjuk lekérdezni. A numerikus típusokon értelmezve van egy rendezés reláció:

```
using System;

public class RelOp
{
    static public void Main()
    {
        int x = 10;
        int y = 23;

        Console.WriteLine(x > y); //Kiírja az eredményt: false
        Console.WriteLine(x == y); //false
        Console.WriteLine(x != y); //x nem egyenlő y -al: true
        Console.WriteLine(x <= y); //x kisebb-egyenlő mint y: true
        Console.ReadKey();
    }
}
```

Az első sor egyértelmű, a másodikban az egyenlőséget vizsgáljuk a kettős egyenlőségjellel. Ilyen esetekben figyelni kell, mert egy elütés is nehezen kideríthető hibát okoz, amikor egyenlőség helyett az értékadó operátort használjuk. Az esetek többségében ugyanis így is le fog fordulni a program, működni viszont valószínűleg rosszul fog.
Minden ilyen operátor logikai típussal tér vissza. A relációs operátorok összefoglalása:

x > y	x nagyobb mint y
x >= y	x nagyobb vagy egyenlő mint y
x < y	x kisebb mint y
x <= y	x kisebb vagy egyenlő mint y
x == y	x egyenlő y -al
x != y	x nem egyenlő y -al

6.5 Logikai/feltételes operátorok

Akárcsak a C++, a C# sem rendelkezik „igazi” logikai típussal, ehelyett 1 és 0 jelzi az igaz és hamis értékeket:

```
using System;

public class RelOp
{
    static public void Main()
    {
        bool l = true;
        bool k = false;
        if(l == true && k == false)
        {
            Console.WriteLine("Igaz");
        }

        Console.ReadKey();
    }
}
```

Először felvettünk két logikai (*bool*) változót, az elsőnek „igaz” a másodiknak „hamis” értéket adtunk. Ezután egy elágazás következik, erről bővebben egy későbbi fejezetben lehet olvasni, a lényege az, hogy ha a feltétel igaz, akkor végrehajt egy bizonyos utasítás(oka)t. A fenti példában az „és” (**&&**) operátort használtuk, ez két operandust vár és akkor ad vissza „igaz” értéket, ha mindkét operandusa „igaz” vagy nullánál nagyobb értéket képvisel. Ebből következik az is, hogy akár az előző fejezetben megismert relációs operátorokból felépített kifejezések, vagy matematikai formulák is lehetnek operandusok. A program maga kiírja, hogy „Igaz”.
Nézzük az „és” igazságtáblázatát:

A	B	Eredmény
hamis	hamis	hamis
hamis	igaz	hamis
igaz	hamis	hamis
igaz	igaz	igaz

A második operátor a „vagy”:

```
using System;

public class RelOp
{
    static public void Main()
    {
        bool l = true;
        bool k = false;

        if(l == true || k == true)
        {
            Console.WriteLine("Igaz");
        }

        Console.ReadKey();
    }
}
```

A „vagy” (||) operátor akkor térít vissza „igaz” értéket, ha az operandusai közül valamelyik „igaz” vagy nagyobb mint nulla. Ez a program is ugyanazt csinálja, mint az előző, a különbség a feltételben van, „k” biztosan nem „igaz” (hiszen épp előtte kapott „hamis” értéket).

A „vagy” igazságtáblázata:

A	B	Eredmény
hamis	hamis	hamis
hamis	igaz	igaz
igaz	hamis	igaz
igaz	igaz	igaz

Az eredmény kiértékelése az ún. „lusta kiértékelés” (vagy „rövidzár”) módszerével történik, azaz a program csak addig vizsgálja a feltételt amíg muszáj. Pl. a „vagy”

példában a „k” soha nem fog kiértékelődni, mivel „l” van az első helyen (balról jobbra haladunk) és ő „igaz”, vagyis a feltétel „k” értékétől függetlenül mindenképpen teljesül.

A harmadik a „tagadás” (**!()**):

```
using System;

public class RelOp
{
    static public void Main()
    {
        int x = 10;

        if(!(x == 11))
        {
            Console.WriteLine("X nem egyenlo 11 -el!");
        }

        Console.ReadKey();
    }
}
```

Ennek az operátornak egy operandusa van, akkor ad vissza igaz értéket, ha az operandusban megfogalmazott feltétel hamis vagy egyenlő nullával. A „tagadás” (negáció) igazságtáblája:

A	Eredmény
hamis	igaz
igaz	hamis

Ez a három operátor ún. feltételes operátor, közölük az „és” és „vagy” operátoroknak létezik a „csonkolt” logikai párja is. A különbség annyi, hogy a logikai operátorok az eredménytől függetlenül kiértékelik a teljes kifejezést, nem élnek a „lusta” kiértékeléssel. A logikai „vagy” művelet:

```
        if(l == true | k == true)
        {
            Console.WriteLine("Igaz");
        }
```

A logikai „és”:

```
        if(l == true & k == true)
        {
            Console.WriteLine("Igaz");
        }
```

A logikai operátorok családjához tartozik (ha nem is szorosan) a feltételes operátor. Ez az egyetlen háromoperandusu operátor, a következőképpen működik:

feltétel ? ***igaz-ág*** : ***hamis-ág***;

```
using System;

public class RelOp
{
    static public void Main()
    {
        int x = 10;
        int y = 10;

        Console.WriteLine( (x == y) ? "Egyenlo" : "Nem egyenlo");

        Console.ReadKey();
    }
}
```

6.6 Bit operátorok

Az előző fejezetben említett logikai operátorok bitenkénti műveletek elvégzésére is alkalmasak numerikus típusokon.
A számítógép az adatokat kettes számrendszer –beli alakban tárolja, így pl. ha van egy *byte* típusú változónk (ami egy byte azaz 8 bit hosszú) aminek a „2” értéket adjuk, akkor az a következőképpen jelenik meg a memóriában:

2 → 00000010

A bit operátorok ezzel a formával dolgoznak.
Az eddig megismert kettő mellé még jön négy másik operátor is. A műveletek:

Bitenkénti „és”: veszi a két operandus bináris alakját és a megfelelő bitpárokon elvégzi az „és” műveletet azaz ha mindkét bit *1* állásban van akkor az adott helyen az eredményben is az lesz, egyébként *0*:

```
01101101
00010001 AND
00000001
```

Példa:

```
using System;
class Program
{
    static public void Main()
    {
        int x = 10;
        Console.WriteLine(x & 2);
        //1010 & 0010 = 0010 = 2
        Console.ReadKey();
    }
}
```

Bitenkénti „vagy”: hasonlóan működik mint az „és”, de a végeredményben egy bit értéke akkor lesz *1*, ha a két operandus adott bitje közül az egyik is az:

```
01101101
00010001 OR
01111101
```

```
using System;

class Program
{
    static public void Main()
    {
        int x = 10;
        Console.WriteLine(x | 2);
        //1010 | 0010 = 1010 = 10
        Console.ReadKey();
    }
}
```

Biteltolás balra: a kettes számrendszerbeli alak „felső” bitjét eltoljuk és a jobb oldalon keletkező üres bitet nullára állítjuk. Az operátor: *<<*:

```
 10001111 LEFT SHIFT
100011110
```

```
using System;

class Program
{
    static public void Main()
    {
        int x = 143;
        Console.WriteLine(x << 1);
        //10001111 (=143) << 1 = 100011110 = 286
        Console.ReadKey();
    }
}
```

Biteltolás jobbra: most az alsó bitet toljuk el és felül pótoljuk a hiányt. Az operátor: *>>*:

```
using System;

class Program
{
    static public void Main()
    {
        int x = 143;
        Console.WriteLine(x >> 1);
        Console.ReadKey();
    }
}
```

6.7 Rövid forma

Vegyük a következő példát:

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

Az *x* nevű változót megnöveltük tízzel. Csakhogy van egy kis baj: ez a megoldás nem túl hatékony. Mi történik valójában? Elsőként értelmezni kell a jobb oldalt, azaz ki kell értékelni *x* –et, hozzá kell adni tízet és eltárolni a veremben. Ezután ismét kiértékeljük *x* –et, ezúttal a bal oldalon.
Szerencsére van megoldás, mégpedig az ún. rövid forma. A fenti sorból ez lesz:

```
x += 10;
```

Rövidebb, szebb és hatékonyabb. Az összes aritmetikai operátornak létezik rövid formája.

A probléma ugyanaz, de a megoldás más a következő esetben:

```
x = x + 1;
```

Szemmel láthatóan ugyanaz a baj, azonban az eggyel való növelésre-csökkentésre van önálló operátorunk:

```
++x/--x;
x++/x--;
```

Ebből az operátorból rögtön két verziót is kapunk, prefixes (*++*/-- elől) és postfixes formát. A prefixes alak pontosan azt teszi amit elvárunk tőle, azaz megnöveli(csökkenti) az operandusát egyel.
A postfixes forma egy kicsit bonyolultabb, elsőként létrehoz egy átmeneti változót, amiben eltárolja az operandusa értékét, majd megnöveli eggyel az operandust, végül visszaadja az átmeneti változót. Ez elsőre talán nem tűnik hasznosnak, de vannak helyzetek amikor lényegesen megkönnyíti az életünket a használa.
Attól függően, hogy növeljük vagy csökkentjük az operandust inkrementális illetve dekrementáló operátorról beszélünk.

6.8 Egyéb operátorok

Unáris -/+: az adott szám pozitív illetve negatív értékét jelezzük vele. Csakis előjeles típusokon működik.

Typeof: az operandusa típusát adja vissza:

```
using System;

class Program
{
    static public void Main()
    {
        int x = 143;
        if(typeof(int) == x.GetType())
        {
            Console.WriteLine("X tipusa int");
        }
        Console.ReadKey();
    }
}
```

A változón meghívott *GetType()* metódus (amit mellesleg minden típus a *System.Object* –től örököl) a változó típusát adja vissza.

7. Vezérlési szerkezetek

Vezérlési szerkezetnek a program utasításainak sorrendiségét szabályozó konstrukciókat nevezzük.

7.1 Szekvencia

A legegyszerűbb vezérlési szerkezet a szekvencia. Ez tulajdonképpen egymás után megszabott sorrendben végrehajtott utasításokból áll.

7.2 Elágazás

Gyakran előfordul, hogy meg kell vizsgálnunk egy állítást, és attól függően, hogy igaz vagy hamis más-más utasítást kell végrehajtanunk. Ilyen esetekben elágazást használunk:

```
using System;

class Conditional
{
    static public void Main()
    {
        int x = 10;

        if(x == 10) //Ha x == 10
        {
            Console.WriteLine("X == 10");
        }
        Console.ReadKey();
    }
}
```

Mi van akkor, ha azt is jelezni akarjuk, hogy „x” nem egyenlő tízzel? Erre való az „else” ág, ez akkor hajtódik végre, ha a program nem talál más feltételt. Ezt az ágat nem kötelező megadni:

```
using System;
class Conditional
{
    static public void Main()
    {
        int x = 10;
        if(x == 10) //Ha x == 10
        {
            Console.WriteLine("X == 10");
        }
        else //De ha nem annyi
        {
            Console.WriteLine("X nem egyenlo tizzel!");
        }
        Console.ReadKey();
    }
}
```

Arra is van lehetőségünk, hogy több feltételt is megvizsgáljunk, ekkor „else-if” –et használunk:

```
using System;

class Conditional
{
    static public void Main()
    {
        int x = 12;

        if(x == 10) //Ha x == 10
        {
            Console.WriteLine("X == 10");
        }
        else if(x == 12) //Vagy x == 12
        {
            Console.WriteLine("X == 12");
        }
        else //De ha nem annyi
        {
            Console.WriteLine("X nem egyenlo tizzel és tizenkettovel sem!");
        }
        Console.ReadKey();
    }
}
```

A program az első olyan ágat fogja végrehajtani aminek a feltétele teljesül (termszetesen előfordulhat, hogy egyik sem lesz jó, ekkor nem történik semmi).
Egy elágazásban pontosan egy darab *if* bármennyi *else-if* és pontosan egy *else* ág lehet. Egy elágazáson belül is írhatunk elágazást.

Az utolsó példában egy darab változó értékét vizsgáljuk. Ilyen esetekben azonban van egy egyszerűbb és elegánsabb megoldás, mégpedig a „switch-case” szerkezet. Ezt akkor használjuk, ha egy változó több lehetséges állapotát akarjuk vizsgálni:

```
using System;

class Switch
{
    static public void Main()
    {
        int x = 10;

        switch(x)
        {
            case 10:
                Console.WriteLine("X == 10");
                break;
            default:
                Console.WriteLine("Default");
                break;
        }
```

```
            Console.ReadKey();
        }
}
```

A *switch* –en belül elsőként megvizsgáljuk, hogy *x* egyenlő –e tízzel. Ha igen, kiírjuk és a *break* –kel kiugrunk a *switch* –ből (egyébként a *break* minden ilyen esetben alkalmazható, pl. megszakítjuk egy elágazás egy ágának végrehajtását).
Ha viszont *x* nem egyenlő tízzel, akkor a *default* –ra ugrunk, ami gyakorlatilag megfelel egy *else* ágnak.
Az egyes esetek utasításai után meg kell adni, hogy hol folytassa a program a végrehajtást. Ezt vagy a *break* –kel tesszük a már látott módon, vagy valamilyen „ugró” utasítással (*jump*, *return*, *goto*, stb…):

```
using System;

class Program
{
    static public void Main()
    {
        Console.WriteLine("Adjon meg egy szamot: ");
        int x = int.Parse(Console.ReadLine());

        switch(x)
        {
            case 12:
                Console.WriteLine("X == 12");
                break;
            case 10:
                goto default;
            default:
                Console.WriteLine("Default");
                break;
        }

        Console.ReadKey();
    }
}
```

Bekérünk egy számot a felhasználótól, és eltároljuk egy változóban. Ezt a *Console.ReadLine(…)* függvényen meghívott *int.Parse(…)* metódussal tesszük, előbbi beolvassa a standard inputot az első sorvége jelig, utóbbi pedig numerikussá konvertálja az eredményt (ha tudja, ha pedig nem… nos, erről egy későbbi fejezetben lesz szó).
Most jön a *switch*. Az első esetet már láttuk, a másodikban pedig a *default* címkére ugrunk a *goto* utasítással.

7.3 Ciklus

Amikor egy adott utasítássorozatot egymás után többször kell végrehajtanunk, akkor cilust használunk. A C# négyféle ciklust biztosít számunkra.
Az első az ún. számlálós ciklus (nevezzük for-ciklusnak):

```
using System;

class Program
{
    static public void Main()
    {
        for(int i = 0;i < 10;++i)
        {
            Console.WriteLine("X == {0}", i);
        }

        Console.ReadKey();
    }
}
```

Mielőtt kivesézzük a ciklust nézzük meg a ciklusmagot:

```
Console.WriteLine("I == {0}", i);
```

Ismerkedjünk meg az ún. formátumstringgel. A ciklusunk kiírja a számokat nullától kilencig, de szeretnénk egy kicsit kicsinosítani, ezért a szám elé írjuk, hogy „*I == *„. Ehhez a szöveghez akarjuk hozzáadni a változó értékét és ezt is tesszük a *{0}* jelöléssel. Ez a formátumstring utáni első (illetve nulladik, hiszen onnan kezdjük a számozást) paramétert helyettesíti be a szövegbe.

Most nézzük a ciklusfejet:

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

Elsőként létrehozunk egy ciklusváltozót (*i*) amivel nyilvántartjuk, hogy éppen hol tart a ciklus. Mivel ez egy számlálós ciklus ezért *i* valamikor eléri (jó esetben) azt az értéket, ahol a ciklus befejeződik. Ezt a végpontot (alias ciklusfeltétel) adjuk meg a második helyen, tehát addig megy a ciklus amíg *i* kisebb tíznél. Végül gondoskodnunk kell arról, hogy növeljük(csökkentsük, mikor mi kell) a ciklusváltozót, ezt a már ismert inkrementális operátorral tesszük, mégpedig a prefix formával mivel nincs szükségünk az átmeneti változóra, csak a memóriát foglalná. Természetesen itt bármilyen más kifejezést is használhatunk, ami hatással van a ciklusváltozó értékére (pl. nem eggyel, hanem kettővel növeljük az értékét).

Második kliensünk az előltesztelős ciklus (mostantól hívjuk *while*-ciklusnak), ami onnan kapta a nevét, hogy a ciklusmag végrehajtása előtt ellenőrzi a ciklusfeltételt ezért előfordulhat az is, hogy a ciklus egyszer sem fut le:

```
using System;

class Program
{
    static public void Main()
    {
        int i = 0; //kezdőérték

        while(i < 10) //ciklusfeltétel
```

```
        {
            Console.WriteLine("I == {0}", i);
            ++i; //ciklusváltozó növelése
        }

        Console.ReadKey();
    }
}
```

A program ugyanazt csinálja mint az előző, viszont itt jól láthatóan elkülönülnek a ciklusfeltételért felelős utasítások (kezdőérték, ciklusfeltétel, növel/csökkent).

A harmadik versenyző következik, őt hátultesztelős ciklusnak hívják (legyen *do-while*), nem nehéz kitalálni, hogy azért kapta ezt a nevet mert a ciklusmag végrehajtása után ellenőrzi a feltételt, így legalább egyszer biztosan lefut:

```
using System;

class Program
{
    static public void Main()
    {
        int i = 0;
        do
        {
            Console.WriteLine("I == {0}", i);
            ++i;
        } while(i < 10);

        Console.ReadKey();
    }
}
```

Végül, de nem utolsósorban a foreach (neki nincs külön neve) ciklus következik. Ezzel a ciklussal végigiterálhatunk egy tömbön vagy gyűjteményen, illetve minden olyan objektumon, ami megvalósítja az *IEnumerable* és *IEnumerator* interfészeket (interfészekről egy későbbi fejezet fog beszámolni, ott lesz szó erről a kettőről is).
A példánk most nem a már megszokott „számoljunk el kilencig” lesz, helyette végigmegyünk egy *string* –en:

```
using System;

class Program
{
    static public void Main()
    {
        string str = "abcdefghijklmnopqrstuvwxyz";

        foreach(char ch in str)
        {
            Console.Write(ch);
        }

        Console.ReadKey();
```

```
    }
}
```

A cilusfejben felveszünk egy *char* típusú változót (egy string karakterekből áll), utána az *in* kulcsszó következik, amivel kijelöljük, hogy min megyünk át.
A foreach pontos működésével az interfészekről szóló fejezet foglalkozik, többek között megvalósítunk egy osztályt, amelyen a *foreach* képes végigiterálni (azaz megvalósítjuk a fentebb említett két interfészt).

7.3.1 Yield

A *yield* kifejezés lehetővé teszi, hogy egy metódust használjunk (metódusokról egy későbbi fejezet szól) foreach ciklussal:

```
using System;
using System.Collections;

class Program
{
    static public IEnumerable EnumerableMethod(int max)
    {
        for(int i = 0;i < max;++i)
        {
            yield return i;
        }
    }

    static public void Main()
    {
        foreach(int x in EnumerableMethod(10))
        {
            Console.WriteLine(x);
        }

        Console.ReadKey();
    }
}
```

7.4 Gyakorló feladatok

1. Készítsünk számkitaláló játékot, amely lehetőséget ad kiválasztani, hogy a felhasználó próbálja kitalálni a program által „sorsolt” számot, vagy fordítva. A kitalált szám legyen pl. 1 és 100 között. Öt próbálkozása lehet a játékosnak, minden tipp után írjuk ki, hogy a tippelt szám nagyobb, vagy kisebb mint a kitalált szám.
Ha a gépen van a sor, akkor használjuk a következő fejezet elején bevezetett véletlenszámgeneráló objektumot (*Random*), a szám létrehozására. A gépi játékos úgy találja ki a számot, hogy mindig felezi az intervallumot (pl.először 50 –t tippel, ha a kitalált szám nagyobb, akkor 75 jön, és így tovább).
A felhasználó egy játék végén rögtönk kezdhessen újabbat.

2. Készítsünk egy egyszerű számológépet! A program indításakor jelenjen meg egy menü, ahol kiválaszthatjuk a műveletet, ezután bekérjük a művelet tagjait (az egyszerűség kedvéért legyen csak két szám).

3. Készítsünk kő-papír-olló játékot! Ebben az esetben is használjuk a véletlenszámgenerátort. A játék folyamatos legyen, vagyis addig tart, amíg a felhasználó kilép (nehezítésképpen lehet egy karakter, amelyre a játék végetér). Tartsuk nyílván a játék állását és minden forduló után írjuk ki.

8. Tömbök

Gyakran van szükségünk arra, hogy több azonos típusú objektumot tároljunk el. Ilyenkor kényelmetlen lenne mindegyiknek változót foglalnunk (képzeljünk el 30 darab *int* típusú értéket, még leírni is egy örökkévalóság lenne), de ezt nem kell megtennünk, hiszen rendelkezésünkre áll a tömb adatszerkezet.
A tömb meghatározott számú, azonos típusú elemek halmaza. Minden elemre egyértelműen mutat egy index (egész szám). A C# mindig folytonos memóriablokkokban helyezi el egy tömb elemeit.
Tömbdeklaráció:

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

Létrehoztunk egy 10 darab *int* típusú elemeket tartalmazó tömböt. A tömb deklarációja után az egyes indexeken lévő elemek automatikusan a megfelelő nullértékre inicializálódnak (ebben az esetben 10 darab nullát fog tartalmazni a tömbünk). Ez a szabály referenciatípusoknál (osztályok) kissé máshogy működik. Ekkor a tömbelemek *null* -ra inicializálódnak. Ez nagy különbség, mivel értéktípusok esetében szimpla nullát kapnánk vissza az általunk nem beállított indexre hivatkozva, míg referenciatípusoknál ugyanez a művelet *NullReferenceException* kivételt fog generálni.
Az egyes elemekre az indexelő operátorral (szögeletes zárójel -> *[ ]*) és az elem indexével(sorszámával) hivatkozunk. A számozás mindig nullától kezdődik, így a legutolsó elem indexe az elemek száma mínusz egy. A következő példában feltöltünk egy tömböt véletlen számokkal és kiíratjuk a tartalmát számlálós és foreach (ugyanis minden tömb implementálja az *IEnumerator* és *IEnumerable* interfészeket) ciklussal:

```
using System;

class Program
{
    static public void Main()
    {
        int[] array = new int[10];
        Random r = new Random();

        for(int i = 0;i < array.Length;++i)
        {
            array[i] = r.Next();
        }

        for(int i = 0;i < array.Length;++i)
        {
            Console.WriteLine("{0}, ", array[i]);
        }
        Console.WriteLine();

        foreach(int i in array)
        {
            Console.WriteLine("{0}, ", i);
        }
        Console.ReadKey();
```

```
    }
}
```

A példánkban egy véletlenszámgenerátor objektumot (*Random r*) használtunk, ami a nevéhez híven véletlenszámokat tud visszaadni, az objektumon meghívott *Next()* metódussal. A példaprogram valószínűleg elég nagy számokat fog produkálni, ezt a *Next()* metódus paramétereként megadott felső határral redukálhatjuk:

```
array[i] = r.Next(100);
```

Ekkor maximum 99, minimum 0 lehet a véletlenszám.
Egy dolog van még ami ismeretlen, a tömb *Length* tulajdonsága, ez a tömb elemeinek számát adja vissza. Tulajdonságokkal egy későbbi fejezet foglalkozik részletesen.
Látható az indexelőoperátor használata is, az *array[i]* az *array* i –edik elemét jelenti. Az indexeléssel vigyázni kell, ugyanis a fordító nem ellenőrzi fordítási időben az indexek helyességét, viszont helytelen indexelés esetén futás időben *System.IndexOutOfRangeException* kivételt fog dobni a program. Kivételkezelésről is egy későbbi fejezet szól.

Egy tömböt akár a deklaráció pillanatában is feltölthetünk a nekünk megfelelő értékekkel:

```
char[] chararray = new char[] { 'b', 'd', 'a', 'c' };
```

Ekkor az elemek száma a megadott értékektől függ. Egy tömb elemeinek száma a deklarációval meghatározott, azon a későbbiekben nem lehet változtatni. Dinamikusan bővíthető adatszerkezetekről a **Gyűjtemények** című fejezet szól.

Minden tömb a *System.Array* osztályból származik, ezért néhány hasznos művelet azonnal rendelkezésünkre áll (pl. rendezhetünk egy tömböt a *Sort()* metódussal):

```
chararray.Sort(); //tömb rendezése
```

8.1 Többdimenziós tömbök

Eddig az ún. egydimenziós tömböt, vagy vektort használtuk. Lehetőségünk van azonban többdimenziós tömbök létrehozására is, ekkor nem egy indexxel hivatkozunk egy elemre hanem annyival ahány dimenziós. Vegyük például a matematikából már ismert mátrixot:

$$A = \begin{bmatrix} 12, 23, 2 \\ 13, 67, 52 \\ 45, 55, 1 \end{bmatrix}$$

Ez egy kétdimenziós tömbnek (mátrix a neve – ki hinné?) felel meg, az egyes elemekre két indexxel hivatkozunk, első helyen a sor áll utána az oszlop. Így a 45 indexe: [3, 0] (ne feledjük, még mindig nullától indexelünk).
Multidimenziós tömböt a következő módon hozunk létre C# nyelven:

```
int[,] matrix = new int[3, 3];
```

Ez itt egy 3x3 –as mátrix, olyan mint a fent látható. Itt is összeköthetjük az elemek megadását a deklarációval, bár egy kicsit trükkösebb a dolog:

```
int[,] matrix = new int[,]
    {
        {12, 23, 2},
        {13, 67, 52},
        {45, 55, 1}
    };
```

Ez a mátrix már pontosan olyan mint amit már láttunk. Az elemszám most is meghatározott, nem változtatható.
Nyílván nem akarjuk mindig kézzel feltölteni a tömböket, a már ismert véletlenszámgenerátoros példa módosított változata jön:

```
using System;

class Program
{
    static public void Main()
    {
        int[,] matrix = new int[3,3];
        Random r = new Random();

        for(int i = 0;i < matrix.GetLength(0);++i)
        {
            for(int j = 0;j < matrix.GetLength(1);++j)
            {
                matrix[i, j] = r.Next();
            }
        }
        Console.ReadKey();
    }
}
```

Most nem írjuk ki a számokat, ezt nem okozhat gondot megírni. A tömbök *GetLength()* metódusa a paraméterként megadott dimenzió hosszát adja vissza (nullától számozva), tehát a példában az első esetben a sor, a másodikban az oszlop hosszát adjuk meg a ciklusfeltételben.

A többdimenziós tömbök egy variánsa az ún. egyenetlen(jagged) tömb. Ekkor legalább egy dimenzió hosszát meg kell adnunk, ez konstans marad, viszont a belső tömbök hossza tetszés szerint megadható:

```
int[][] jarray = new int[3][];
```

Készítettünk egy három sorral rendelkező tömböt, azonban a sorok hosszát (az egyes sorok nyílván önálló vektorok) ráérünk később megadni és nem kell ugyanolyannak lenniük:

```
using System;

class Program
{
    static public void Main()
    {
        int[][] jarray = new int[3][];
        Random r = new Random();

        for(int i = 0;i < jarray.Length;++i)
        {
            jarray[i] = new int[r.next(1, 5)];
            for(int j = 0;j < jarray[i].Length;++j)
            {
                jarray[i][j] = r.next(100);
            }
        }

         Console.ReadKey();
    }
}
```

Véletlenszámgenerátorral adjuk meg a belső tömbök hosszát, persze értelmes kereteken belül. A belső ciklusban jól látható, hogy a tömb elemei valóban tömbök, hiszen használtuk a *Length* tulajdonságot.

Az inicializálás a következőképpen alakul ebben az esetben:

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

A C# 3.0 megengedi a *var* kulcsszó használatát is, az előbbi példa így néz ki ekkor:

```
var jarray = new int[][]
    {
        new int[] {1, 2},
        new int[] {3, 4}
    };
```

A fordító fordításkor dönti el a pontos típust, azon nem lehet változtatni.

8.2 ArrayList

*A fejezet olvasása előtt ajánlott elolvasni az **Öröklődés** és **Típuskonverziók** című fejezetet.*

Az *ArrayList* két problémára, a fix méretre és a meghatározott típusra nyújt megoldást. Azt már tudjuk, hogy minden osztály a *System.Object* –ből származik, így

ha van egy olyan tárolónk, amely ilyentípusú objektumokat tárol, akkor nyert ügyünk van. Hogy ez miért van így, arról az **Öröklődés** című fejezet nyújt információt.
Az *ArrayList* hasonlóan működik mint egy tömb, két különbség van, az egyik, hogy nem adunk meg méretet a deklarációnál, a másik pedig, hogy típuskonverziót kell alkalmaznunk, amikor hivatkozunk egy elemére.
Mielőtt használnánk egy *ArrayList* –et, a programban hivatkoznunk kell a *System.Collections* névtérre.

```
using System;
using System.Collections;

class Program
{
    static public void Main()
    {
        ArrayList list = new ArrayList();
        Random r = new Random();

        for(int i = 0;i < 10;++i)
        {
            list.Add(i);
        }

        for(int i = 0;i < list.Count;++i)
        {
            Console.WriteLine("{0}, ", list[i]);
        }

        Console.ReadKey();
    }
}
```

Az *Add()* metódus segítségével tudunk elemeket betenni a listába. Bár azt mondtam, hogy a kivételnél konverzió szükséges, most nem csináltunk ilyet. Ez azért van így, mert a *Console.WriteLine* –nak van olyan túlterhelt változata, amely egy *System.Object* típusú elemet vár. Egész más a helyzet, ha konkrét típusú változónak akarom értékül adni az adott indexen lévő elemet:

```
int x = list[4];
```

Feltételezzük, hogy *int* típusú elemeket tároltunk el, és van elem a negyedik indexen. A fenti sor már a fordításon fennakad a „Cannot implicitly convert type object to int” kezdetű hibaüzenettel . Egy *Object* típusú elemet ugyanis nem készítettek fel implicit konvertálásra, ez a programozó dolga. A fenti sor így néz ki helyesen:

```
int x = (int)list[4];
```

Ez egy ún. explicit típuskonverzió, bővebben lásd a **Típuskonverziók** című fejezetet. Az ArrayList a C# 1.0 –ban jelent meg, a 2.0 bevezette a típusos változtait a generikus adatszerkezetekkel.

Egy *ArrayList* deklarációjánál nem adjuk meg, hány elemet fogunk eltárolni. Egy normális tömb kivételt dob, ha több elemet próbálunk elhelyezni benne, mint

amennyit megadtunk. Akkor honnan tudja az *ArrayList*, hogy mennyi helyre van szüksége? A válasz egyszerű, van neki egy alapértelmezett kapacitása (ezt a *Capacity* tulajdonsággal kérdezhetjük le). Ez a kezdet kezdetén 16, de ez nem mindig elég, így mielőtt elhelyeznénk a 17 –ik elemet, a kapcitás 32 –re ugrik (és így tovább). A kapacitást kézzel is beállíthatjuk a már említett tulajdonság segítségével, ekkor figyeljünk arra, hogy az aktuális elemek számánál nem kisebb értéket adjunk meg, egyébként *ArgumentOutOfRange* kivétel dobódik.

Az *ArrayList* –en kívűl számos adatszerkezetet is szolgáltat nekünk a C#, de helyettük érdemesebb a nekik megfelelő generikus változatokat használni, ezekről bővebben a generikus gyűjteményekről szóló fejezetben olvashatunk.

8.3 Gyakorló feladatok

1. Töltsünk fel egy tömböt véletlenszámokkal és válasszuk ki közülük a legnagyobbat (az mindegy, ha több egyenlő legnagyobb szám van).

2. Töltsünk fel egy tömböt véletlenszámokkal és döntsük el, hogy hány darab páros szám van benne.

9. Stringek

A C# beépített karaktertípusa (*char*) egy Unicode karaktert képes eltárolni két byte –on. A szintén beépített *string* típus ilyen karakterekből áll (tehát *char* –ként hivatkozhatunk az egyes betűire).

```
using System;

class Program
{
    static public void Main()
    {
        string s = "ezegystring";
        Console.WriteLine(s);
        Console.ReadKey();
    }
}
```

Nem úgy tűnik, de a string referenciatípus, ennek ellenére nem kell használnunk a *new* operátort.
Egy string egyes betűire az indexelő operátorral hivatkozhatunk:

```
using System;

class Program
{
    static public void Main()
    {
        string s = "ezegystring";
        Console.WriteLine(s[0]); //e
        Console.ReadKey();
    }
}
```

Ekkor a visszaadott típus *char* lesz. A foreach ciklussal indexelő operátor nélkül is végigiterálhatunk a karaktersorozaton:

```
foreach(char ch in s)
{
    Console.WriteLine(ch);
}
```

Az indexelő operátort nem csak változókon, de „nyers” szövegen is alkalmazhatjuk:

```
Console.WriteLine("ezegystring"[0]); //e
```

9.1 Metódusok

A C# számos hasznos metódust biztosít a stringek hatékony kezeléséhez. Most bemutatunk néhányat, de emlékezzünk arra, hogy a metódusoknak számos változata lehet, itt a leggyakrabban használtakat mutajuk be:

Összehasonlítás:

```
string s = "other";
string c = "another";

int x = String.Compare(s, c)
if(x == 0)
{
     Console.WriteLine("A ket string egyenlo...");
}
else if(x > 0 || x < 0)
{
     Console.WriteLine("A ket string nem egyenlo...");
}

x = String.CompareOrdinal(s, c);
x = s.CompareTo(c);
```

Mindhárom metódus nullát ad vissza, ha a két string egyenlő és nullánál kisebbet/nagyobbat, ha nem (pontosabban ha lexikografikusan kisebb/nagyobb). Az első változat nagyság szerint sorbarendezi a karaktereket és úgyhasonlítja össze a két karaktersorozatot, a második az azonos indexen lévő karaktereket nézi, míg a harmadik az elsőhöz hasonló, de ez nem statikus hanem példánymetódus.

Keresés:

```
using System;
class Program
{
     static public void Main()
     {
          string s = "verylonglongstring";
          char[] chs = new char[]{ 'y', 'z', '0' };

          Console.WriteLine(s.IndexOf('r')); //2
          Console.WriteLine(s.IndexOfAny(chs)); //3
          Console.WriteLine(s.LastIndexOf('n')); //16
          Console.WriteLine(s.LastIndexOfAny(chs)); //3
          Console.WriteLine(s.Contains("long")); //true

          Console.ReadKey();
     }
}
```

Az *IndexOf()* és *LastIndexOf()* metódusok egy string vagy karakter első illetve utolsó előfordulási indexét (stringek esetén a kezdés helyét) adják vissza. Ha nincs találat, akkor a visszaadott érték -1 lesz. A két metódus *Any* –re végződő változata egy karaktertömböt fogad paramétereként. A *Contains()* metódus igaz értékkel tér vissza, ha a paramétereként megadott karakter(sorozat) benne van a stringben.

Módosítás:

```
using System;

class Program
{
    static public void Main()
    {
        string s = "smallstring";
        char[] chs = new char[]{ 's', 'g' };

        Console.WriteLine(s.Replace('s', 'l')); //lmalllring
        Console.WriteLine(s.Trim(chs)); //mallstrin
        Console.WriteLine(s.Insert(0, "one")); //onesmallstring
        Console.WriteLine(s.Remove(0, 2)); //allstring
        Console.WriteLine(s.Substring(0, 3)); //sma
        string c = s.ToUpper();
        Console.WriteLine(s); //SMALLSTRING
        Console.WriteLine(c.ToLower()); //smallstring

        Console.ReadKey();
    }
}
```

A *Replace()* metódus az első paraméterének megfelelő karaktert lecseréli a második paraméterre. A *Trim()* metódus a string elején és végén lévő karaktereket vágja le, a *Substring()* kivág egy karaktersorozatot, paraméterei a kezdő és végindexek (van egyparaméteres változata is, ekkor a csak a kezdőindexet adjuk meg és a végéig megy). Az *Insert()/Remove()* metódusok hozzáadnak illetve elvesznek a stringből. Végül a *ToLower()* és *ToUpper()* metódusok pedig kis- illetve nagybetűssé alakítják az eredeti stringet.
Fontos megjegyezni, hogy ezek a metódusok soha nem az eredeti stringen végzik a módosításokat, hanem egy új példányt hoznak létre és azt adják vissza.

9.2 StringBuilder

Amikor módosítunk egy stringet akkor automatikusan egy új példány jön létre a memóriában, hiszen az eredeti változat nem biztos, hogy ugyanakkora mint az új.
Ha sokszor van szükségünk erre, akkor használjuk inkább a *StringBuilder* típust, ez automatikusan lefoglal egy nagyobb darab memóriát és ha ez sem elég, akkor allokál egy nagyobb területet és átmásolja magát oda. A *StringBuilder* a *System.Text* névtérben található.

```
using System;
using System.Text;

class Program
{
    static public void Main()
    {
        StringBuilder builder = new StringBuilder(50);
```

```
            for(char ch = 'a';ch <= 'z';++ch)
            {
                  builder.Append(ch);
            }

            Console.WriteLine(builder);
            Console.ReadKey();
      }
}
```

A *StringBuilder* fenti konstruktora (van több is) helyet foglal ötven karakter számára (létezik alapértelmezett konstruktora is, ekkor az alapértelmezett tizenhat karakternek foglal helyet). Az *Append()* metódussal tudunk karaktereket (vagy egész stringeket) hozzáfűzni.

A for ciklust érdekesen használtuk, most nem numerikus típuson iteráltunk, hanem karaktereken. Azt, hogy ez miért működik a következő fejezetből tudhatjuk meg.

10. Típuskonverziók

Azt már tudjuk, hogy az egyes típusok másként jelennek meg a memóriában. Azonban gyakran kerülünk olyan helyzetbe, hogy egy adott típusnak úgy kellene viselkednie, mint egy másiknak. Ilyen helyzetekben típuskonverziót (vagy castolást) kell elvégeznünk. Kétféleképpen konvertálhatunk: implicit és explicit módon. Az előbbi esetben nem kell semmit tennünk, a fordító elvégzi helyettünk. Implicit konverzió általában „hasonló" típusokon mehet végbe, szinte minden esetben a szűkebb típusról a tágabbra:

```
int x = 10;
long y = x; //y == 10, implicit konverzió
```

Ebben az esetben a *long* és *int* mindketten egész numerikus típusok, és a *long* a tágabb, ezért a konverzió gond nélkül végbemegy. Egy implicit konverzió minden esetben sikeres és nem jár adatvesztéssel.

Egy explicit konverzió nem feltétlenül fog működni és adatvesztés is felléphet. Vegyük a következő példát:

```
int x = 300;
byte y = (byte)x; //explicit konverzió
```

A *byte* szűkebb típus mint az *int*, ezért explicit konverzitó hajtottunk végre, ezt a változó előtti zárójelbe írt típussal jelöltük. Ha lehetséges a konverzió, akkor végbemegy, egyébként a fordító figyelmeztetni fog. Vajon mennyi most az *y* változó értéke? A válasz elsőre meglepő lehet: ***44***. A magyarázat: a 300 egy kilenc biten felírható szám (*100101100*), persze az *int* kapacitása ennél nagyobb, de a többi része nulla lesz. A *byte* viszont (ahogy a nevében is benne van) egy nyolcbites értéket tárolhat, ezért a 300 –nak csak az első 8 bitje fog értékül adódni az *y* –nak, ami pont 44.

10.1 Ellenőrzött konverziók

A programfejlesztés alatt hasznos lehet tudnunk, hogy minden konverzió gond nélkül végbement –e. Ennek ellenőrzésére ún. ellenőrzött konverziót fogunk használni, amely kivételt dob (erről hamarosan), ha a forrás nem fér el a célváltozóban:

```
checked
{
    int x = 300;
    byte y = (byte)x;
}
```

Ez a kifejezés kivételt fog dobni. Figyeljünk arra, hogy ilyen esetekben csak a blokkon belül deklarált, statikus és tag változókat vizsgálhatjuk.
Előfordul, hogy csak egy-egy konverziót szeretnénk vizsgálni, amihez nincs szükség egy egész blokkra:

```
int x = 300;
byte y = checked((byte)x);
```

Az ellenőrzés kikapcsolását is megtehetjük az *unchecked* használatával:

```
int x = 300;
byte y = unchecked((byte)x);
```

Az ajánlás szerint ellenőrzött konverziókat csak a fejlesztés ideje alatt használjunk, mivel némi teljesítményvesztéssel jár.

10.2 Is és as

Az *is* operátort típusok futásidejű lekérdezésére használjuk:

```
int x = 10;

if(x is int) //ha x egy int
{
    Console.WriteLine("X tipusa int");
}
```

Ennek az operátornak a legnagyobb haszna az, hogy le tudjuk kérdezni, hogy egy adott osztály megvalósít –e egy bizonyos interfészt (erről később).

Párja az *as* az ellenőrzés mellett egy explicit típuskonverziót is végrehajt:

```
int x = 10;
byte y = x as y;
```

Amennyiben ez a konverzió nem hajtható végre a célváltozóhoz a megfelelő nulla érték rendelődik (értéktípusokhoz *0*, referenciatípusokhoz *null*).

10.3 Karakterkonverziók

A *char* típust implicit módon tudjuk numerikus típusra konvertálni, ekkor a karakter Unicode értékét kapjuk vissza:

```
using System;

class Program
{
    static public void Main()
    {
        for(char ch = 'a';ch <= 'z';++ch)
        {
            int x = ch;
            Console.WriteLine(x);
        }
        Console.ReadKey();
    }
}
```

Erre a kimenet a következő lesz:

```
97
98
99
100
…
```

A kis ’a’ betű hexadecimális Unicode száma *0061h*, ami a 97 decimális számnak felel meg, tehát a konverzió a tízes számrendszerbeli értéket adja vissza.

11. Gyakorló feladatok I.

Most néhány gyakorló feladat következik. Itt nem szerepel teljes megoldás, viszont a megoldást segítő tippek, tanácsok igen. Jó szórakozást!

11.1 Másodfokú egyenlet

Készítsünk programot, amely bekéri az egyenlet három együtthatóját és kiszámolja a gyökeit(gyökét), illetve jelzi, ha nem lehet kiszámolni az eredményt (a == 0, a diszkrimináns negatív, stb…).

Tipp: nyílván megvizsgáljuk, hogy a diszkrimináns egyenlő –e nullával. Azonban ezt az értéket valamilyen lebegőpontos (pl. *double*) típussal érdemes reprezentálni. Ekkor viszont az ellenőrzésnél gond adódhat. Nézzük a következő esetet:

```
double y = 0;
if(y == 0)
{
    //itt csinálunk valamit
}
```

A meglepő az, hogy a feltétel soha nem lesz igaz. Hogy miért? Nos, az egyenlőség csak akkor igaz, ha a két operandusa bitenként megegyezik, de mivel az első egy *double* ez nem valószínű (a meggondolás egyéni feladat, de gondoljunk a típus nagyságára). Szerencsére van beépített megoldás, a *Double.Epsilon* statikus tulajdonság értéke pont egy *double* típus által felvehető legkisebb pozitív szám:

```
double y = 0;
if(y == Double.Epsilon)
{
    //itt csinálunk valamit
}
```

Ez már biztosan működni fog.

11.2 Fibonacci számok

Készítsünk programot amely kiírja az első *n* tagját a Fibonacci számoknak. Az *n* változót a felhasználótól kérjük be.

Tipp: a Fibonacci sor tagjainak értéke a megelőző két tag összege. Az első két elemet így nyílván meg kell majd adnunk, ez a 0 és az 1 legyen.

11.3 Felderítés

Készítsünk programot, amely egy felderítőrepülőgépet szimulál. A feladata, hogy megkeresse egy adott területen a legmagasabb szárazföldi- és a legmélyebben fekvő vízalatti pont magasságát/mélységét. A mérési adatok egy tömbben legyenek eltárolva, pozitív számok jelezzék a szárazföldet, negatívak a vízet. Nehezítésként használhatunk kétdimenziós tömböt.

Tipp: véletlenszámgenerátorral töltsük fel a tömböt, negatív számot a következőképpen tudunk sorsolni:

```
Random r = new Random();
r.Next(-100, 100);
```

Ebben az esetben a generált szám *-100* és *100* közé esik majd.

11.4 Maximum/minimumkeresés

Készítsünk programot, amely egy tömb elemei közül kiválasztja a legnagyobbat/legkisebbet. Az elemeket a felhasználótól kérjük be.

12. Objektum-orientált programozás - elmélet

A korai programozási nyelvek nem az adatokra, hanem a műveletekre helyezték a hangsúlyt. Ekkoriban még főleg matematikai számításokat végeztek a számítógépekkel. Ahogy aztán a számítógépek széles körben elterjedtek, megváltoztak az igények, az adatok pedig túl kompexekké váltak ahhoz, hogy a procedurális módszerrel kényelmesen és hatékonyan kezelni lehessen őket.
Az első objektum-orientált programozási nyelv a Simula 67 volt. Tervezői Ole-Johan Dahl és Kristen Nygaard hajók viselkedését szimulálták, ekkor jött az ötlet, hogy a különböző hajótípusok adatait egy egységként kezeljék, így egyszerűsítve a munkát.
Az OOP már nem a műveleteket helyezi a középpontra, hanem az egyes adatokat (adatszerkezeteket) és köztük lévő kapcsolatot (hierarchiát).
Ebben a fejezetben az OOP elméleti oldalával foglalkozunk, a cél a paradigma megértése, gyakorlati példákkal a következő részekben találkozhatunk (szintén a következő részekben található meg néhány elméleti fogalom amely gyakorlati példákon keresztül érthetőbben megfogalmazható, ezért ezek csak később lesznek tárgyalva, pl.: polimorfizmus).

12.1 UML

Az OOP tervezés elősegítésére hozták létre az UML –t (*Unified Modelling Language*). Ez egy általános tervezőeszköz, a célja egy minden fejlesztő által ismert közös jelrendszer megvalósítása. A következőkben az UML eszközeit fogjuk felhasználni az adatok közti relációk grafikus ábrázolásához.

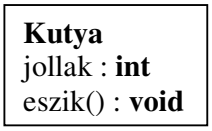
12.2 Osztály

Az OOP világában egy osztály olyan adatok és műveletek összessége, amellyel leírhatjuk egy modell (vagy entitás) tulajdonságait és működését. Legyen például a modellünk a kutya állatfaj. Egy kutyának vannak tulajdonságai (pl. életkor, súly, stb.) és van meghatározott viselkedése (pl. csóválja a farkát, játszik, stb.).
Az UML a következőképpen jelöl egy osztályt:

Amikor programot írunk, akkor az adott osztályból (osztályokból) létre kell hoznunk egy (vagy több) példányt, ezt pédányosításnak nevezzük. Az osztály és példány közti különbségre jó példa a recept (osztály) és a sütemény (példány).

12.3 Adattag és metódus

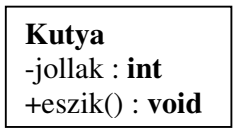
Egy objektumnak az életciklusa során megváltozhat az állapota, tulajdonságai. Ezt az állapotot valahogy el kell tudnunk tárolni illetve biztosítani kell a szükséges műveleteket a tulajdonságok megváltoztatásához (pl. a kutya eszik(ez egy művelet), ekkor megváltozik a „jóllakottság” tulajdonsága).
A tulajdonságokat tároló változókat adattagnak (vagy mezőnek), a műveleteket metódusnak nevezzük. A műveletek összességét felületnek is nevezik.
Módosítsuk a diagramunkat:



Az adattagokat *név : típus* alakban ábrázoljuk, a metódusokat pedig *név(paraméterlista) : visszatérési_érték* formában. Ezekkel a fogalmakkal egy későbbi fejezet foglalkozik.

12.4 Láthatóság

Az egyes tulajdonságokat, metódusokat nem biztos, hogy jó közszemlére bocsátani. Az OOP egyik alapelve, hogy a felhasználó csak annyi adatot kapjon amennyi feltétlenül szükséges. A kutyás példában az *eszik()* művelet magába foglalja a rágást, nyelést, emésztést is, de erről nem fontos tudnunk, csak az evés ténye számít.
Ugyanígy egy tulajdonság (adattag) esetében sem jó, ha mindenki hozzájuk fér (az elfogadható, ha a közvetlen család hozzáfér a számlámhoz, de idegenekkel nem akarom megosztani).
Az ős-OOP szabályai háromféle láthatóságot fogalmaznak meg, ez nyelvtől függően bővülhet, a C# láthatóságairól a következő részekben lesz szó.
A háromféle láthatóság:
Public: mindenki láthatja (UML jelölés: **+**).
Private: csakis az osztályon belül elérhető, illetve a leszármazott osztályok is láthatják, de nem módosíthatják (a származtatás/öröklődés hamarosan jön) (UML jelölés: **-**).
Protected: ugyanaz mint a **private**, de a leszármazott osztályok módosíthatják is (UML jelölés: **#**).
A *Kutya* osztály most így néz ki:

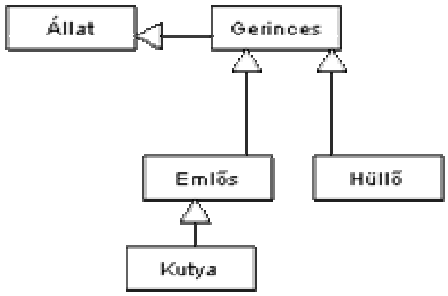


12.5 Egységbezárás

A „hagyományos”, nem OO programnyelvek (pl. a C) az adatokat és a rajtuk végezhető műveleteket a program külön részeként kezelik. Bevett szokás ezeket elkülöníteni egy önálló forrásfile –ba, de ez még mindig nem elég biztonságos. A kettő között nincs összerendelés, ezért más programozók gond nélkül átírhatják egyiket vagy másikat, illetve hozzáférnek a struktúrákhoz és nem megfelelően használják fel azokat.
Az OO paradigma egységbe zárja az adatokat és a hozzájuk tartozó felületet, ez az ún. egységbezárás (*encapsulation* vagy *information hiding*). Ennek egyik nagy előnye, hogy egy adott osztály belső szerkezetét gond nélkül megváltoztathatjuk, mindössze arra kell figyelni, hogy a felület ne változzon (pl. egy autót biztosan tudunk kormányozni, attól függetlenül, hogy az egy személyautó, traktor vagy forma-1 –es gép).

12.6 Öröklődés

Az öröklődés vagy származtatás az új osztályok létrehozásának egy módja. Egy (vagy több) már létező osztályból hozunk létre egy újat, úgy, hogy az minden szülőjének tulajdonságát örökli vagy átfogalmazza azokat. A legegyszerűbben egy példán keresztül érthető meg. Legyen egy „Állat" osztályunk. Ez egy eléggé tág fogalom, ezért szűkíthetjük a kört, mondjuk a „Gerinces Állatok" –ra. Ezen belül megkülönböztethetünk „Emlős" –t vagy „Hüllő" –t. Az „Emlős" osztály egy leszármazottja lehet a „Kutya" és így tovább. Az öröklődést specializálásnak is nevezik. A specializálás során az osztályok között ún. „az-egy" (is-a) reláció áll fenn. Így amikor azt mondjuk, hogy a „Kutya" az egy „Állat" akkor arra gondolunk, hogy a „Kutya" egy specializáltabb forma, amelynek megvan a saját karakterisztikája, de végeredményben egy „Állat". Ennek a gondolatmenetnek a gyakorlati felhasználás során lesz jelentősége.



A diagram a fenti példát ábrázolja. UML -ül az öröklődést „üres" nyíllal jelöljük, amely a specializált osztály felől mutat az általánosabbra. Az osztályok között fenálló kapcsolatok összeségét hierachiának nevezzük.

Előfordul, hogy nem fontos számunkra a belső szerkezet, csak a felületet szeretnénk átörökíteni, hogy az osztályunkat fel tudja használni a programunk egy másik része (ilyen például a már említett *foreach* ciklus). Ilyenkor nem egy „igazi" osztályról, hanem egy interfészről beszélünk, amelynek nincsenek adattagjai csakis a műveleteket deklarálja. A C# rendelkezik önálló interfészekkel, de ez nem minden programnyelvre igaz, ezért ők egy hasonló szerkezetet ún. absztrakt osztályokat használnak. Ezekben előfordulnak adattagok is, de leginkább a felület definiálására koncentrálnak. A C# nyelvi szinten támogat absztrakt osztályokat is, a kettő között ugyanis lényegi különbség van. Az interfészek az osztálytól függetlenek, csakis felületet biztosítanak (például az *IEnumerable* és *IEnumerator* a *foreach* –nek (is) biztosítanak felületet, az nem lényeges, hogy milyen osztályról van szó). Az absztrakt osztályok viszont egy őshöz kötik az utódokat (erre az esetre példa az „Állat" osztály, amelyben mondjuk megadunk egy absztrakt „evés" metódust, amit az utódok megvalósítanak - egy krokodil nyílván máshogy eszik mint egy hangya, de az evés az állatokhoz köthető valami, ezért közös).

13. Osztályok

Osztályt a *class* kulcsszó segítségével deklarálhatunk:

```
using System;

class Dog
{

}

class Program
{
    static public void Main()
    {

    }
}
```

Látható, hogy a főprogram és a saját osztályunk elkülönül. Felmerülhet a kérdés, hogy a fordító, honnan tudja, hogy melyik a „fő" osztály? A helyzet az, hogy a *Main* egy speciális metódus, ezt a fordító automatikusan felismeri és megjelöli, mint a program belépési pontját. Igazából lehetséges több *Main* nevű metódust is létrehozni, ekkor a fordítóprogramnak meg kell adni (a */main* kapcsoló segítségével), hogy melyik az igazi belépési pont. Ettől függetlenül ezt a megoldást ha csak lehet kerüljük el.

A fenti példában a lehető legegyszerűbb osztályt hoztuk létre. A C++ programozók figyeljenek arra, hogy az osztálydeklaráció végén nincs pontosvessző.
Az osztályunkból a következőképpen tudunk létrehozni egy példányt:

```
Dog d = new Dog();
```

Hasonlóan a C++ nyelvhez itt is a *new* operátorral tudunk új objektumot létrehozni.

13.1 Konstruktorok

Minden esetben amikor példányosítunk egy speciális metódus, a konstruktor fut le. Bár a fenti osztályunkban nem definiáltunk semmilyen metódust, ettől függetlenül rendelkezik alapértelmezett (azaz paraméter nélküli) konstruktorral. Ez igaz minden olyan osztályra, amelynek nincs konstruktora (amennyiben bármilyen konstruktort létrehoztunk akkor ez a lehetőség megszűnik). Az alapértelmezett konstruktor semmi mást nem tesz, mint helyet foglal a memóriában és létrehoz egy példányt. Az adattagok – ha vannak – automatikusan a nekik megfelelő nullértékre inicializálódnak(pl.: *int* -> 0, *bool* -> *false*, referenciatípusok -> *null*).
A C++ nyelvet ismerők vigyázzanak, mivel itt csak alapértelmezett konstruktort kapunk automatikusan, értékadó operátort illetve másoló konstruktort nem. Ugyanakkor minden osztály a *System.Object* –ből származik (még akkor is ha erre nem utal semmi), ezért néhány metódust (például a típus lekérdezéséhez) a konstruktorhoz hasonlóan azonnal használhatunk.

Jelen pillanatban az osztályunkat semmire nem tudjuk használni, ezért készítsünk hozzá néhány adattagot és egy konstruktort:

```
using System;

class Dog
{
    private string name;
    private int age;

    public Dog(string n, int a)
    {
        this.name = n;
        this.age = a;
    }
}

class Program
{
    static public void Main()
    {
        Dog d = new Dog("Rex", 2);
    }
}
```

A konstruktor neve meg kell egyezzen az osztály nevével és semmilyen visszatérési értéke sem lehet. A mi konstruktorunk két paramétert vár, a nevet és a kort (metódusokkal és paramétereikkel a következő rész foglalkozik bővebben). Ezeket a példányosításnál muszáj megadni, egyébként nem fordul le a program. Egy osztálynak paraméterlistától függően bármennyi konstruktora lehet és egy konstruktorból hívhatunk egy másikat a *this* –el:

```
class MyClass
{
    public MyClass() : this(10) //A másik konstruktort hívtuk
    {

    }

    public MyClass(int x)
    {

    }
}
```

Ilyen esetekben a paraméter típusához leginkább illeszkedő konstruktor kerül majd hívásra.
A példában a konstruktor törzsében értéket adtunk a mezőknek *this* hivatkozással, amely mindig arra a példányra mutat, amelyen meghívták. Nem kötelező használni, ugyanakkor hasznos lehet, ha sok adattag/metódus van, illetve ha a paraméterek neve megegyezik az adattagokéval. A fordítóprogram automatikusan „odaképzeli” magának a fordítás során, így mindig tudja mivel dolgozik.
Az adattagok *private* elérésűek (ld. elméleti rész), azaz most csakis az osztályon belül használhatjuk őket, például a konstruktorban, ami viszont publikus.

Nem csak a konstruktorban adhatunk értéket a mezőknek, hanem használhatunk ún. inicializálókat is:

```
class Dog
{
    private string name = "Rex";
    private int age = 5;

    public Dog(string n, int a)
    {
        this.name = n;
        this.age = a;
    }
}
```

Az inicializálás mindig a konstruktor előtt fut le, ez egyben azt is jelenti, hogy az felülbírálhatja. Ha a *Dog* osztálynak ezt a módosított változatát használtuk volna fentebb, akkor a példányosítás során felülírnánk az alapértelmezettnek megadott kort. Az inicializálás sorrendje megegyezik a deklarálás sorrendjével (fölűlről lefelé halad).

A konstruktorok egy speciális változata az ún. másoló- vagy copy – konstruktor. Ez paramétereként egy saját magával megegyező típusú objektumot kap és annak értékeivel inicializálja magát. Másoló konstruktort általában az értékadó operátorral szoktak implementálni, de az operátortúlterhelés egy másik fejezet témája így most egyszerűbben oldjuk meg:

```
class Dog
{
    private string name = "Rex";
    private int age = 5;
    public Dog(string n, int a)
    {
        this.name = n;
        this.age = a;
    }

    public Dog(Dog otherDog)
    {
        this.name = otherDog.Name;
        this.age = otherDog.Age;
    }

    public int Age
    {
        get { return age; }
    }
    public string Name
    {
        get { return name; }
    }
}
```

Ún. tulajdonságokat használtunk fel (erről hamarosan). Most már felhasználhajuk az új konstruktort:

```
Dog d = new Dog("Rex", 6);
Dog newDog = new Dog(d);
```

13.2 Adattagok

Az adattagok vagy mezők olyan változók, amelyeket egy osztályon (vagy struktúrán) belül deklaráltunk. Az eddigi példáinkban is használtunk már adattagokat, ilyenek voltak a *Dog* osztályon belüli *name* és *age* változók.
Az adattagokon használhatjuk a *const* típusmódosítót is, ekkor a deklarációnál értéket kell adnunk a mezőnek, hasonlóan az előző fejezetben említett inicializáláshoz. Ezek a mezők pontosan ugyanúgy viselkednek mint a hagyományos konstansok. Egy konstans mezőt nem lehet explicite statikusnak jelölni, mivel a fordító egyébként is úgy fogka kezelni (ha egy adat minden objektumban változatlan, felesleges minden alkalommal külön példányt készíteni belőle).

A mezőkön alkalmazható a *readonly* módosító is, ez két dologban különbözik a konstansoktól: az értékadás elhalasztható a konstruktorig és az értékül adott kifejezés értékének nem kell ismertnek lennie fordítási időben.

13.3 Láthatósági módosítók

A C# nyelv ötféle módosítót ismer:

public: az osztályon/struktúrán kívűl és belül teljes mértékben hozzáférhető.
private: csakis a tartalmazó osztályon belül látható, a leszármazottak sem láthatják, osztályok/struktúrák esetében az alapértelmezés.
protected: csakis a tartalmazó osztályon és leszármazottain belül látható.
internal: csakis a tartalmazó (és a barát) assembly(ke) –n belül látható.
protected internal: a *protected* és *internal* keveréke.

13.4 Parciális osztályok

C# nyelven létrehozhatunk ún. parciális osztályokat (*partial class*). Egy parciális osztály definiciója több részből (tipikusan több forrásfile –ból) is állhat.

```
//file1.cs
partial class PClass
{
    public PClass()
    {
        /*...*/
    }
}
//file2.cs
partial class PClass
{
    public void DoSomeThing(){ /*...*/ }
}
```

Az osztály összes „darabjánál” kötelező kitenni a *partial* kulcsszót.
A C# a parciális osztályokat főként olyan esetekben használja, amikor az osztály egy részét a fordító generálja (pl. a grafikus felületű alkalmazásoknál a kezdeti beállításokat az InitializeComponent() metódus végzi, ezt teljes egészében a fordító készíti el).

A következő téma megértését elősegítheti a következő Metódusok című rész elolvasása

A C# 3.0 már engedélyezi parciális metódusok használatát is, ekkor a metódus deklarációja és definíciója szétoszlik:

```
//file1.cs
partial class PMClass
{
    //deklaráció
    partial public void PartialMethod(string param);
}

//file2.cs
partial class PMClass
{
    //definíció
    partial public void PartialMethod(string param)
    {
        /*...*/
    }
}
```

A *partial* kulcsszót ilyenkor is ki kell tenni minden előfordulásnál. Csakis parciális osztály tartalmazhat parciális metódust.

13.5 Beágyazott osztályok

Egy osztály tartalmazhat metódusokat, adattagokat és más osztályokat is. Ezeket a „belső” osztályokat beágyazott (*nested*) osztálynak nevezzük. Egy beágyazott osztály hozzáfér az őt tartalmazó osztály minden tagjához (beleértve a *private* elérésű tagokat és más beágyazott osztályokat is). Egy ilyen osztályt általában elrejtünk, de ha mégis publikus elérésűnek deklaráljuk, akkor a „külső” osztályon keresztül érhetjük el.

```
//a beágyazott osztály nem látható
class Outer
{
    private class Inner
    {

    }
}

//de most már igen
class Outer
{
```

```
    public class Inner
    {

    }
}

//példányosítás:
Outer.Inner innerClass = new Outer.Inner();
```

13.6 Objektum inicializálók

A C# 3.0 objektumok példányosításának egy érdekesebb formáját is tartalmazza:

```
class Person
{
    private string name;

    public Person() { }

    public string Name
    {
        get { return name; }
        set { this.name = value; }
    }
}

Person p = new Person { Name = "Istvan" };
```

Ilyen esetekben vagy egy nyílvános tagra, vagy egy tulajdonságra hivatkozunk (ez utóbbit használtuk). Természetesen, ha létezik paraméteres konstruktor, akkor is használhatjuk ezt a beállítási módot, a következőképpen:

```
Osztaly valtozo = new Osztaly(/*parameterek*/) { };
```

13.7 Destruktorok

A destruktorok a konstruktorokhoz hasonló speciális metódusok amelyek az erőforrások felszabadításáért felelnek. A C# objektumai által elfoglalt memóriát a szemétgyüjtő (Garbage Collector) szabadítja fel, abban az esetben ha az objektumra nincs érvényes referencia:

```
MyClass mc = new MyClass(); //mc egy MyClass objektumra mutat
mc = null; //az objektumra már nem mutat semmi, felszabadítható
```

A szemétgyüjtő működése nem determinisztikus, azaz előre nem tudjuk megmondani, hogy mikor fut le, ugyanakkor kézzel is meghívható, de ez nem ajánlott:

```
MyClass mc = new MyClass();
mc = null;
GC.Collect();
GC.WaitForPendingFinalizers();
```

A GC a hatékonyság szempontjából a legjobb időt választja ki arra, hogy elvégezze a gyűjtést (de legkésőbb akkor működésbe lép amikor elfogy a memória).

A GC mielőtt megsemmísiti az objektumokat meghívja a hozzájuk tartozó destruktort, másnéven *Finalizert.* Vegyük a következő kódot:

```
class DestructableClass
{
    public DestructableClass() { }

    ~DestructableClass() { }
}
```

A destruktor neve *tilde* jellel (~) kezdődik, neve megegyezik az osztályéval és nem nem lehet semmilyen módosítója vagy paramétere. A fenti kód valójában a következő formában létezik:

```
protected override void Finalize()
{
    try
    {

    }
    finally
    {
        base.Finalize();
    }
}
```

Minden típus örökli a *System.Object* –től a *Finalize()* metódust, amelyet a destruktorral felülírunk. A *Finalize()* először felszabadítja az osztály erőforrásait (a destruktorban általunk megszabott módon), aztán meghívja az ősosztály *Finalize()* metódusát (ez legalább a *System.Object* –é lesz) és így tovább amíg a lánc végére nem ér:

```
class Base
{
    ~Base()
    {
        Console.WriteLine("Base destruktor..");
    }
}

class Derived
{
    ~Derived()
    {
        Console.WriteLine("Derived destruktor...");
    }
}
```

Ezután:

```
class Program
{
    static public void Main()
    {
        Derived d = new Derived();

        Console.ReadKey();
    }
}
```

Amikor elindítjuk a programot nem történik semmi, ahogy az várható is volt. Amikor viszont lenyomunk egy gombot a *Console.ReadKey()* –nek elindul a GC és a kimenet a következő lesz:

```
Derived destruktor…
Base destruktor…
```

Elsőként a leszármazott, majd az ősobjektum semmisül meg.
A destruktorokra vonatkozik néhány szabály, ezek a következőek:

- Egy osztálynak csak egy destruktora lehet
- A destruktor nem örökölhető
- A destruktort nem lehet direkt hívni, ez mindig automatikusan történik
- Destruktora csakis osztálynak lehet, struktúrának nem

14. Metódusok

Bizonyára szeretnénk, ha a kutyánk nem csak létezne a semmiben, hanem csinálna is valamit. Készítsünk néhány metódust (ugye senki nem felejtette el, a metódusokal művelteket tudunk végezni az objektumon):

```
class Dog
{
    private string name;
    private int age;

    public Dog(string n, int a)
    {
        name = n;
        age = a;
    }

    public void Sleep(int time)
    {
        Console.WriteLine("A kutya {0} orat alszik...");
    }

    public void Eat()
    {
        Consol.WriteLine("A kutya most eszik...");
    }
}
```

Most már tud enni és aludni, a két legfontosabb dolgot. Nézzük az alvást:

```
    public void Sleep(int time)
    {
        Console.WriteLine("A kutya {0} orat alszik...", time);
    }
```

Először megadjuk a láthatóságot, ez publikus lesz, hogy meghívhassuk. Ezután a metódus visszatérési értékének a típusa jön.
Mire jó a visszatérési érték? Az objektumainkon nem csak műveleteket végzünk, de szeretnénk lekérdezni az állapotukat is és felhasználni ezeket az értékeket.
Egy másik lehetőség, amivel már találkoztunk az *int.Parse(…)* metódus kapcsán, azok a „kisegítő” függvények amelyek nem kapcsolódnak közvetlenül az objektum életciklusához, de hasznosak és logikailag az osztály hatókörébe tartoznak. Visszatérési értékkel rendelkező metódust használhatunk minden olyan helyen, ahol a program valamilyen típust vár (értékadás, logikai kifejezések, metódus paraméterei, etc..).
Kanyarodjunk vissza az eredeti szálhoz! Az alvás metódusunk nem tér vissza semmivel, ezt a *void* kulcsszóval jelezzük, ez az ún. általános típus (a későbbiekben még lesz róla szó).
Ezután a metódus neve és paraméterlistája következik.

A függvényeket az objektum neve után írt pont operátorral hívhatjuk meg (ugyanez érvényes a publikus adattagokra, tulajdonságokra, stb. is):

```
Dog d = new Dog("Rex", 2);
d.Eat();
d.Sleep(3);
```

Erre a kimenet:

```
A kutya most eszik...
A kutya 3 orat alszik...
```

14.1 Paraméterek

Az objektummal való kommunikáció érdekében képesnek kell lennünk kívűlről megadni adatokat, vagyis paramétereket. A paraméterek számát és típusait a függvény deklarációjában, vesszővel elválasztva adjuk meg. Egy metódusnak gyakorlatilag bármennyi paramétere lehet. A metódus nevét és paraméterlistáját aláírásnak, szignatúrának vagy prototípusnak nevezzük. A paraméterek a metóduson belül lokális változókként viselkednek, és a paraméter nevével hivatkozunk rájuk.

A C# nyelvben paraméterek átadhatunk érték és cím szerint is. Előbbi esetben egy teljesen új példány jön létre az adott osztályból, amelynek értékei megegyeznek az eredetiével. A másik esetben egy az objektumra mutató referencia adódik át valójában, tehát az eredeti objektummal dolgozunk. Az érték- és referenciatípusok különbözően viselkednek az átadás szempontjából. Az értéktípusok alapértelmezetten érték szerint adódnak át, míg a referenciatípusoknál a cím szerinti átadás az előre meghatározott viselkedés. Utóbbi esetben van azonban egy kivétel, mégpedig az, hogy míg a referenciatípus értékeit megváltoztathatjuk (és ez az eredeti objektumra is hat) addig magát a referenciát már nem. Ha ezt mégis megtesszük, az a program futására nem hat, de a változás csakis a metóduson belül lesz észlelhető.

```
using System;

class Program
{
    static public void AllocArray(int[] array, int size)
    {
        array = new int[size];
        Console.WriteLine(array.Length); //100
    }

    static public void Main()
    {
        int[] a = new int[10];
        AllocArray(a, 100);
        Console.WriteLine(a.Length); //10
    }
}
```

Ha mégis módosítani akarjuk a referenciatípus referenciáját, akkor külön jeleznünk kell azt, hogy cím szerint akarjuk átadni.
Kétféleképpen adhatunk át paramétert cím szerint. Az első esetben az átadott objektum inicializálva kell legyen. A cím szerinti átadást a forráskódban is jelölni kell, mind a metódus prototípusánál, mind a hívás helyén:

```
using System;

class Dog
{
    private int age;
    private string name;

    public Dog(string s, int a)
    {
        name = s;
        age = a;
    }

    /*
    már elkészített metódusok
   */

    public void PlayOtherDog(ref Dog other)
    {
        Console.WriteLine("Ket kutya jatszik...");
    }
}



class Program
{
    static public void Main()
    {
        Dog d1 = new Dog("Rex", 2);
        Dog d2 = new Dog("Rinti", 3);
        d1.PlayOtherDog(ref d2);
        Console.ReadKey();
    }
}
```

A *ref* kulcsszóval jelöljük meg azokat a paramétereket, amelyeknek a címét akarjuk átadni. Ezt a hívás helyén is meg kell tennünk.

A cím szerinti átadás másik formájában nem inicializált paramétert is átadhatunk, de ekkor feltétel, hogy a metóduson belül állítsuk be. A használata megegyezik a *ref* –el, azaz a szignatúrában és a hívásnál is jelezni kell a szándékunkat. A használandó kulcsszó az *out* (Nomen est omen – A név kötelez):

```
using System;

class Dog
{
    private int age;
```

```
    private string name;

    public Dog(string s, int a)
    {
        name = s;
        age = a;
    }

    /*
    már elkészített metódusok
   */

    public void PlayOtherDog(ref Dog other)
    {
        Console.WriteLine("Ket kutya jatszik...");
    }

    public void InitDog(out Dog idog)
    {
        idog = new Dog("Lassie", 4);
    }
}

class Program
{
    static public void Main()
    {
        Dog d1 = new Dog("Rex", 2);
        Dog d2 = new Dog("Rinti", 3);
        d1.PlayOtherDog(ref d2);
        Dog d3;
        d2.InitDog(out d3);
        Console.ReadKey();
    }
}
```

Ebben a példában nem lett volna muszáj cím szerint átadnunk a paramétereket, hiszen az objektumok értékeit nem módosítottuk.

Amikor nem tudjuk, hogy pontosan hány paramétert akarunk átadni, akkor használhatunk paramétertömböket:

```
static void Func(params int[] paramarray)
{
    /*Itt csinalunk valamit*/
}

static public void Main()
{
    int[] array = new int[]
        {
            1, 2, 3, 4
        };

    Func(array);
}
```

A paramétertömböt a *params* kulcsszóval jelezzük, ezután a metódus belsejében pontosan úgy viselkedik, mint egy normális tömb.

A cím és érték szerinti átadásról a következő oldalakon olvashatunk többet:

http://msdn.microsoft.com/en-us/library/9t0za5es.aspx
http://msdn.microsoft.com/en-us/library/s6938f28.aspx

14.2 Parancssori paraméterek

A *Main* –nek létezik paraméteres változata is:

```
using System;

class Program
{
    static public void Main(String[] args)
    {

    }
}
```

Ekkor a *Main* paramétere egy tömb, ami a parancssori paramétereket tárolja el. Pl. így futtatjuk a programot:

```
program.exe elso masodik harmadik
```

Ha ki szeretnénk iratni a paramétereket, akkor így módosítjuk a programot:

```
using System;

class Program
{
    static public void Main(String[] args)
    {
        foreach(string s in args)
        {
            Console.WriteLine(s);
        }
    }
}
```

Erre a kimenet a következő:

```
elso
masodik
harmadik
```

14.3 Kiterjesztett metódusok

A C# 3.0 lehetőséget ad arra, hogy egy már létező típushoz új funkciókat adjunk, anélkül, hogy azt közvetlenül módosítanánk. Egy kiterjesztett metódus (extension method) minden esetben egy statikus osztály statikus metódusa kell legyen.
Egészítsük ki a *string* típust egy metódussal, ami kiírja a képernyőre az adott karaktersorozatot:

```
static public class StringPrinter
{
    static public void Print(this string s)
    {
        Console.WriteLine(s);
    }
}
```

A this módosító után a paraméter típusa következik, amely meghatározza a kiterjesztett osztály típusát. Ezután a metódust így használjuk:

```
string s = "abcd";
s.Print();

//vagy hagyományos statikus metódusként:

StringPrinter.Print(s);
```

Nemcsak osztályt, de interfészt is bővíthetünk.
Ha két kiterjesztett metódus ugyanazzal a szignatúrával rendelkezik, akkor a hagyományos statikus úton kell hívnunk őket. Ha nem így teszünk akkor a speciálisabb paraméterű metódus fog meghívódni.

Egy kiterjesztett metódust nem defiinálhatunk beágyazott osztályban.

A kiterjesztett metódusokról a következő oldalon olvashatunk többet:

http://msdn.microsoft.com/en-us/library/bb383977.aspx

14.4 Gyakorló feladatok

1. Készítsünk metódusokat a kutya osztályhoz, amelyekkel elvégezhetjük a szokásos teendőket: pl.: fürdetés, sétáltatás, stb.

2. Készítsünk osztályt, amely egy tömböt tartalmaz. Az osztály konstruktorában foglaljuk le a tömböt (a méretét a konstruktor paramétere tartalmazza). Írjunk hozzá egy metódust, amely kiírja a tömböt a konzolra. Pl.:

```
MyArray ma = new MyArray(10); //10 elemű tömb
ma.Print(); //kiírjuk
```

3. Készítsünk egy kiterjesztett metódust, amely kiírja egy egész számokból (int) álló tömb elemeit (a szükséges statikus osztályokról a következő fejezet szól).

15. Statikus tagok

A hagyományos adattagok és metódusok objektumszinten léteznek, azaz minden példány saját példánnyal rendelkezik. Gyakran van azonban szükségünk arra, hogy minden példány egy közös tagot használjon, pl. ha szeretnénk számolni, hogy hány objektumot hoztunk létre. Az összes statikus tagból összesen egy darab létezik.
A statikus tagok jelentősége a C# tisztán objektum orientáltságában rejlik, ugyanis nem definiálhatunk globális mindenki számára egyformán elérhető tagokat. Ezt váltják ki a statikus adattagok és metódusok. Így megkülönböztetünk példány (*instance*) és statikus tagokat.

15.1 Statikus adattagok

Statikus adattagot (és metódust ld. következő fejezet) a *static* kulcsszó segítségével hozhatunk létre:

```
public Animals
{
    static public int animalCounter = 0;

    public Animals()
    {
        ++animalCounter;
    }
}
```

A példában a statikus adattag értékét minden alkalommal megnöveljük eggyel, amikor meghívjuk a konstruktort. Vagyis a példányok számát tároljuk el benne. A statikus tagokhoz az osztály nevén (és nem egy példányán) keresztül férünk hozzá:

```
Console.WriteLine("A peldanyok szama: {0}", Animals.animalCounter);
```

A statikus tagok azelőtt inicializálódnak, hogy először hozzáférnénk, illetve mielőtt a statikus konstruktor – ha van – lefut.

15.2 Statikus metódusok

A statikus metódusokból, akárcsak az adattagokból osztályszinten egy darab létezik. Statikus metódus létrehozása a következőképpen történik:

```
public class Math
{
    static public void Pi()
    {
        /*...*/
    }
}
```

Ezt így tudjuk meghívni:

```
Console.WriteLine("Pi erteke: {0}", Math.Pi());
```

Bármely statikus taghoz az osztályon – és nem egy példányon – keresztűl férünk hozzá. A példányon való hívás fordítási hibát jelent.

Statikus metódust általában akkor használunk, ha nem egy példány állapotának a megváltoztatása a cél, hanem egy osztályhoz kapcsolódó művelet elvégzése. A statikus metódusok nem férhetnek hozzá a példánytagokhoz (legalábbis direkt módon nem).

15.3 Statikus tulajdonságok

Ennek a fejezetnek a megértéséhez ajánlott elolvasni a Tulajdonságok című részt

A statikus tulajdonságok a C# egy viszonylag ritkán használt lehetősége. Általában osztályokhoz kapcsolódó konstans értékek lekérdezésére használjuk:

```
public class Math
{
    static public double Pi
    {
        get { return 3.14; }
    }
}
```

15.4 Statikus konstruktor

A statikus konstruktor a statikus tagok beállításáért felel. A statikus konstruktor azután fut le, hogy a program elindult, de azelőtt, hogy egy példány keletkezik.
A statikus konstruktornak nem lehet láthatóságot adni, illetve nincsenek paraméterei sem. Mivel ez is egy statikus metódus nem férhet hozzá a példánytagokhoz. Példa statikus konstruktorra:

```
public class Person
{
    public static string name;

    static Person()
    {
        name = "Anonymus";
    }
}
```

15.5 Statikus osztályok

Egy osztályt statikusnak jelölhetünk, ha csak és kizárólag statikus tagjai vannak. Egy statikus osztályból nem hozható létre példány, nem lehet példánykonstruktora (de statikus igen) és mindig lezárt (ld. Öröklődés). A fordító minden esetben ellenőrzi ezeknek a feltételeknek a teljesülését.

```
static class Math
{
    static private double pi = 3.141516;
```

```
    static public double Pi() { return pi; }
    static public double Cos(double x) { /*...*/ }
    /*Egyéb tagok*/
}
```

15.6 Gyakorló feladatok

1. Készítsük el a *Math* statikus osztályt! Az osztály tartalmazzon metódusokat a négy alapműveletre (illetve tetszőleges matematikai műveletre). Az összeadás és szorzás metódusok a paramétereiket a *params* tömbön (ld. metódusok) kapják meg, vagyis bármennyi taggal elvégezhetőek legyenek a műveletek.

2. Az előző feladat osztályát felhasználva készítsünk egy egyszerű számológépet! Azoknál a metódusoknál, amelyek meghatározatlan számú paramétert várnak ciklussal olvassuk be a tagokat!

16. Tulajdonságok

A tulajdonságokat (vagy property –ket) a mezők közvetlen módosítására használjuk, anélkül, hogy megsértenénk az egységbezárás elvét. A tulajdonságok kívűlről nézve pontosan ugyanolyanok, mint a hagyományos változók:

```
using System;

class Person
{
    private string name;

    public Person(string name)
    {
        this.name = name;
    }

    //Ez a tulajdonság
    public string Name
    {
        get { return this.name; }
        set { this.name = value; }
    }
}

class Program
{
    static public void Main()
    {
        Person p = new Person("Istvan");
        Console.WriteLine(p.Name);
    }
}
```

A tulajdonsággal lekérdeztük a „személy” nevét. Hagyomány - de nem kötelező - hogy a tulajdonság neve az adattag nevének nagybetűvel kezdődő változata.

```
    public string Name
    {
        get { return this.name; }
        set { this.name = value; }
    }
```

Elsőként megadtuk a láthatóságot, aztán a visszatérési értéket és a nevet. A tulajdonság törzse két részre az ún. *getter* –re és *setter* –re oszlik. Előbbivel lekérdezzük az adott értéket, épp úgy mint egy hagyományos metódus esetében tennénk. A *setter* már érdekesebb egy kicsit. Minden esetben amikor az adott tulajdonságnak értéket adunk egy „láthatatlan” paraméter jön létre, aminek a neve *value*, ez fogja tartalmazni a tulajdonság új értékét. A *settert* így használhatjuk:

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

```
    {
        Person p = new Person("Istvan");
        p.Name = "Dezso";
        Console.WriteLine(p.Name); //Dezso
    }
}
```

Látható, hogy pontosan úgy működik mint egy változó.
Egyik esetben sem vagyunk rákényszerítve, hogy azonnal visszadjuk/beolvassuk az adttag értékét, tetszés szerint végezhetünk műveleteket is rajtuk:

```
public string Name
{
    get { return ("Mr." + this.name); }
    set { this.name = value; }
}
```

Arra is van lehetőség, hogy csak az egyiket használjuk, ekkor csak-írható/csak-olvasható tulajdonságokról beszélünk (bár előbbi használata elég ritka).

A *getter/setter* láthatóságát külön megadhatjuk, ráadásul ezeknek nem kell egyezniük sem:

```
public string Name
{
    get { return ("Mr." + this.name); }
    private set { this.name = value; }
}
```

Ebben az esetben a *getter* megtartja az alapértelmezett publikus láthatóságát, míg a *setter private* elérésű lesz, kívűlről nem lehet meghívni.

A C# 3.0 rendelkezik egy nagyon érdekes újítással az ún. automatikus tulajdonságokkal. Nem kell létrehoznunk sem az adattagot, sem a teljes tulajdonságot, a fordító mindkettőt legenerálja nekünk:

```
class Person
{
    public string Name { get; set; }
}
```

A fordító automatikusan létrehoz egy *private* elérésű *name* nevű adattagot és elkészíti hozzá a *gettert/settert* is. Van azonban egy probléma, méghozzá az, hogy a fordítás pillanatában ez a változó még nem létezik. Viszont van *setter*, amit használhatunk és már létezik:

```
class Person
{
    private string Name { get; set; }

    public Person(string n)
    {
```

```
            name = n; //Ez nem működik
            Name = n; //Ez viszont igen
        }
}
```

17. Indexelők

Az indexelők hasonlóak a tulajdonságokhoz, azzal a különbséggel, hogy nem névvel, hanem egy indexxel férünk hozzá az adott információhoz. Általában olyan esetekben használják, amikor az osztály/struktúra tartalmaz egy tömböt vagy valamilyen gyűjteményt (vagy olyan objektumot, amely maga is megvalósít egy indexelőt).
Egy indexelőt így implementálhatunk:

```
class Names
{
    private ArrayList nameList;

    public Names()
    {
        nameList = new ArrayList();
        nameList.Add("Istvan");
        nameList.Add("Judit");
        nameList.Add("Bela");
        nameList.Add("Eszter");
    }

    public string this [int idx]
    {
        get
        {
            if(idx < nameList.Count)
            {
                return nameList[idx].ToString();
            }
            else return null;
        }
    }
}
```

Ez gyakorlatilag egy „névtelen tulajdonság”, a *this* mutató mutat az aktuális objektumra, amin az indexelőt definiáltuk. Nemcsak egy indexet adhatunk meg, hanem tetszés szerintit, illetve az index típusa sem kötött. Pl.:

```
public int this [int idx1, int idx2]
{
    /*...*/
}
```

18. Gyakorló feladatok II.

18.1 Tömb típus

Valósítsunk meg egy tömb típust. Az osztály tartalmazzon metódusokat/tulajdonságokat/indexelőket a szokásos tömbműveletek (kiírás, rendezés, stb…) elvégzésére. Az indexelőt úgy készítsük el, hogy ellenőrizze, hogy létező indexekre hivatkoznuk. Ha nem akkor írjon hibaüzenetet a képernyőre.

19. Öröklődés

Öröklődéssel egy már létező típust terjeszthetünk ki vagy bővíthetjük tetszőleges szolgáltatással. A C# csakis egyszeres öröklődést engedélyez, ugyanakkor megengedi több interfész impementálását (interfészekről hamarosan).

Készítsük el az elméleti rész példáját (*Állat-Kutya-Krokodil*) C# nyelven. Az egyszerűség kedvéért hagyjuk ki az *Állat* és *Kutya* közti speciálisabb osztályokat:

```
class Animal
{
    public Animal() { }
}

class Dog : Animal
{
    public Dog() { }
}

class Crocodile : Animal
{
    public Crocodile() { }
}
```

A *Kutya* és *Krokodil* osztályok egyaránt megvalósítják az ősosztály (egyelőre szegényes) funkcionalitását. Bővítsük ki az ősosztályt:

```
class Animal
{
    public Animal() { }
    public void Eat()
    {
        Console.WriteLine("Az allat eszik...");
    }
}
```

Ezt az új metódust az útódosztályok is örökölni fogják, hívjuk is meg valamelyik ősosztályon:

```
Dog d = new Dog();
d.Eat();
```

Az eredmény az lesz, hogy kiírjuk a képernyőre a megadott üzenetet. Honnan tudja vajon a fordító, hogy egy ősosztálybeli metódust kell meghívnia? A referenciatípusok (minden osztály az) speciális módon jelennek meg a memóriában, rendelkeznek többek közt egy ún. metódustáblával, ami mutatja, hogy az egyes metódushívásoknál melyik metódust kell meghívni. Persze ezt is meg kell határozni valahogy, ez nagy vonalakban úgy történik, hogy a fordító a fordítás pillanatában megkapja a metódus nevét és elindul „visszafelé” az osztályhierarchia mentén. A fenti példában a hívó osztály nem rendelkezik *Eat()* nevű metódussal és nem is definiálja át annak a viselkedését (erről hamarosan), ezért az eggyel feljebbi őst kell

megnéznünk. Ez egészen a lehető „legújabb” metódusdefinícióig megy és amikor megtalálja a megfelelő implementációt bejegyzi azt a metódustáblába.

19.1 Konstruktor

Egy leszármazott konstruktorában meghívhatjuk az ősosztály konstruktorát:

```
class BaseClass
{
    protected int data;

    public Base(int _data)
    {
        this.data = _data;
    }
}

class DerivedClass : BaseClass
{
    private int data2;

    public DerivedClass(int d2) : base(d2)
    {
        data2 = data;
    }
}
```

Ezt a *base* függvénnyel tehetjük meg és, ahogy a példában is látszik a paramétereket is átadhatjuk.

19.2 Polimorfizmus

Korábban már beszéltünk arról, hogy az ős és leszármazottak közt *az-egy* reláció áll fent. Ez a gyakorlatban azt jelenti, hogy minden olyan helyen, ahol egy őstípust használunk ott használhatunk leszármazottat is (pl. egy állatkertben állatok vannak, de az állatok helyére (nyílván) behelyettesíthetek egy speciális fajt).
Például gond nélkül írhatom a következőt:

```
Animal a = new Dog();
```

A *new* operátor meghívása után *a* úgy fog viselkedni mint a *Dog* osztály egy példánya, használhatja annak metódusait, adattagjait. Arra azonban figyeljünk, hogy ez visszafelé nem működik, a fordító hibát jelezne.

Abban az esetben ugyanis, ha a fordító engedné a visszafelé konverziót az ún. leszeletelődés (slicing) effektus lépne fel, azaz az adott objektum elveszítené a speciálisabb osztályra jellemző karakterisztikáját. A C++ nyelvben sokszor jelent gondot ez a probléma, mivel ott egy pointeren keresztül megtehető a „lebutítás”. Szerencsére a C# nyelvben ezt megoldották, így nem kell aggódnunk miatta.

19.3 Virtuális metódusok

A virtuális (vagy polimorfikus) metódusok olyan ősosztályok olyan metódusai amelyek viselkedését a leszármazottak átdefiniálhatják.
Virtuális metódust a *virtual* kulcsszó segítségével deklarálhatunk:

```
class Animal
{
    public Animal() { }

    public virtual void Eat()
    {
        Console.WriteLine("Az allat eszik...");
    }
}

class Dog
{
    Public Dog() { }

    public override void Eat()
    {
        Console.WriteLine("A kutya eszik...");
    }
}
```

A leszármazott osztályban az *override* kulcsszóval mondjuk meg a fordítónak, hogy szándékosan hoztunk létre az ősosztályéval azonos nevű metódust és a leszármazott osztályon ezt kívánjuk használni mostantól. Egy *override* –al jelölt metódus automatikusan virtuális is lesz, így az ő leszármazottai is átdefiniálhatják a működését.

Próbáljuk ki az új metódust:

```
Dog d = new Dog();
d.Eat();
```

A kimenet:

```
A kutya eszik...
```

Mi történik vajon a következő esetben:

```
Animal[] aniArray = new Animal[2];

aniArray [0] = new Animal();
aniArray [1] = new Dog();

aniArray [0].Eat();
aniArray [1].Eat();
```

Amit a fordító lát, az az, hogy készítettünk egy *Animal* típusú elemekből álló tömböt és, hogy az elemein meghívtuk az *Eat()* metódust. Csakhogy az *Eat()* egy virtuális

metódus, ráadásul van leszármazottbeli implementációja is, amely átdefinálja az eredeti viselkedést, és ezt explicit jelöltük is az *override* kulcsszóval. Így a fordító el tudja dönteni a futásidejű típust, és ezáltal ütemezi a metódushívásokat. Ez az ún. késői kötés (*late binding*). A kimenet így már nem lehet kétséges:

```
Az állat eszik...
A kutya eszik...
```

Az utódosztály metódusának szignatúrája, visszatérési értéke és láthatósága meg kell egyezzen azzal amit át akarunk definiálni.
Már beszéltünk arról, hogyan épül fel a metódustábla, a fordító megkeresi a legkorábbi implementációt, és most már azt is tudjuk, hogy az első ilyen implementáció egy virtuális metódus lesz, azaz a keresés legkésőbb az első virtuális változatnál megáll.

Tegyük fel, hogy nem ismerjük az ősosztály felületét és a hagyományos módon deklaráljuk az *Eat()* metódust (ugye nem tudjuk, hogy már létezik). Ekkor a program ugyan lefordul, de a fordító figyelmezet minket, hogy eltakarjuk az öröklött metódust. És valóban, ha meghívnánk akkor az új metódus futna le. Ezt a jelenséget árnyékolásnak (*shadow*) nevezik.
Természetesen mi azt szeretnénk, hogy a fordítás hiba nélkül menne végbe, így tájékoztatnunk kell a fordítót, hogy szándékosan takarjuk el az eredeti implementációt. Ezt a *new* kulcsszóval tehetjük meg:

```
public class Animal
{
    public Animal() { }

    public virtual void Eat()
    {
        Console.WriteLine("Az allat eszik...");
    }
}



public class Dog : Animal
{
    public Dog() { }

    public new void Eat()
    {
        Console.WriteLine("A kutya eszik...");
    }
}
```

Ezután a *Dog* utódjai már nem látják az eredeti *Eat()* metódust. Viszont készíthetünk belőle virtuális metódust, amelyet az utódjai már kedvükre használhatnak.
Azaz, a *new* módosítóval ellátott metódus új sort kezd, amikor a fordító felépíti a metódustáblát, vagyis a *new virtual* kulcsszavakkal ellátott metódus lesz az új metódussorozat gyökere.

A következő fejezet lezárt osztályaihoz hasonlóan egy metódust is deklarálhatunk lezártként, ekkor a leszármazottak már nem definiálhatják át a működését:

```
class Animal
{
    public Animal() { }

    public virtual void Eat()
    {
        Console.WriteLine("Egy allat eszik...");
    }
}

class Dog : Animal
{
    public Dog() { }

    public sealed override void Eat()
    {
        Console.WriteLine("Egy kutya eszik...");
    }
}

class Dobermann : Dog
{
    public Dobermann() { }

    public override void Eat() //Ez nem fog lefordulni
    {
        //valamit csinálunk
    }
}
```

Nem jelölhetünk virtuálisnak statikus, absztrakt és *override* –al jelölt tagokat (az utolsó kettő egyébként virtuális is lesz, de ezt nem kell külön jelölni).

19.4 Lezárt osztályok

Egy osztályt lezárhatunk, azaz megtilthatjuk, hogy új osztályt származtassunk belőle:

```
//Ez már egy végleges osztály
public sealed class Dobermann : Dog
{

}

//Ez nem fog lefordulni
public class MyDobermann : Dobermann
{

}
```

A struktúrák (róluk később) alapértelmezetten lezártak.

19.5 Absztrakt osztályok

Egy absztrakt osztályt nem lehet példányosítani. A létrehozásának célja az, hogy közös felületet biztosítsunk a leszármazottainak:

```
abstract class Animal
{
    abstract public void Eat();
}

class Dog : Animal
{
    Animal() { }

    public override void Eat()
    {
        Console.WriteLine("A kutya eszik...");
    }
}
```

Látható, hogy mind az osztály, mind a metódus absztraktként lett deklarálva, ugyanakkor a metódus nem virtuális és nincs definíciója.
Egy absztrakt osztály csak a fordítás közben absztrakt, a lefordított kódban teljesen normális osztályként szerepel, virtuális metódusokkal. A fordító feladata az, hogy betartassa a rá vonatkozó szabályokat. Ezek a szabályok a következőek:

- absztrakt osztályt nem lehet példányosítani
- abszrakt metódusnak nem lehet definíciója
- a leszármazottaknak definiálnia kell az öröklött absztrakt metódusokat.

Absztrakt osztály tartalmazhat nem absztrakt metódusokat is, ezek pont úgy viselkednek, mint a hagyományos nem-virtuális függvények. Az öröklött absztrakt metódusokat az *override* kulcsszó segítségével tudjuk definiálni (hiszen virtuálisak, még ha nem is látszik).
Amennyiben egy osztálynak van legalább egy absztrakt metódusa az osztályt is absztraktként kell jelölni.
Annak ellenére, hogy egy absztrakt osztályt nem példányosíthatunk még lehet konstruktora, mégpedig azért, hogy beállíthassuk vele az adattagokat:

```
abstract class Animal
{
    protected int age;
    public Animal(int _age)
    {
        this.age = _age;
    }
    //Egyéb metódusok...
}
class Dog : Animal
{
    public Dog(int _age) : base(_age) { }
    //Egyéb metódusok...
}
```

Vajon, hogyan működik a következő példában a polimorfizmus elve? :

```
Animal[] animArray = new Animal[2];

animArray[0] = new Dog(2);
animArray[1] = new Crocodile(100);
```

Ennek a kódnak hiba nélkül kell fordulnia, hiszen ténylegesen egyszer sem példányosítottuk az absztrakt ősosztályt. A fordító csak azt fogja megvizsgálni, hogy mi van a *new* operátor jobb oldalán, az alaposztály nem érdekli. Természetesen a következő esetben nem fordulna le:

```
animArray[0] = new Animal(2);
```

Az absztrakt osztályok rendelkeznek néhány gyenge ponttal. Ha egy leszármazottjából származtatok, akkor az új osztálynak nem kell megvalósítania az absztrakt metódusokat, viszont használhatja azokat:

```
public abstract class Animal
{
    public Animal() { }
    public abstract void Eat();
}

public class Dog : Animal
{
    public Dog() { }

    public override void Eat()
    {
        Console.WriteLine("A kutya eszik...");
    }
}

public class Dobermann : Dog
{
    public Dobermann() { }
}

Dobermann d = new Dobermann();
d.Eat(); //Ez működik
```

20. Interfészek

Az interfészek hasonlóak az absztrakt osztályokhoz, abban az értelemben, hogy meghatározzák egy soztály viselkedését, felületét. A nagy különbség a kettő közt az, hogy míg előbbi eleve meghatároz egy osztályhierarchiát, egy interfész nem köthető közvetlenül egy osztályhoz, mindössze előír egy mintát, amit meg kell valósítania az osztálynak. Egy másik előnye az interfészek használatának, hogy míg egy osztálynak csak egy őse lehet, addig bármennyi intefészt megvalósíthat. Ezen felül interfészt használhatunk struktúrák esetében is.
Interfészt a következőképpen deklarálunk:

```
public interface IAnimal
{
    void Eat();
}
```

Az interfész nevét általában nagy *I* betűvel kezdjük. Látható, hogy nincs definíció, csak deklaráció. A megvalósító osztály dolga lesz majd megvalósítani a tagjait. Egy interfész a következőket tartalmazhatja: metódusok, tulajdonságok, indexelők és események. A tagoknak nincs külön láthatóságuk, ehelyett minden tagra az interfész elérhetősége vonatkozik.
A következő forráskód megmutatja, hogyan valósíthatjuk meg a fenti interfészt:

```
public interface IAnimal
{
    void Eat();
}


public class Dog : IAnimal
{
    public Dog() { }

    public void Eat()
    {
        Console.WriteLine("A kutya eszik...");
    }
}
```

Fontos, hogy amennyiben egy másik osztályból is származtatunk, akkor a felsorolásnál az ősosztály nevét kell előrevenni, utána jönnek az interfészek:

```
public interface IFace1 { }
public interface IFace2 { }

class Base { }

class Derived : Base, IFace1, IFace2
{
    /*...*/
}
```

Egy interfészt származtathatunk más interfészekből:

```
public interface IAnimal
{
    void Eat();
}

public interface IDog : IAnimal
{
    void Vau();
}

public class Dog : IAnimal, IDog
{
    public Dog() { }

    public void Eat()
    {
        Console.WriteLine("A kutya eszik...");
    }

    public void Vau()
    {
        Console.WriteLine("Vau-vau...");
    }
}
```

Egy adott interfészt megvalósító objektumot implicit átkonvertálhatjuk az interfész „típusára”:

```
public IEnumerable
{
    /*...*/
}

class EnumClass : IEnumerable
{
    /*...*/
}

IEnumerable ie = new EnumClass(); //Ez működik
```

Az *is* és *as* operátorokkal megtudhatjuk, hogy egy adott osztály megvalósít –e egy interfészt:

```
public interface IMyIf { }

class MyIClass : IMyIf
{

}

MyIClass mic = new MyIClass();
```

```
if(mic is IMyIf)
{
    Console.WriteLine("Az osztaly megvalositja az interfeszt...");
}

IMyIf test = mic as IMyIf;
if(test != null)
{
    Console.WriteLine("Az osztaly megvalositja az interfeszt...");
}
```

Az *is* használata egyértelmű, az *as* pedig *null* –t ad vissza, ha a konverzió sikertelen volt.

20.1 Explicit interfészimplementáció

Ha több interfészt is implementálunk, az névütközéshez is vezethet. Ennek kiküszöbölésére explicit módon megadhatjuk a megvalósítani kívánt funkciót:

```
public interface IOne
{
    void Method();
}

public interface ITwo
{
    void Method();
}

public class MyClass : IOne, ITwo
{
    IOne.Method() { }
    ITwo.Method() { }
}
```

A két *Method()* szignatúrája megegyezik, így valahogy meg kell őket különböztetnünk. Ekkor azonban a függvényhívásnál is problémába ütközünk, ezért konverziót kell végrehajtanunk:

```
MyClass mc = new MyClass();
((IOne)mc).Method();
((ITwo)mc).Method();
```

Egy másik lehetőség:

```
MyClass mc = new MyClass();
IOne iface = (IOne)mc;
iface.Method();
```

Ekkor explicit konverziót hajtunk végre és a megfelelő interfészen hívjuk meg a metódust.

20.2 Virtuális tagok

Egy interfész tagjai alapértelmezés szerint lezártak, de a megvalósításnál jelölhetjük őket virtuálisnak. Ezután az osztály leszármazottjai tetszés szerint módosíthatják a definíciót, a már ismert *override* kulcsszóval:

```
public interface IAnimal
{
    void Eat();
}



public class Dog : IAnimal
{
    public Dog() { }

    public virtual void Eat()
    {
        Console.WriteLine("A kutya eszik...");
    }
}

public class Dobermann : Dog
{
    public Dobermann() { }

    public override void Eat()
    {
        Console.WriteLine("A dobermann eszik...");
    }
}
```

Egy leszármazott újraimplementálhatja az adott interfészt, amennyiben nemcsak az ősnél, de az utódnál is jelöljük a megvalósítást:

```
public class Dobermann : Dog, IAnimal
{
    public Dobermann() { }

    public new void Eat()
    {
        Console.WriteLine("A dobermann sokat eszik...");
    }
}
```

Ez esetben nyílván használnuk kell a *new* kulcsszót annak jelölésére, hogy eltakarjuk az ős megvalósítását.

20.3 Gyakorlati példa 1.

Korábban már találkoztunk a *foreach* ciklussal, és már tudjuk, hogy csak olyan osztályokon képes végigiterálni, amelyek megvalósítják az *IEnumerator* és *IEnumerable* interfészeket. Mindkettő a *System.Collections* névtérben található.

Elsőként nézzük az *IEnumerable* interfészt:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Ez a *foreach* –nek fogja szolgáltatni a megfelelő felületet, ugyanis a ciklus meghívja a metódusát, és annak vissza kell adnia az osztályt *IEnumerator* –ként (ld. implicit konverzió). Ezért kell megvalósítani egyúttal az IEnumerator interfészt is, ami így néz ki:

```
public interface IEnumerator
{
    bool MoveNext();
    void Reset();

    object Current { get; }
}
```

A *MoveNext()* a következő elemre mozgatja a mutatót, ha tudja, de ha a végére ért akkor *false* értékkel tér vissza. A *Reset()* alapértelmezésre állítja a mutatót, azaz *-1* –re. Végül a *Current* (read-only) tulajdonság az aktuális pozicióban lévő elemet adja vissza. Ennek *object* típussal kell visszatérnie, hiszen minden típusra működnie kell (létezik generikus változata is, de erről később).

Használjuk az *Animal* osztályunk egy kissé módosított változatát:

```
public class Animal
{
    private string name;

    public Animal(string _name)
    {
        this.name = _name;
    }
     public Name
    {
        get { return this.name; }
    }
}
```

Most készítsünk egy osztályt, amelyen megvalósítjuk a két interfészt, és ami tartalmaz egy *Animal* objektumokból álló listát:

```
public class AnimalContainer : IEnumerable, IEnumerator
{
    private ArrayList container = new ArrayList();
    private int currPosition = -1;

    public AnimalContainer()
    {
        container.Add(new Animal("Rex"));
        container.Add(new Animal("Rin-Tin-Tin"));
        container.Add(new Animal("Cheetah"));
    }
}
```

Ez persze még nem az egész osztály, felvettünk egy *ArrayList* –et, amiben eltároljuk az objektumokat, illetve deklaráltunk egy egész számot, ami az aktuális pozicíót tárolja el és kezdőértékéül *-1* –et adtunk (ld. *Reset()*).
A konstruktorban látható, hogy rögtön a *new* –al hoztunk létre új objektumokat, nem pedig külön tettük ezt.
Készítsük el az *IEnumerator* által igényelt metódusokat:

```
public bool MoveNext()
{
    return (++position < container.Count)
}

public object Current
{
    get { return container[currPosition]; }
}

public void Reset()
{
    currPosition = -1;
}
```

Végül az *IEnumerable* interfészt valósítjuk meg:

```
public IEnumerator GetEnumerator()
{
    return (IEnumerator)this;
}
```

Ezután használhatjuk is az osztályt:

```
AnimalContainer ac = new AnimalContainer();

foreach(Animal a in ac)
{
    Console.WriteLine(a.Name);
}
```

20.4 Gyakorlati példa 2.

A második gyakorlati példánkban az *IComparable* interfészt fogjuk megvalósítani, amelyre gyakran van szükségünk. Ez az interfész általában olyan adatszerkezeteknél követelmény amelyek az elemeiken megvalósítanak valamilyen rendezést. A generikus *List* típusmak is van rendező metódusa, amely ezzel a metódusal dolgozik. Az *IComparable* egyetlen metódussal a *CompareTo()* –val rendelkezik, amely egy *object* típust kap paraméteréül:

```
class ComparableClass : IComparable
{
    int value;

    public ComparableClass(int val)
    {
        this.value = val;
    }

    public int Value
    {
        get { return value; }
    }

    public int CompareTo(object o)
    {
        if(o is ComparableClass)
        {
            ComparableClass c = (ComparableClass)o;
            return value.CompareTo(c.Value);
        }
        else throw(new Exception("Nem megfelelo objektum..."));
    }
}
```

Az osztályban a beépített típusok *CompareTo()* metódusát használtuk, hiszen ők mind megvalósítják ezt az interfészt. Ez a metódus *-1* –et ad vissza ha a hívó fél kisebb, *0* –át, ha egyenlő és *1* –et ha nagyobb. A használata:

```
List<ComparableClass> list = new List<ComparableClass>();
Random r = new Random();

for(int i = 0;i < 10;++i)
{
    list.Add(new ComparableClass(r.Next(1000)));
}

foreach(ComparableClass c in list)
{
    Console.WriteLine(c.Value);
}


Console.WriteLine("A rendezett lista:");

list.Sort();
```

```
foreach(ComparableClass c in list)
{
    Console.WriteLine(c.Value);
}
```

Hasonló feladatot lát el, de jóval rugalmasabb az *IComparer* interfész. A *List* rendezésénél megadhatunk egy összehasonlító osztályt is, amely megvalósítja az *IComparer* interfészt. Most is csak egy metódust kell elkészítenünk, ez a *Compare()* amely két *object* típust vár paramétereként:

```
class ComparableClassComparer : IComparer
{
    public int Compare(object x, object y)
    {
        if(x is ComparableClass && y is ComparableClass)
        {
            ComparableClass _x = (ComparableClass)x;
            ComparableClass _y = (ComparableClass)y;

            return _x.CompareTo(_y);
        }
        else throw(new Exception("Nem megfelelo parameter...");
    }
}
```

Ezután a következőképpen rendezhetjük a listát:

```
list.Sort(new ComparableClassComparer());
```

Az *IComparer* előnye, hogy nem kötődik szorosan az osztályhoz (akár anélkül is megírhatjuk, hogy ismernénk a belső szerkezetét), így többféle megvalósítás is lehetséges.

21. Operátor túlterhelés

Nyílván szeretnénk, hogy az általunk készített típusok hasonló funkcionalitással rendelkezzenek mint a beépített típusok (*int*, *string*, stb…).
Vegyük pl. azt a példát, amikor egy mátrix típust valósítunk meg. Jó lenne, ha az összeadás, kivonás, szorzás, stb. műveleteket úgy tudnánk végrehajtani, mint egy egész szám esetében, nem pedig metódushívásokkal. Szerencsére a C# ezt is lehetővé teszi számunkra, ugyanis engedi az operátortúlterhelést, vagyis egy adott operátort tetszés szerinti funkcióval ruházhatunk fel az osztályunkra vonatkoztatva.

```
Matrix m1 = new Matrix();
Matrix m2 = new Matrix();

//ehelyett
Matrix m3 = m1.Add(m2);

//írhatjuk ezt
Matrix m4 = m1 + m2;
```

A túlterhelhető operátorok listája:

+(unáris)	-(unáris)	!	~	++
--	+	-	*	/
%	&	|	^	<<
>>	==	!=	>	<
>=	<=			

A C# nyelvben a paraméterek statikus metódusok, paramétereik az operandusok, visszatérési értékük pedig az eredmény. Egy egyszerű példa:

```
class MyInt
{
    int value;

    public MyInt(int _value)
    {
        this.value = _value;
    }

    public int Value
    {
        get { return value; }
    }

    public static MyInt operator+(MyInt lhs, MyInt rhs)
    {
        return new MyInt(lhs.Value + rhs.Value);
    }
}
```

A '*+*' operátort működését fogalmaztuk át. A paraméterek (operandusok) nevei konvenció szerint lhs (*left-hand-side*) és rhs (*right-hand-side*), utalva a jobb és baloldali operandusra.
Tehát ha a következőt írom:

```
MyInt m1 = new MyInt(10);
MyInt m2 = new MyInt(11);
MyInt sum = m1 + m2;
```

Mivel definiáltunk az osztályunkon egy saját operátort így a fordító tudni fogja, hogy azt használja és átalakítja az utolsó sort a következőre:

```
MyOnt sum = MyInt.operator+(m1, m2);
```

A következőkben megnézünk néhány operátort, végül pedig megvalósítjuk a mátrix osztályt és operátorait.

21.1 Egyenlőség operátorok

A C# nyelv megfogalmaz néhány szabályt az operátortúlterheléssel kapcsolatban. Ezek egyike az, hogy ha túlterheljük az egyenlőség operátort (*==*) akkor definiálnunk kell a nem-egyenlő (*!=*) operátort is:

```
class MyEqual
{
    int value;

    public MyEqual(int _value)
    {
        this.value = _value;
    }

    public int Value
    {
        get { return value; }
    }

    static public bool operator==(MyEqual lhs, MyEqual rhs)
    {
        return (lhs.Value == rhs.Value);
    }

    static public bool operator!=(MyEqual lhs, MyEqual rhs)
    {
        return !(lhs == rhs);
    }
}
```

A nem-egyenlő operátor esetében a saját egyenlőség operátort használtuk fel (a megvalósítás elve nem feltétlenül világos, elsőként megvizsgáljuk, hogy a két elem egyenlő –e, de mi a nem-egyenlőségre vagyunk kíváncsiak, ezért tagadjuk az eredményt, ami pontosan ezt a választ adja meg).

Ezekhez az operátorokhoz tartozik az *object* típustól örökölt virtuális *Equals()* metódus is, ami a CLS kompatibilitást hívatott megőrizni, erről később még lesz szó. A fenti esetben ezt a metódust is illik megvalósítani. Az *Equals* azonban egy kicsit különbözik, ő egyetlen *object* típusú paramétert vár, ezért meg kell majd győződnünk arról, hogy valóban a saját objektumunkkal van –e dolgunk:

```
public override bool Equals(object o)
{
    if(!(o is MyEqual))
    {
        return false;
    }

    return this == (MyEqual)o;
}
```

Az *is* operátorral ellenőrizzük a futásidejű típust. Mivel ez egy példány tag ezért a *this* –t használjuk az objektum jelölésére, amin meghívtuk a metódust.

Az *Equals()* mellett illik megvalósítani a szintén az *object* –től öröklött *GetHashCode()* metódust is, így az osztály használható lesz gyűjteményekkel és a *HashTable* típussal is. A legegyszerűbb implementáció visszad egy számot az adattag(ok)ból számolva (pl.: hatványozás, biteltolás, stb…):

```
class MyHash
{
    int value;

    public MyHash(int _value)
    {
        this.value = _value;
    }

    public override int GetHashCode()
    {
        return value;
    }
}
```

21.2 Relációs operátorok

Hasonlóan a logikai operátorokhoz a relációs operátorokat is csak párban lehet elkészíteni, vagyis (<, >) és (<=, >=).
Ebben az esetben az *IComparable* és *IComparable<T>* interfészek megvalósítása is szükséges lehet a különböző gyűjteményekkel való együttműködés érdekében. Ezekről az előző, interfészekkel foglalkozó fejezet nyújt bővebb információt.

21.3 Konverziós operátorok

A C# a szűkebbről tágabbra konverziókat implicit módon (azaz különösebb jelölés nélkül), míg a tágabbról szűkebbre kovertálást explicite (ezt jelölnünk kell) végzi. Természetesen szeretnénk, ha a saját típusunk ilyesmire is képes legyen, és bizony

erre van operátor, amit túlterhelhetünk. Ezeknél az operátoroknál az *implicit* illetve *explicit* kulcsszavakkal fogjuk jelölni a konverzió típusát:

```
class MyConversion
{
    int value;

    public MyConversion(int _value)
    {
        this.intValue = _value;
    }

    public int Value
    {
        get { return value; }
    }

    public static implicit operator MyConversion(int rhs)
    {
        return new MyConversion(rhs);
    }

    public static explicit operator MyConversion(long rhs)
    {
        return new MyConversion((int)rhs);
    }
}
```

Ezt most így használhatjuk:

```
int x = 10;
long y = 12;
MyConversion mc1 = x; //implicit konverzió
MyConversion mc2 = (MyConversion)y; //explicit konverzió
```

Fontos, hogy a konverziós operátorok mindig statikusak.

21.4 Kompatibilitás más nyelvekkel

Mivel nem minden nyelv teszi lehetővé az operátortúlterhelést ezért a CLS javasolja, hogy készítsük el a hagyományos változatot is:

```
class Operators
{
    private int value;

    public Operators(int _value)
    {
        this.value = _value;
    }
    public int Value
    {
        get { return value; }
    }
```

```
    static public Operators operator+(Operators lhs, Operators rhs)
    {
        return new Operators(lhs.Value + rhs.Value);
    }

    static public Operators Add(Operators lhs, Operators rhs)
    {
        return new Operators(lhs.Value + rhs.Value);
    }
}
```

21.5 Gyakorlati példa

Most pedig elkészítjük a mátrix típust:

```
class Matrix
{
    int[,] matrix;

    public Matrix(int n, int m)
    {
        matrix = new int[n, m];
    }

    public int N
    {
        get { return matrix.GetLength(0); }
    }

    public int M
    {
        get { return matrix.GetLength(1); }
    }

    public int this[int idxn, int idxm]
    {
        get { return matrix[idxn, idxm]; }
        set { matrix[idxn, idxm] = value; }
    }

    static public Matrix operator+(Matrix lhs, Matrix rhs)
    {
        if(lhs.N != rhs.N || lhs.M != rhs.M) return null;

        Matrix result = new Matrix(lhs.N, lhs.M);

        for(int i = 0;i < lhs.N;++i)
        {
            for(int j = 0;j < lhs.M;++j)
            {
                result[i, j] = lhs[i, j] + rhs[i, j];
            }
        }
        return result;
    }
}
```

Mátrixokat úgy adunk össze, hogy az azonos indexeken lévő értékeket öszeadjuk:

```
123    145    1+1 2+4 3+5
456 + 532 = (stb…)
789    211
```

Az összeadás műveletet csakis azonos nagyságú dimenziók mellet lehet elvégezni (3x3 –as mátrixhoz nem lehet hozzáadni egy 4x4 –eset). Ezt ellenőriztük is az operátor megvalósításánál.

Most csak az összeadás műveletet valósítottuk meg, a többi a kedves olvasóra vár. Plusz feladatként indexellenőrzést is lehet végezni az indexelőnél.

22. Struktúrák

A struktúrák hasonlóak az osztályokhoz, viszont van néhány különbség:

- Egy osztály referenciatípus míg egy struktúra értéktípus
- Struktúrából nem lehet származtatni ill. egy struktúra nem származhat más struktúrából (kivétel a *System.Object*).
- Egy struktúrának nem lehet paraméter nélküli (default) konstruktora
- Nem használhatnak finalizer –t.

Minden struktúra közvetlenül a *System.ValueType* típusból származik, amely pedig a *System.Object* –ből.
Struktúrát általában akkor használunk, ha adatok egyszerű összeségével kell dolgoznunk, de nincs szükségünk egy osztály minden szolgáltatására. Struktúrák használatának van még egy előnye, ugyanis példányosításnál nem kell helyet foglalni a halomban, így sok objektum esetében hatékonyabb a használata.
Minden struktúra alapértelmezetten rendelkezik paraméternélküli konstruktorral, amelyet nem rejthetünk el. Ez a konstruktor a struktúra minden mezőjét nullértékkel tölti fel:

```
struct MyStruct
{
    public int x;
}

MyStruct ms = new MyStruct(); // x == 0
Console.WriteLine(ms.x);
```

Nem kötelező használni a *new* –t, ekkor automatikusan az alapértelmezett konstruktor hívódik meg.

```
MyStruct x; //ez is működik
```

Készíthetünk saját konstruktort, de ekkor minden mező értékadásáról gondoskodnunk kell. Egy struktúra mezőit nem inicializálhatjuk:

```
struct MyStruct
{
    int x = 10; //ez hiba
    int y;

    public MyStruct(int _x, int _y)
    {
        x = _x; //hiba, y nem kap értéket
    }
}
```

Egy struktúra tartalmazhat referenciatípust, ekkor a verembe (az értéktípusok memóriahelyére) egy referencia kerül amely a referenciatípusra hivatkozik a halomban. Amennyire lehet kerüljük ezt a megoldást, struktúrák esetén csakis értéktípusokkal dolgozzunk.

23. Kivételkezelés

Nyílván vannak olyan esetek, amikor az alkalmazásunk nem úgy fog működni, ahogy elképzeltük. Az ilyen „abnormális” működés kezelésére találták ki a kivételkezelést. Amikor az alkalmazásunk „rossz” állapotba kerül, akkor egy ún. kivételt fog dobni, ilyennel már találkoztunk a tömböknél, amikor túlindexeltünk:

```
using System;

class Program
{
    static public void Main()
    {
        int[] x = new int[2];
        x[2] = 10;
        Console.ReadKey();
    }
}
```

Itt az utolsó érvényes index az 1 lenne, így kivételt kapunk, mégpedig egy *System.IndexOutOfRangeException* –t. Ezután a program leáll. Természetesen mi azt szeretnénk, hogy valahogy kijavíthassuk ezt a hibát, ezért el fogjuk kapni a kivételt. Ehhez a művelethez három dologra van szükségünk: kijelölni azt a programrészt ami dobhat kivételt, elkapni azt és végül kezeljük a hibát:

```
using System;

class Program
{
    static public void Main()
    {
        int[] x = new int[2];
        try
        {
            x[2] = 10;
        }
        catch(System.IndexOutOfRangeException e)
        {
            Console.WriteLine("A hiba: {0}", e.Message);
        }
        Console.ReadKey();
    }
}
```

A *try* blokk jelöli ki a lehetséges hibaforrást, a *catch* pedig elkapja a megfelelő kivételt (arra figyeljünk, hogy ezek is blokkok, azaz a blokkon belül deklarált változók a blokkon kívűl nem láthatóak). A fenti programra a következő lesz a kimenet:

```
A hiba: Index was outside the bounds of the array.
```

Kivételt a *throw* utasítással dobhatunk:

```
using System;

class Program
{
    static public void Main()
    {
        try
        {
            throw(new System.Exception("Exception..."));
        }
        catch(System.Exception e)
        {
            Console.WriteLine("A hiba: {0}", e.Message);
        }
    }
}
```

A C++ -tól eltérően itt példányosítanunk kell a kivételt.

A *catch* –nek nem kötelező megadni a kivétel típusát, ekkor minden kivételt elkap:

```
try
{
    throw(new System.Exception("Exception..."));
}
catch
{
    Console.WriteLine("Kivétel történt…");
}
```

23.1 Kivétel hierarchia

Amikor kivétel dobódik, akkor a vezérlést az első alkalmas *catch* blokk veszi át. Az összes kivétel ugyanis a *System.Exception* osztályból származik, így ha ezt adjuk meg a *catch* –nél, akkor az összes lehetséges kivételt el fogjuk kapni vele:

```
catch(System.Exception e)
{
    Console.WriteLine("A hiba: {0}", e.Message);
}
```

Egyszerre több *catch* is állhat egymás után, de ha van olyan amelyik az ős kivételt kapja el, akkor a program csak akkor fog lefordulni, ha az az utolsó helyen áll, hiszen az mindent elkapna.

```
catch(System.IndexOutOfRangeException e)
{
    Console.WriteLine("A hiba: {0}", e.Message);
}
catch(System.Exception e)
{
    Console.WriteLine("A hiba: {0}", e.Message);
}
```

23.2 Saját kivétel készítése

Mi magunk is csinálhatunk kivételt, a *System.Exception* –ból származtatva:

```
public class MyException : System.Exception
{
    public MyException() { }

    public MyException(string message) : base(message) { }

    public MyException(string message, Exception inner)
    : base(message, inner) { }
}
```

Az utolsó konstruktorról hamarosan lesz szó. Most így használhatjuk:

```
try
{
    throw(new MyException("Sajat kivetel...");
}
catch(MyException e)
{
    Console.WriteLine("A hiba: {0}", e.Message);
}
```

23.3 Kivételek továbbadása

Egy kivételt az elkapása után ismét eldobhatunk. Ez hasznos olyan esetekben, amikor feljegyzést akarunk készíteni illetve, ha egy specifikusabb kivételkezelőnek akarjuk átadni a kivételt:

```
try
{

}
catch(System.ArgumentException e)
{
    throw; //továbbadjuk
    throw(new System.ArgumentNullException()); //vagy egy újat dobunk
}
```

Természetesen a fenti példában csak az egyik *throw* szerepelhetne „legálisan”, ebben a formában nem fog lefordulni.

23.4 Finally blokk

A kivételkezelés egy problémája, hogy a kivétel keletkezése után az éppen végrehajtott programrész futása megszakad, így előfordulhat, hogy nem szabadulnak fel időben az erőforrások (megnyitott file, hálózati kapcsolat, stb), illetve objektumok olyan formában maradnak meg a memóriában, amely hibát okozhat.

Megoldást a *finally* – blokk használata jelent, amely függetlenül attól, hogy történt –e kivétel mindig végrehajtódik:

```
int x = 10;

try
{
      Console.WriteLine("X erteke a kivetel elott: {0}", x);
      throw(new Exception());
}
catch(Exception)
{
      Console.WriteLine("Kivetel keletkezett...");
}
finally
{
      x = 11;
}

Console.WriteLine("X erteke a kivetel utan: {0}", x);
```

A kimenet a következő lesz:

```
X erteke a kivetel elott: 10
Kivetel keletkezett...
X erteke a kivetel utan: 11
```

„Valódi” erőforrások kezelésekor kényelmesebb a *using* – blokk használata:

```
using(StreamReader reader = File.OpenText("text.txt"))
{
}
```

Ez pontosan megegyezik ezzel:

```
StreamReader reader = File.OpenText("text.txt");
try
{
}
finally
{
      if(reader != null)
      {
            ((IDisposable)reader).Dispose();
      }
}
```

Filekezeléssel egy későbbi fejezet foglalkozik majd.

24. Generikusok

Az objektum – orientált programozás egyik alapköve a kódújrafelhasználás, vagyis, hogy egy adott kódrészletet elég általánosra írjunk meg ahhoz, hogy minél többször felhasználhassuk. Ennek megvalósítására két eszköz áll rendelkezésünkre, az egyik az öröklődés, a másik pedig jelen fejezet tárgya a generikus típusok, metódusok.

24.1 Generikus metódusok

Vegyük a következő metódust:

```
static public void Swap(ref int x, ref int y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

Ha szeretnénk, hogy ez a metódus más típusokkal is működjön, akkor bizony sokat kell gépelnünk. Kivéve, ha írunk egy generikus metódust:

```
static public void Swap<T>(ref T x, ref T y)
{
    T tmp = x;
    x = y;
    y = tmp;
}
```

A T betű fogja jelképezni az aktuális típust (lehet más nevet is adni neki, eredetileg a Template szóból jött), ezt generikus paraméternek hívják. Generikus paramétere csakis osztálynak vagy metódusnak lehet és ebből többet is használhatnak. Ezután a metódust a hagyományos úton használhatjuk, a fordító felismeri, hogy melyik típust használjuk (ezt megadhatjuk mi magunk is explicite):

```
int x = 10;
int y = 8;
Swap(ref x, ref y);
//vagy
Swap<int>(ref x, ref y);
```

A C# generikusai hasonlítanak a C++ sablonjaira, de annál kevésbé hatékonyabbak, cserébe sokkal biztonságosabb a használatuk. Két fontos különbség van a kettő közt, míg a C++ fordítási időben készíti el a specializált metódusokat/osztályokat, addig a C# ezt a műveletet futási időben végzi el. A másik eltérés az elsőből következik, mivel a C++ fordításkor ki tudja szűrni azokat az eseteket, amelyek hibásak, pl. összeadunk két típust a sablonmetódusban, amelyeken nincs értelmezve összeadás. A C# ezzel szemben kénytelen az ilyen problémákat megelőzni, a fordító csakis olyan műveletek elvégzését fogja engedélyezni, amelyek mindenképpen működni fognak. A következő példa nem fog lefordulni:

```
static public T Sum<T>(T x, T y)
{
    return x + y;
}
```

Fordításkor csak azt tudja ellenőrizni, hogy létező típust adtunk –e meg, így nem tudhatja a fordító, hogy sikeres lesz –e a végrehajtás, ezért a fenti kód az összeadás miatt (nem feltétlenül valósítja meg minden típus) „rossz”.

24.2 Generikus osztályok

Képzeljük el, hogy azt a feladatot kaptunk, hogy készítsünk egy verem típust, amely bármely típusra alkalmazható. Azt is képzeljük el, hogy még nem hallottunk generikusokról. Így a legkézenfekvőbb megoldás egy *object* típusú tömb lesz:

```
class Stack
{
    object[] t;
    int pointer;
    readonly int size;

    public Stack(int capacity)
    {
        t = new object[capacity];
        size = capacity;
        pointer = 0;
    }

    public void Push(object _in)
    {
        if(pointer >= size)
        {
            throw(new StackOverflowException("Tele van..."));
        }

        t[pointer] = _in;
        ++pointer;
    }

    public object Pop()
    {
        --pointer;
        if(pointer >= 0)
        {
            return t[pointer];
        }

        pointer = 0;
        throw(new InvalidOperationException("Ures..."));
    }
}
```

Ezt most a következőképpen használhatjuk:

```
Stack s = new Stack(10);

for(int i = 0;i < 10;++i)
{
    s.Push(i);
}

for(int i = 0;i < 10;++i)
{
    Console.WriteLine(Console.WriteLine(int.Parse(s.Pop())));
}
```

Működni működik, de se nem hatékony se nem kényelmes. A hatékonyság az érték/referenciatípusok miatt csökken jelentősen (ld. boxing/unboxing), a kényelem pedig amiatt, hogy mindig figyelni kell épp milyen típussal dolgozunk, nehogy olyan kasztolással éljünk ami kivételt dob.
Ezeket a problémákat könnyen kiküszöbölhetjük, ha genrikus osztályt készítünk:

```
class Stack<T>
{
    T[] t;
    int pointer;
    readonly int size;

    public Stack(int capacity)
    {
        t = new T[capacity];
        size = capacity;
        pointer = 0;
    }

    public void Push(T _in)
    {
        if(pointer >= size)
        {
            throw(new StackOverflowException("Tele van..."));
        }
        t[pointer] = _in;
        ++pointer;
    }

    public object Pop()
    {
        --pointer;
        if(pointer >= 0)
        {
            return t[pointer];
        }

        pointer = 0;
        throw(new InvalidOperationException("Ures..."));
    }
}
```

Ezután akármelyik típuson könnyen használhatjuk:

```
Stack<int> s = new Stack<int>(10);
Stack<string> c = new Stack<string>(10);

for(int i = 0;i < 10;++i)
{
     s.Push(i);
     c.Push(i.ToString());
}

for(int i = 0;i < 10;++i)
{
     Console.WriteLine("{0}, {1}", s.Pop(), c.Pop());
}
```

A generikus típusok nem kovariánsak, azaz egy generikus példányt nem kasztolhatunk olyan típusra, amelyre normális esetben igen:

```
class Base
{

}

class Derived : Base
{

}

Stack<Derived> s = new Stack<Derived>(10);
Stack<Base> b = s; //ez nem működik
```

24.3 Generikus megszorítások

Alapértelmezetten egy generikus paraméter bármely típust jelképezheti. A deklarációnál azonban kiköthetünk megszorításokat a paraméterre. A megszorításokat a *where* kulcsszóval vezetjük be:

```
where T : alaposztály
where T : interfész
where T : osztály
where T : struktúra
where T : new() //alapértelmezett konstruktor
where T : U
```

Egy példa:

```
public interface IFace { }
public class BaseClass { }

class NewClass<T> where T : BaseClass, IFace, new()
{ }
```

A *NewClass* osztály csak akkor példányosítható, ha a megadott típusa a *BaseClass* osztályból származik, megvalósítja az *IFace* interfészt és rendelkezik alapértelmezett konstruktorral:

```
class GoodClass : BaseClass, IFace
{
    public GoodClass() { }
}

NewClass<GoodClass> nc = new NewClass<GoodClass>();
```

Az osztály és struktúra megszorítás azt jelenti, hogy a paraméternek osztálynak/struktúrának kell lennie.

24.4 Öröklődés

Generikus osztályból származtathatunk is, ekkor vagy az ősosztály egy specializált változatából származtatunk, vagy a nyers generikus osztályból:

```
class Base<T>
{

}

class Derived<T> : Base<T>
{

}

//vagy

class IntDerived : Base<int>
{

}
```

24.5 Statikus tagok

Generikus típusok esetében minden típushoz külön statikus tag tartozik:

```
class MyClass<T>
{
    public static int data;
}

MyClass<int>.data = 10;
MyClass<string>.data = 20;
Console.WriteLine(MyClass<int>.data); //10
Console.WriteLine(MyClass<string>.data); //20
```

24.6 Generikus gyűjtemények

A C# 2.0 bevezetett néhány hasznos generikus adatszerkezetet, többek közt listát és vermet. Ezeket a típusokat a tömbökhöz hasonlóan használhatjuk. A következőkben megviszgálunk ezek közül néhányat. Ezek a szerkezetek a *System.Collections.Generic* névtérben találhatóak:

```
List<int> list = new List<int>();

for(int i = 0;i < 10;++i)
{
    list.Add(i);
}

foreach(int i in list)
{
    Console.WriteLine(i);
}
```

Az *Add()* metódus a lista végéhez adja hozzá a praméterként megadott elemet, hasonlóan az *ArrayList* –hez (tulajdonképpen a *List<>* az *ArrayList* generikus változata). Használhatjuk rajta az indexelő operátort is.

A *SortedList* kulcs – érték párokat tárol el és a kulcs alapján rendezi is őket:

```
SortedList<string, int> slist = new SortedList<string, int>();
slist.Add("elso", 1);
slist.Add("masodik", 2);
slist.Add("harmadik", 3);
```

A lista elemei tulajdonképpen nem a megadott értékek, hanem a kulcs – érték párokat reprezentáló *KeyValuePair<>* objektumok. A lista elemeinek eléréséhez is használhatjuk ezeket:

```
foreach(KeyValuePair<string, int> kv in slist)
{
    Console.WriteLine("Kulcs: {0}, Ertek: {1}", kv.Key, kv.Value);
}
```

A lista kulcsai csakis olyan típusok lehetnek, amelyek megvalósítják az *IComparable* interfészt, hiszen ez alapján történik a rendezés. Ha ez nem igaz, akkor mi magunk is definiálhatunk ilyet, részletekért ld. az Interfészek fejezetet.
A *SortedList* rendezetlen párja a *Dictionary<>:*

```
Dictionary<string, int> dict = new Dictionary<string, int>();
dict.Add("elso", 1);
dict.Add("masodik", 2);

foreach(KeyValuePair<string, int> kv in dict)
{
    Console.WriteLine("Key: {0}, Value: {1}", kv.Key, kv.Value);
}
```

24.7 Generikus interfészek

A letöbb hagyományos interfésznek létezik generikus változata is. Például az *IEnumerable* és *IEnumerator* is ilyen:

```
class MyClass<T> : IEnumerable<T>, IEnumerator<T>
{

}
```

Ekkor a megvalósítás teljesen ugyanúgy működik mint a hagyományos esetben, csak épp használnunk kell a generikus paraméter(eke)t.

A generikus adatszerkezetek a generikus *ICollection*, *IList* és *IDictionary* interfészeken alapulnak, így ezeket megvalósítva akár mi magunk is létrehozhatunk ilyet.

25. Delegate -ek

A delegate –ek olyan típusok, amelyek egy metódusra hivatkoznak. Egy delegate deklarációjánál megadjuk, hogy milyen szignatúrával rendelkező metódusokra mutathat:

```
delegate int MyDelegate(int x);
```

Ez a delegate olyan metódusra mutathat amelynek visszatérési értéke *int* típusú és egyetlen *int* paramétere van:

```
static public int GoodMethod(int x)
{
    return (x * x);
}
```

A használata:

```
MyDelegate delgt = GoodMethod;
int result = delgt(10); //result == 100
```

A delegate példányosítását a hagyományos úton is megtehetjük:

```
MyDelegate delgt = new MyDelegate(GoodMethod);
```

A delegate –ek legnagyobb haszna, hogy nem kell előre megadott metódusokat használnunk, ehelyett dinamikusan adhatjuk meg az elvégzendő műveletet:

```
class ArrayTransform
{
    public delegate int Transformer(int _input);

    int[] array;

    public ArrayTransform(int length)
    {
        array = new int[length];
        Random r = new Random();
        for(int i = 0;i < length;++i)
        {
            array[i] = r.Next(1000);
        }
    }

    public void Transform(Transformer t)
    {
        for(int i = 0;i < array.Length;++i)
        {
            array[i] = t(array[i]);
        }
    }
```

```
    public void Print()
    {
        foreach(int i in array)
        {
            Console.Write("{0}, ", i);
        }
        Console.WriteLine();
    }
}
```

A *Transform()* metódus egy delegate –et kap paraméteréül, ami elvégzi a változtatásokat a tömbön. Pl.:

```
class Program
{
    static public int Square(int _in)
    {
        return (_in * _in);
    }

    static public void Main()
    {
        ArrayTransform at = new ArrayTransform(10);

        at.Print();

        at.Transform(Square);

        at.Print();

        Console.ReadKey();
    }
}
```

Két delegate nem egyenlő, még akkor sem ha a szignatúrájuk megegyezik:

```
delegate void DelOne();
delegate void DelTwo();

void Method() { }

DelOne dlo = Method;
DelTwo dlt = dlo; //hiba
```

25.1 Többszörös delegate -ek

Egy delegate nem csak egy metódusra hivatkozhat:

```
delegate void MyDelegate();

void MethodOne()
{
    Console.Write("Method One…");
}
```

```
void MethodTwo()
{
        Console.Write("Method Two…");
}

MyDelegate d = MethodOne;
d += MethodTwo;
d();
```

Most *d* hívásakor mindkét metódus meghívódik. Egy delegate –ből el is vehetünk metódust:

```
d -= MethodOne();
```

25.2 Paraméter és visszatérési érték

Egy delegate használhat generikus paramétereket is:

```
delegate T GenericDelegate<T>(T param);

int Square(int _in)
{
     return (_in * _in);
}

GenericDelegate<int> gd = Square;
```

Egy delegate –nek átadott metódus paraméterei lehetnek olyan típusok, amelyek az eredeti paraméternél általánosabbak:

```
class MyClass { }

delegate void MyDelegate(MyClass mc);

void Method(object i) { }

MyDelegate md = Method();
```

Ez az ún. kontravariáns (*contravariant*) viselkedés. Ennek a fordítottja igaz a visszatérési értékre, azaz az átadott metódus visszatérési értéke lehet specifikusabb az eredetinél:

```
class Base { }

class Derived { }

delegate Base MyDelegate();

Derived Method() { }

MyDelegate md = Method();
```

Ezt kovariáns (*covariant*) viselkedésnek nevezzük.

25.3 Névtelen metódusok

Egy delegate –nek nem kötelező létező metódust megadnunk, akár explicit módon is megadhatjuk azt:

```
delegate int MyDelegate(int x);

MyDelegate md = delegate(int x)
{
    return  (x * x);
};
```

26. Események

Egy osztály eseményeket használhat, hogy értesítsen más osztályokat, hogy valami történt (felhasználói aktivitás, stb…). Gyakran találkozunk majd eseményekkel a grafikus felületű alkalmazásokkal foglalkozó fejezetekben.
Egy esemény egy olyan metódust (vagy metódusokat) fog meghívni, amelyekre egy delegate hivatkozik. Ezeket a metódusokat eseménykezelőknek nevezzük.
Az eseményekhez rendelt delegate –eknek konvenció szerint (ettől eltérhetünk, de a Framework eseményei mind ilyenek) két paramétere van, az első az az objektum, amely kiváltotta az eseményt, a második pedig az eseményhez kapcsolódó információk. A második paraméter csakis olyan típus lehet, amely az *EventArgs* osztályból származik.
A következő példában egy listához kapcsolunk eseményt, amely akkor hajtódik végre, amikor hozzáadunk a listához egy elemet:

```
using System;
using System.Collections.Generic;

public delegate void ChangedEventHandler(object sender, EventArgs e);

class MyListWithEvent
{
    private List<int> list = null;


    public event ChangedEventHandler Changed;

    public MyListWithEvent()
    {
        list = new List<int>();
    }

    public void Add(int value)
    {
        list.Add(value);
        OnChanged(EventArgs.Empty);
    }

    protected virtual void OnChanged(EventArgs e)
    {
        if(Changed != null)
        {
            Changed(this, e);
        }
    }
}



class ListListener
{
    private MyListWithEvent list = null;

    public ListListener(MyListWithEvent l)
```

```
    {
        list = l;
        list.Changed += new ChangedEventHandler(ListChanged);
    }

    public void Add(int value)
    {
        list.Add(value);
    }

    private void ListChanged(object sender, EventArgs e)
    {
        Console.WriteLine("A lista megváltozott...");
    }
}

class Program
{
    static public void Main()
    {
        MyListWithEvent list = new MyListWithEvent();
        ListListener llist = new ListListener(list);
        llist.Add(10);
        Console.ReadKey();
    }
}
```

Az események segítségével megvalósíthatjuk (és a .Net meg is valósítja) az Observer (Figyelő) tervezési mintát, ennek lényege nagy vonalakban, hogy ún. kliensek felíratkozhatnak a kiszolgálóra, illetve annak egyes eseményeire. Amikor a kiszolgálón történik valami, amely megváltoztatja az állapotát, akkor értesíti a klienseit, akik az érkező információkat felhasználják a saját adataik módosítására.

Tervezési mintákról a következő oldalon olvashatunk többet:

http://www.dofactory.com/Patterns/Patterns.aspx#list

26.1 Gyakorló feladatok

1. Készítsük el a hőmérő osztályt! Tartalmazzon egy eseményt, amely egy bizonyos hőmérséklet alatt vagy fölött riaszt (ezt pl. egy ciklussal tesztelhetjük, véletlenszámokat megadva az aktuális hőmérsékletnek).

2. Készítsünk tőzsdeszimulátort! Hozzunk létre kliens osztályokat is, amelyek a tőzsdét jelképező osztály egy adott eseményére irattkozhatnak fel (vagyis bizonyos részvények árfolyamát figyelik).

27. Lambda kifejezések

A C# 3.0 bevezeti a lambda kifejezéseket. Egy lambda kifejezés gyakorlatilag megfelel egy névtelen metódusnak.
Minden lambda kifejezés tartalmazza az ún. lambda operátort (=>), ennek jelentése nagyjából annyi, hogy „legyen”. Az operátor bal oldalán a bemenő változók, jobb oldalán a kifejezés áll.
Mivel névtelen metódus ezért egy lambda kifejezés állhat egy delegate értékadásában:

```
delegate int MyDelegate(int _in);

MyDelegate d = x => (x * x);

Console.WriteLine(d(2)); //4
```

Vagyis *x legyen x*x*, azaz a bemenő paraméter négyzetét adjuk vissza. Természetesen nem csak egy bemenő paramétert használhatunk:

```
delegate bool MyDelegate(int x, int y);
MyDelegate d = (x, y) => x > y;
Console.WriteLine(d(10, 5)); //True
```

Ebben a példában megvizsgáltuk, hogy az első paraméter nagyobb –e mint a második.
A lambda kifejezések elsősorban a LINQ miatt kerültek a nyelvbe, erről egy későbbi fejezetben olvashatunk.
A következőkben a lambda kifejezések felhasználási területeit nézzük meg.

27.1 Generikus kifejezések

Generikus kifejezéseknek (tulajdonképpen ezek generikus delegate –ek) is megadhatunk lambda kifejezéseket amelyek nem igénylik egy előzőleg definiált delegate jelenlétét, ezzel önálló lambda kifejezéseket hozva létre (ugyanakkor a generikus kifejezések kaphatnak névtelen metódusokat is). Kétféle generikus kifejezés létezik a *Func* amely adhat visszatérési értéket és az *Action*, amely nem (*void*). Elsőként a *Func* –ot vizsgáljuk meg:

```
Func<int, int> square = x => x * x;
square(3); //9
```

A generikus paraméterek között utolsó helyen mindig a visszatérési érték áll, előtte pedig a bemenő paraméterek kapnak helyet. Itt is használhatunk több (maximum négy bemenő) paramétert is:

```
Func<int, int, bool> BiggerOrLesser = (x, y) => x > y;
Console.WriteLine(BiggerOrLesser(10, 5)); //True
```

Amikor *Func* –ot használunk minden esetben kell visszatérési értéknek lennie. Létezik olyan változata is, amely csak a visszatérési érték típusát kapja meg generikus paraméterként:

```
Func<bool> RetTrue = () => true;
```

A *Func* párja az *Action* amely maximum négy bemenő paramétert kaphat, és nem lehet visszatérési értéke:

```
Action<int> IntPrint = x => Console.WriteLine(x);
```

27.2 Kifejezésfák

Generikus kifejezések segítségével felépíthetünk kifejezésfákat, amelyek olyan formában tárolják a kifejezésben szereplő adatokat és műveleteket, hogy futási időben a CLR ki tudja azt értékelni. Egy kifejezésfa egy generikus kifejezést kap generikus paraméterként:

```
using System;
using System.Linq.Expressions;

class Program
{
    static public void Main()
    {
        Expression<Func<int, int, bool>> BiggerOrLesserExpression =
            (x, y) => x > y;

        Console.WriteLine(BiggerOrLesserExpression.Compile().Invoke(10, 5));
        Console.ReadKey();
    }
}
```

A programban először IL kódra kell fordítani (*Compile()*), csak azután hívhatjuk meg. Kifejezésfákról még olvashatunk a LINQ –ről szóló fejezetekben.

27.3 Lambda kifejezések változóinak hatóköre

Egy lambda kifejezésben hivatkozhatunk annak a metódusnak a paramétereire és lokális változóira amelyben definiáltuk. A külső változók akkor értékelődnek ki amikor a delegate ténylegesen meghívódik, nem pedig a deklaráláskor, vagyis az adott változó legutolsó értékadása számít majd. A felhasznált változókat inicializálni kell mielőtt használnánk egy lambda kifejezésben. A lambda kifejezés fenntart magának egy másolatot a lokális változóból/paraméterből, még akkor is, ha az időközben kifut a saját hatóköréből:

```
using System;

delegate int MyDelegate(int x);

class LambdaTest
{
```

```
    public MyDelegate d;

    public void Test()
    {
        int localVar = 11;

        d = x =>
            {
                return (localVar * x);
            };

        localVar = 100; //ez lesz a delegate -ben
    }
}

class Program
{
    static public void Main()
    {
        LambdaTest lt = new LambdaTest();
        lt.Test();
        //localVar már kifutott a hatóköréből, de a lambdának még megvan
        Console.WriteLine(lt.d(10)); //1000
        Console.ReadKey();
    }
}
```

A lokális változók és paraméterek módosíthatóak egy lambda kifejezésben.
A lambda kifejezésben létrehozott változók ugyanúgy viselkednek, mint a hagyományos lokális változók, a delegate minden hívásakor új példány jön létre belőlük.

27.4 Eseménykezelők

Rövid eseménykezelők megírásához kiválóan alkalmasak a lambda kifejezések.
Vegyük például azt az esetet, amikor egy gomb Click eseményét kívánjuk kezelni (erről a Windows Forms –ról szóló fejezetekben olvashat az olvasó):

```
EventHandler<EventArgs> ClickHandler = null;
ClickHandler = (sender, e) =>
    {
        //itt csinálunk valamit
    };
button1.Click += ClickHandler;
```

Vegyük észre, hogy a kifejezés a típus megadása nélkül is tudja kezelni a pramétereket.
Ebben az esetben egyetlen dologra kell figyelni, mégpedig arra, hogy az eseménykezelő nem az alkalmazás főszálában fog lefutni.

28. Attribútumok

Már találkoztunk nyelvbe épített módosítokkal, mint amilyen a *static* vagy a *virtual*. Ezek általában be vannak betonozva az adott nyelvbe, mi magunk nem készíthetünk újakat – kivéve, ha a .NET Framework –el dolgozunk.
Az alapértelmezett telepítés is rengeteg előre definiált attribútumot szolgáltat, ezek a megfelelő fejezetekben bemutatásra kerülnek majd.
Egy attribútum a fordítás során beépül a Metadata információkba, amelyeket a futtató környezet (a CLR) felhasznál majd az objektumok kreálása során.

Egy tesztelés során általánosan használt attribútum a *Conditional*, amely egy előrdító (ld. következő fejezet) által definiált szimbólumhoz köti programrészek végrehajtását. A *Conditional* a *System.Diagnostics* névtérben rejtőzik:

```
#define DEBUG_ON

using System;
using System.Diagnostics;

class DebugClass
{
    [Conditional("DEBUG_ON")]
    static public void DebugMessage(string message)
    {
        Console.WriteLine("Debug message: {0}", message);
    }
}

class Program
{
    static public void Main()
    {
        DebugClass.DebugMessage("Main started...");
        Console.ReadKey();
    }
}
```

Egy attribútumot mindig szögletes zárójelek közt adunk meg. Ha nem lenne definiálva az adott szimbólum a metódus nem futna le.
Konvenció szerint minden attribútum neve a névből és az utána írt „attribute” szóból áll, így a *Conditional* eredeti neve is *ConditionalAttribute*, de ezt elhagyhatjuk, mivel a fordító feloldja majd.

Ahogy az már elhangzott, mi magunk is készíthetünk attribútumokat:

```
[System.AttributeUsage(System.AttributeTargets.Class)]
public class VersionAttribute : System.Attribute
{
    private double version;

    public VersionAttribute() : this(1.0) { }
```

```
    public VersionAttribute(double _version)
    {
    }

    public double Version
    {
        get { return version; }
        set { version = value; }
    }
}
```

Az attribútumosztály maga is kapott egy attribútumot, amellyel azt jelezzük, hogy hol használhajuk azt, ebben az esetben csakis referenciatípusokon.

Ezután (például) a következőképpen használhatjuk:

```
[VersionAttribute(Version = 1.0)]
class MyVersionedClass
{

}
```

Az attribútumok értékeihez pedig így férünk hozzá:

```
System.Attribute[] attr =
System.Attribute.GetCustomAttributes(typeof(MyVersionedClass));

foreach(System.Attribute a in attr)
{
    if(a is VersionAttribute)
    {
        Console.WriteLine("The version of the class: {0}", a.Version);
    }
}
```

29. Az előfordító

Az előfordító a tényleges fordítás előtt végzi el a dolgát, pl. a fordító számára értelmezhetővé alakítja a forráskódot.
A C és C++ nyelvekhez hasonlóan a C# is támogatja az előfordítónak szánt utasításokat, ezt a lehetőséget leggyakrabban feltételes végrehajtásra használjuk:

```
#define DEBUG_ON

#if DEBUG_ON
Console.WriteLine("Debug symbol exist...");
#endif
```

Az első sorban utasítjuk a fordítót, hogy az ún. szimbólumtáblázatba tárolja el a „DEBUG_ON” szimbólumot, ezután pedig ellenőrizzük, hogy definiálva van –e ez a szimbólum. Ha igen, akkor az adott kódrészlet lefordul, egyébként nem.

Összetett feltételt is készíthetünk:

```
//ha mindkét szimbólum definiálva van
#if SYMBOL_1 && SYMBOL_2

#endif
```

Szimbólum eltávolítására az *undef* utasítást használjuk:

```
#define SYMBOL

#if SYMBOL
Console.WriteLine("SYMBOL is defined...");
#endif

#undef SYMBOL
#define OTHER

//ha nincs definiálva
#if !SYMBOL
Console.WriteLine("SYMBOL is not defined...");
#elif OTHER //vagy ha egy másik szimbólumot definiáltunk
Console.WriteLine("OTHER is defined...");
#else
Console.WriteLine("No symbol defined...");
#endif
```

Az *elif* az else-if rövidítése, az *else* pedig már ismerős kell legyen.
A Visual Studio –ban használható egy kakukktojás, ami nem igazán előfordító utasítás, ez pedig a region-endregion, amellyel egy kódrészletet jelölünk ki, amit egy osztálydefinícióhoz hasonlóan kinyithatunk-becsukhatunk:

```
#region LOTOFCODE

#endregion
```

30. Unsafe kód

A C# lehetővé teszi a memória direkt elérését mutatókon keresztül, igaz ritkán fordul elő olyan probléma amikor ezt ki kell használnunk (pl.: amikor együtt kell működnünk az operációs rendszerrel, vagy meg kell valósítanunk egy bonyolult és időigényes algoritmust).
Ahhoz, hogy használhassunk mutatókat (pointereket) az adott metódust, osztályt, adattagot vagy blokkot az *unsafe* kulcsszóval kell jelölnünk:

```
class UnSafeClass
{
    public unsafe void UnSafeMethod()
    {

    }

    public void MethodWithUnSafeBlock()
    {
        unsafe
        {

        }
    }
}
```

Írjuk meg a „Hello Pointer!” programot nevéhez hűen mutatók használatával (A C/C++ programozók számára ismerős lesz):

```
using System;

class Program
{
    static public void Main()
    {
        unsafe
        {
            string message = "Hello Pointer!";
            string* messagePointer = &message;

            Console.WriteLine(messagePointer);
            Console.ReadKey();
        }
    }
}
```

Elsőként létrehoztunk egy változót az üzenettel, ezután pedig deklaráltunk egy *string* objektumra mutató mutatót (minden típushoz létezik a hozzá tartozó mutatótípus) és értékül adtuk neki az eredeti objektumunk memóriabeli címét, ezt a „címe operátorral” (*&*) és a „*” operátorral végeztük el, utóbbi az adott memóriacím értékét adja visza.

A programot prancssorból az /unsafe kapcsolóval fordíthatjuk le, Visual Studio esetén jobb klikk a projecten, Properties és ott állítsuk be.

csc **/unsafe** test.cs

A megfelelő explicit konverzióval a memóriacímet is lekérhetjük:

```
Console.WriteLine((int)messagePointer);
```

Unsafe kód esetén az osztálytagokat is másként érjük el:

```
class Test
{
    public string message = "TestClass";
}

unsafe
{
    Test t = new Test();
    Test* tp = &t;

    Console.WriteLine(tp->message);
}
```

30.1 Fix objektumok

Normális esetben a szemétgyűjtő a memória töredezettségmentesítése érdekében mozgatja az objektumokat a memóriában. Egy pointer azonban mindig egy fix helyre mutat, hiszen a mutatott objektumnak a címét kértük le, ez pedig nem fog frissülni, ez pedig néhány esetben gondot okozhat (pl. amikor hoszabb ideig van a memóriában a mutatott adat, pl. valamilyen erőforrás). Ha szeretnénk, hogy az objektumok a helyükön maradjanak a *fixed* kulcsszót kell használnunk:

```
unsafe
{
    fixed(Test t = new Test())
    {
        Test* tp = &t;
        Console.WriteLine(tp->message);
    }
}
```

31. Többszálú alkalmazások

Egy Windows alapú operációs rendszerben minden „*.exe” indításakor egy ún. *process* jön létre, amely teljes mértékben elkülönül az összes többitől. Egy process – t az azonosítója (PID – Process ID) alapján különböztetünk meg a többitől.
Minden egyes process rendelkezik egy ún. fő szállal (*primary*– vagy *main thread*), amely a belépési pontja a programnak (ld. Main).
Azokat az alkalmazásokat, amelyek csak a fő szállal rendelkeznek thread-safe alkalmazásoknak nevezzük, mivel csak egy szál fér hozzá az összes erőforráshoz. Ugyanakkor ezek az alkalmazások hajlamosak „elaludni”, ha egy komplexebb feladatot hajtanak végre, hiszen a fő szál ekkor nem tud figyelni a felhasználó interakciójára.
Az ilyen helyzetek elkerülésére a Windows (és a .NET) lehetővé teszi másodlagos szálak (ún. *worker thread*) hozzáadását a fő szálhoz.
Az egyes szálak (a process –ekhez hasonlóan) önállóan működnek a folyamaton belül és „versenyeznek” az erőforrások használatáért (*concurrent access*).
Jó példa lehet a szálkezelés bemutatására egy szövegszerkesztő használata: amíg kinyomtatunk egy dokumentumot (egy mellékszállal) az alkalmazás fő szála továbbra is figyeli a felhasználótól érkező utasításokat.
A többszálú programozás legnagyobb kihívása a szálak és feladataik megszervezése, az erőforrások elosztása.

Fontos megértenünk, hogy valójában a többszálúság a számítógép által nyújtott illúzió, hiszen a processzor egyszerre csak egy feladatot tud végrehajtani (bár terjednek a többmagos rendszerek, de gondoljunk bele, hogy amikor hozzá sem nyúlunk a számítógéphez is ötven – száz szál fut), így el kell osztania az egyes feladatok közt a processzoridőt (ezt a a szálak prioritása alapján teszi) ez az ún. időosztásos (*time slicing*) rendszer. Amikor egy szál ideje lejár a futási adatait eltárolja az ún. *Thread Local Storage* –ben (ebből minden szálnak van egy) és átadja a helyet egy másik szálnak, amely – ha szükséges – betölti a saját adatait a TLS -ből és elvégzi a feladatát.

A .NET számos osztályt és metódust bocsájt rendelkezésünkre, amelyekkel az egyes process –eket felügyelhetjük, ezek a *System.Diagnostics* névtérben vannak. Írjunk egy programot amely kiírja az összes futó folyamatot és azonosítójukat:

```
using System;
using System.Diagnostics;
class Program
{
    static public void Main()
    {
        //lekérjük az összes futó folyamatot
        Process[] processes = Process.GetProcesses(".");
        foreach(Process p in processes)
        {
            Console.WriteLine("PID: {0}, Name: {1}", p.Id, p.ProcessName);
        }
        Console.ReadKey();
    }
}
```

Amennyiben tudjuk a process azonosítóját, akkor használhatjuk a *Process.GetProcessById(azonosító)* metódust is.

A következő programunk az összes futó process minden szálát és azoknak adatait fogja kilistázni:

```
using System;
using System.Diagnostics;

class Program
{
    static public void Main()
    {
        Process[] processes = Process.GetProcesses(".");

        foreach(Process p in processes)
        {
            Console.WriteLine("The process {0}'s threads:", p.ProcessName);

            ProcessThreadCollection ptc = p.Threads;

            foreach(ProcessThread pt in ptc)
            {
                Console.WriteLine
                (
                    "This thread ({0}), start at {1} and its current state is {2}",
                    pt.Id, pt.StartTime, pt.ThreadState
                );
            }
        }

        Console.ReadKey();
    }
}
```

Elég valószínű, hogy a program futásakor kivételt kapunk, hiszen a szálak listájába olyan szál is bekerülhet amely a kiíráskor már befejezte futását (ez szinte mindig az Idle process (a rendszer üresjárati…) esetében fordul elő, a meggondolás házi feladat, akárcsak a program kivételbiztossá tétele).
A fenti osztályok segítségével remekül bele lehet látni a rendszer „lelkébe”, az MSDN –en megtaláljuk a fenti osztályok további metódusait, tulajdonságait amelyek az „utazáshoz” szükségesek.

A következő programunk a process –ek irányítását szemlélteti, indítsuk el az Internet Explorert, várjunk öt másodpercet és állítsuk le:

```
using System;
using System.Diagnostics;
using System.Threading;


class Program
{
```

```
    static public void Main()
    {
        Process ie = Process.Start("IExplore.exe");

        Thread.Sleep(5000); //A főszál pihentetése, ezredmásodpercben

        ie.Kill();
    }
}
```

Egyúttal felhasználtuk az első igazi szálkezeléshez tartozó metódusunkat is, a *Thread.Sleep()* –et (ld. a kommentet), a *Thread* osztály a *System.Threading* névtérben található.

31.1 Application Domain -ek

Egy .NET program nem direkt módon process –ként fut, hanem be van ágyazva egy ún. *application domain* –be a processen belül (egy process több AD –t is tartalmazhat egymástól teljesen elszeparálva).
Ezzel a megoldással egyrészt elősegítik a platformfüggetlenséget, hiszen így csak az *Application Domaint* kell portolni egy másik platformra, a benne futó folymatoknak nem kell ismerniük az operációs rendszert, másrészt biztosítja a programok stabilitását ugyanis ha egy alkalmazás összeomlik egy AD –ben, a többi még tökéletesen működik majd.
Amikor elindítunk egy .NET programot elsőként az alapértelmelmezett AD *(default application domain*) jön létre, ezután – ha szükséges – a CLR további AD –ket hoz létre.

A következő program kiírja az aktuális AD nevét:

```
using System;

class Program
{
    static public void Main()
    {
        AppDomain currAD = AppDomain.CurrentDomain;
        Console.WriteLine(currAD.FriendlyName);
        Console.ReadKey();
    }
}
```

A képernyőn megjelenik az alkalmazás neve, hiszen ő az alapértelmezett AD és egyelőre nincs is több.

Hozzunk létre egy második AppDomaint:

```
using System;
class Program
{
    static public void Main()
    {
        AppDomain secondAD = AppDomain.CreateDomain("SecondAD");
```

```
            Console.WriteLine(secondAD.FriendlyName);

            AppDomain.UnLoad(secondAD); //megszüntetjük

            Console.ReadKey();
        }
}
```

31.2 Szálak

Elérkeztünk a fejezet eredeti tárgyához, már eleget tudunk ahhoz, hogy megértsük a többszálú alkalmazások elvét. Első programunkban lekérjük az adott programrész szálának az azonosítóját:

```
using System;
using System.Threading;

class Program
{
     static public void Main()
     {
       Console.WriteLine("Hello from thread {0}",
Thread.CurrentThread.ManagedThreadId);
            Console.ReadKey();
     }
}
```

A kimenet minden esetben a következő lesz:

```
Hello from thread 1
```

Azaz az első, a „fő” szálban vagyunk.

A program utasításainak végrehajtása szerint megkülönböztetünk szinkron- és aszinkron működést. A fenti program szinkron módon működik, az utasításait egymás után hatja végre, ha esetleg egy hosszas algoritmusba ütközik akkor csak akkor lép a következő utasításra ha azt befejezte,
Az aszinkron végrehajtás ennek épp az ellentéte az egyes feladatokat el tudjuk küldeni egy másik szálba a fő szál pedig fut tovább, amíg a mellékszál(ak) vissza nem térnek.

31.3 Aszinkron delegate -ek

A következőkben delegate –ek segítségével fogunk aszinkron programot írni, így tisztában kell lennünk a delegate –ek mibenlétével ezért ha valami nem világos inkább olvassunk vissza.
Minden egyes delegate rendelkezik azzal a képességgel, hogy aszinkron módon hívjuk meg, ez a *BeginInvoke()* és *EndInvoke()* metódusokkal valósul meg.
Vegyük a következő delegate –et:

```
delegate int MyDelegate(int x);
```

Ez valójában így néz ki:

```
public sealed class MyDelegate : System.MulticastDelegate
{
    //...metódusok...

    public IAsyncResult BeginInvoke(int x, AsyncCallback cb, object state);

    public int EndInvoke(IAsyncResult result);
}
```

Egyelőre ne foglalkozzunk az ismeretlen dolgokkal, nézzük meg azt amit ismerünk. A *BeginInvoke* –kal fogjuk meghívni a delegate –et, ennek első paramétere megegyezik a delegate paraméterével (vagy paramétereivel).
Az *EndInvoke* fogja majd az eredményt szolgáltatni, ennek visszatérési értéke megegyezik a delegate –éval.
Az *IasyncResult* amit a *BeginInvoke* visszatérít segít elérni az eredményt és az *EndInvoke* is ezt kapja majd paraméteréül.
A *BeginInvoke* másik két paraméterével most nem foglalkozunk, készítsünk egy egyszerű delegate –et és hívjuk meg aszinkron:

```
using System;
using System.Threading;

class Program
{
    public delegate int MyDelegate(int x);

    static int Square(int x)
    {
        Console.WriteLine("Hello from thread {0}",
Thread.CurrentThread.ManagedThreadId);
        return (x * x);
    }

    static public void Main()
    {
        MyDelegate d = Square;
        Console.WriteLine("Hello from thread {0}",
Thread.CurrentThread.ManagedThreadId);

        IAsyncResult iar = d.BeginInvoke(12, null, null);

        Console.WriteLine("BlaBla...");

        int result = d.EndInvoke(iar);

        Console.WriteLine(result);

        Console.ReadKey();
    }
}
```

A kimenet a következő lesz:

```
Hello from thread 1
BlaBla...
Hello from thread 3
144
```

Látható, hogy egy új szál jött létre. Amit fontos megértenünk, hogy a *BeginInvoke* azonnal megkezdi a feladata végrehajtását, de az eredményhez csak az *EndInvoke* hívásakor jutunk hozzá, tehát külső szemlélőként úgy látjuk, hogy csak akkor fut le a metódus. A háttérben futó szál üzenete is látszólag csak az eredmény kiértékelésénél jelenik meg, az igazság azonban az, hogy a *Main* üzenete előbb ért a processzorhoz, ezt hamarosan látni fogjuk.

Többszálú program írásánál össze kell tudnunk hangolni a szálak munkavégzését, pl. ha az egyik szálban kiszámolt eredményre van szüksége egy másik, később indult szálnak. Az ilyen eseteket szinronizálásnak nevezzük.
Szinkronizáljuk az eredeti programunkat, vagyis várjuk meg amíg a delegate befejezi a futását (természetesen a szinkronizálás ennél jóval bonyolultabb, erről a következő fejezetekben olvashatunk):

```
IAsyncResult iar = d.BeginInvoke(12, null, null);

while(!iar.IsCompleted)
{
    Console.WriteLine("BlaBla...");
    Thread.Sleep(1000);
}

int result = d.EndInvoke(iar);

Console.WriteLine(result);
```

Ezt a feladatot az *IAsyncResult* interfész *IsCompleted* tulajdonságával oldottuk meg.
A kimenet:

```
Hello from thread 1
BlaBla...
Hello from thread 3
BlaBla...
BlaBla...
BlaBla...
BlaBla...
144
```

Itt már tisztán látszik, hogy az aszinkron metódus futása azonnal elkezdődött, igaz a *Main* itt is megelőzte.

Valljuk be elég macerás mindig meghívogatni az *EndInvoke* –ot, felmerülhet a kérdés, hogy nem lehetne valahogy automatizálni az egészet. Nos, épp ezt a gondot oldja meg a *BeginInvoke* harmadik *AsyncCallback* típusú paramétere. Ez egy delegate amely egy olyan metódusra mutathat, amelynek visszatérési értéke *void*,

valamint egy darab *IAsyncResult* típusú paraméterrel rendelkezik. Ez a metódus azonnal le fog futni, ha a mellékszál elvégezte a feladatát:

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;


class Program
{
    public delegate int MyDelegate(int x);

    static int Square(int x)
    {
        Console.WriteLine("Hello from thread {0}",
Thread.CurrentThread.ManagedThreadId);
        return (x * x);
    }

    static void AsyncMethodComplete(IAsyncResult iar)
    {
        Console.WriteLine("Async thread complete...");

        AsyncResult result = (AsyncResult)iar;
        MyDelegate d = (MyDelegate)result.AsyncDelegate;

        Console.WriteLine("Result: {0}", d.EndInvoke(iar));
    }

    static public void Main()
    {
        MyDelegate d = Square;
        Console.WriteLine("Hello from thread {0}",
Thread.CurrentThread.ManagedThreadId);

        IAsyncResult iar = d.BeginInvoke(12, new
AsyncCallback(AsyncMethodComplete), null);

        Console.WriteLine("BlaBla...");

        Console.ReadKey();
    }
}
```

A kimenet a következő lesz:

```
Hello from thread 1
BlaBla...
Hello from thread 3
Async thread complete...
Result: 144
```

Egyetlen dolog van hátra mégpedig a *BeginInvoke* utolsó paraméterének megismerése. Ez egy *object* típusú változó, azaz bármilyen objektumot átadhatunk. Ezt a paramétert használjuk, ha valamilyen plusz információt akarunk továbbítani:

```
IAsyncResult iar = d.BeginInvoke(12, new AsyncCallback(AsyncMethodComplete), "This is a message");

//és

static void AsyncMethodComplete(IAsyncResult iar)
{
    Console.WriteLine("Async thread complete...");

    AsyncResult result = (AsyncResult)iar;
    MyDelegate d = (MyDelegate)result.AsyncDelegate;

    Console.WriteLine("State: {0}", iar.AsyncState);
    Console.WriteLine("Result: {0}", d.EndInvoke(iar));
}
```

31.4 Szálak létrehozása

Ahhoz, hogy másodlagos szálakat hozzunk létre nem feltétlenül kell delegate –eket használnunk, mi magunk is elkészíthetjük. Vegyük a következő programot:

```
using System;
using System.Threading;

class TestClass
{
    public void PrintThreadInfo()
    {
        Console.WriteLine("I'm in thread: {0}", Thread.CurrentThread.Name);
    }
}

class Program
{
    static public void Main()
    {
        Thread first = Thread.CurrentThread;
        tmp.Name = "FirstThread";

        TestClass tc = new TestClass();
        tc.PrintThreadInfo();
        Console.ReadKey();
    }
}
```

Elsőként elneveztük az elsődleges szálat, hogy később azonosítani tudjuk, melyik-melyik. Most készítsünk egy másik objektumot, de ezúttal küldjük el a háttérbe:

```
Thread first = Thread.CurrentThread;
tmp.Name = "FirstThread";

TestClass tc = new TestClass();
tc.PrintThreadInfo();
```

```
TestClass ct = new TestClass();

Thread backgroundThread = new Thread(new ThreadStart(ct.PrintThreadInfo));
backgroundThread.Name = "SecondThread";
backgroundThread.Start();
```

A kimenet pedig ez lesz:

```
I'm in thread: FirstThread
I'm in thread: SecondThread
```

Ez eddig szép és jó, de mi van akkor, ha a meghívott metódusnak paraméterei is vannak? Ilyenkor a *ThreadStart* parametrizált változatát használhatjuk, ami igen eredeti módon a *ParameterizedThreadStart* névre hallgat. A *ThreadStart* –hoz hasonlóan ez is egy delegate, szintén *void* visszatérési típusa lesz, a paramétere pedig *object* típusú lehet:

```
//az új metódus
public void ParameterizedThreadInfo(object parameter)
{
    if(parameter is string)
    {
        Console.WriteLine("Parameter: {0}", (string)parameter);
    }
}
```

Most már hívhatjuk:

```
Thread backgroundThread = new
        Thread(newParameterizedThreadStart(ct.ParameterizedThreadInfo));

backgroundThread.Name = "SecondThread";
//itt adjuk meg a paramétert
backgroundThread.Start("This is a parameter");
```

31.5 Foreground és background szálak

A .NET két különböző száltípust különböztet meg: amelyek előtérben és amelyek a háttérben futnak. A kettő közti különbség a következő: a CLR addig nem állítja le az alkalmazást, amíg egy előtérbeli szál is dolgozik, ugyanez a háttérbeli szálakra nem vonatkozik.
Logikus (lenne) a feltételezés, hogy az elsődleges és másodlagos szálak fogalma megegyezik jelen fejezetünk tárgyaival. Az igazság viszont az, hogy ez az állítás nem igaz, ugyanis alapértelmezés szerint minden szál (a létrehozás módjától és idejétől függetlenül) előtérben fut. Természetesen van arra is lehetőség, hogy a háttérbe küldjük őket:

```
Thread backgroundThread = new Thread(new ThreadStart(SomeUndeclaredMethod));
backGroundThread.IsBackground = true;
backgroundThread.Start();
```

31.6 Szinkronizáció

A szálak szinkronizációjának egy primitívebb formáját már láttuk a delegate –ek esetében, most egy kicsit komolyabban közelítünk a témához.
Négyféleképpen szinkronizálhatjuk a szálainkat, ezek közül az első a blokkolás. Ennek már ismerjük egy módját, ez a *Thread.Sleep* metódus:

```
using System;
using System.Threading;

class Program
{
    static public void Main()
    {
        Console.WriteLine("Start...");
        Thread.Sleep(5000);
        Console.WriteLine("Stop...");
        Console.ReadKey();
    }
}
```

Ez a statikus metódus a paramétereként ezredmásodpercben megadott ideig várakoztatja azt a szálat, amelyben meghívták.
Amikor egy szálat leblokkolunk az azonnal „elereszti” a processzort és addig inaktív marad, amíg a blokkolás feltételének a környezet eleget nem tesz, vagy a folyamat valamilyen módon megszakad.
Egy viszonylag ritkán használt metódusa a *Thread* osztálynak a *SpinWait*. Ez hasonlóan működik mint a *Sleep*, de benne marad a processzorban, csak éppen nem csinál semmit, folyamatosan üres utasításokat hajt végre. Látszólag nem sok értelme van egy ilyen metódus használatának, de olyankor hasznos, amikor csak nagyon rövid (néhány ezredmásodperc) időt kell várni, mondjuk egy erőforrásra, ekkor nagy költség lenne cserélgetni a szálakat, mint azt a *Sleep* teszi.
A *Join* metódus addig várakoztatja a hívó szálat, amíg az a szál amin meghívták nem végezte el a feladatát:

```
using System;
using System.Threading;

class Program
{
    static public void Main()
    {
        Thread t = new Thread( delegate(){ Thread.Sleep(3000); });
        t.Start();
        t.Join();
        Console.WriteLine("The End");
        Console.ReadKey();
    }
}
```

A *Join* –nak megadhatunk egy „timeout” paramétert (ezredmásodpercben), amely idő lejárta után – ha a szál nem végzett feladatával – hamis értékkel tér vissza. A következő példa ezt mutatja meg:

```
using System;
using System.Threading;

class Program
{
    static public void Main()
    {
        Thread t = new Thread( delegate()
            {
                Thread.Sleep(3000);
                Console.WriteLine("Hello from Thread");
            });

        t.Start();

        if(t.Join(1000) == false)
        {
            Console.WriteLine("Timed out...");
            t.Abort(); //megszakítjuk a szál futását
        }
        Console.ReadKey();
    }
}
```

A példában a szál üzenete már nem fog képernyőre kerülni, még előtte megszakítjuk.

A következő szinkronizációs módszer a lezárás (locking). Ez azt jelenti, hogy erőforrásokhoz, illetve a program bizonyos részeihez egyszerre csak egy szálnak engedünk hozzáférést. Ha egy szál hozzá akar férni az adott dologhoz, amelyet egy másik szál már használ, akkor automatikusan blokkolódik és várólistára kerül, ahonnan érkezési sorrendben lehet hozzájutni az erőforráshoz (ha az előző szál már végzett).
Nézzük a következő példát:

```
static int x, y;

static public Divide()
{
    if(x != 0) Console.WriteLine(y / x);

    x = 0;
}
```

Tegyük fel, hogy megérkezik egy szál, eljut odáig, hogy kiírja az eredményt és épp ekkor érkezik egy másik szál is. Megvizsgálja a feltételt, rendben találja és továbblép. Ebben a pillanatban azonban az elsőként érkezett szál lenullázza a változót és amikor a második szál osztani akar, akkor kap egy szép kis kivételt.
A műveletet lezárhatjuk a következő módon:

```
static int x, y;
static object locker = new object();

static public Divide()
```

```
{
    lock(locker)
    {
        if(x != 0) Console.WriteLine(y / x);
        x = 0;
    }
}
```

A *lock* kijelöl egy blokkot, amelyhez egyszerre csak egy szál fér hozzá. Ahhoz azonban, hogy ezt megtehessük ki jelölnünk egy ún. tokent, amelyet lezárhat. A tokennek minden esetben referenciatípusnak kell lennie. Természetesen nem csak statikus metódusokból áll a világ, egy „normális" metódust is lezárhatunk, de ilyen esetekben az egész objektumra kell vonatkoztatni a zárolást, hogy ne változzon meg az állapota egy másik szál keze nyomán:

```
class LockedClass
{
    public void LockedMethod()
    {
        lock(this)
        {
            /*BlaBla*/
        }
    }
}
```

A *lock* –hoz hasonlóan működik a *Mutex* is, a legnagyobb különbség az a kettő közt, hogy utóbbi processz szinten zárol, azaz a számítógépen futó összes folyamat elöl elzárja a használat lehetőségét. Az erőforrás/metódus/stb. használata előtt meg kell hívnunk a *WaitOne* metódust, a használat után pedig el kell engednünk az erőforrást a *ReleaseMutex* metódussal (ha ezt nem tesszük meg a kódból, akkor az alkalmazás futásának végén a CLR automatikusan megteszi helyettünk). A következő példában létrehozunk több szálat és versenyeztetjük őket egy metódus használatáért:

```
using System;
using System.Threading;

class Program
{
    private static Mutex m = new Mutex();

    static void ResourceMethod()
    {
        m.WaitOne();
        Console.WriteLine("Thread {0} use this resource...",
Thread.CurrentThread.Name);

        Thread.Sleep(1000);
        Console.WriteLine("Thread {0} release this resource",
Thread.CurrentThread.Name);

        m.ReleaseMutex();
    }
```

```
    static void ThreadProc()
    {
        for(int i = 0;i < 10;++i)
        {
            ResourceMethod();
        }
    }

    static public void Main()
    {
        Thread[] threads = new Thread[10];

        for(int i = 0;i <threads.Length;++i)
        {
            threads[i] = new Thread(new ThreadStart(ThreadProc));
            threads[i].Name = i.ToString();
            threads[i].Start();
        }

        Console.ReadKey();
    }
}
```

A *Semaphore* hasonlít a *lock* –ra és a *Mutex* –re, azzal a különbséggel, hogy megadhatunk egy számot, amely meghatározza, hogy egy erőforráshoz maximum hány szál férhet hozzá egy időben. A következő program az előző átirata, egy időben maximum három szál férhet hozzá a metódushoz:

```
using System;
using System.Threading;

class Program
{
    private static Semaphore s = new Semaphore(3, 3);

    static void ResourceMethod()
    {
        s.WaitOne();

        Console.WriteLine("Thread {0} use this resource...",
Thread.CurrentThread.Name);

        Thread.Sleep(500);

        Console.WriteLine("Thread {0} release this resource",
Thread.CurrentThread.Name);

        s.Release();
    }

    static void ThreadProc()
    {
        for(int i = 0;i < 10;++i)
        {
            ResourceMethod();
        }
```

```
    }

    static public void Main()
    {
        Thread[] threads = new Thread[10];

        for(int i = 0;i <threads.Length;++i)
        {
            threads[i] = new Thread(new ThreadStart(ThreadProc));
            threads[i].Name = i.ToString();
            threads[i].Start();
        }

        Console.ReadKey();
    }
}
```

32. Reflection

A programozástechnikában a „reflection” fogalmát olyan programozási technikára alkalmazzuk, ahol a program (futás közben) képes megváltoztatni saját struktúráját és viselkedését. Az erre a paradigmára épülő programozást reflektív programozásnak nevezzük.
Ebben a fejezetben csak egy rövid példát találhat a kedves olvasó, a Reflection témaköre óriási és a mindennapi programozási feladatok végzése közben viszonylag ritkán szorulunk a használatára (ugyanakkor bizonyos esetekben nagy hatékonysággal járhat a használata).
Vegyük a következő példát:

```
using System;

class Test { }

class Program
{
    static public void Main()
    {
        Test t = new Test();
        Type tp = t.GetType();
        Console.WriteLine(tp);
        Console.ReadKey();
    }
}
```

Futásidőben lekértük az objektum típusát (a *GetType* metódust minden osztály örökli az *object* –től), persze a reflektív programozás ennél többről szól, lépjünk előre egyet:

```
using System;
using System.Reflection;

class Test
{
    public void Method()
    {
        Console.WriteLine("Hello Reflection!");
    }
}

class Program
{
    static public void Main()
    {
        Type tp = Assembly.GetCallingAssembly().GetType("Test");
        tp.InvokeMember("Method", BindingFlags.InvokeMethod, null,
Activator.CreateInstance(tp), null);
        Console.ReadKey();
    }
}
```

Ebben az esetben az objektum létrehozása és a metódusa meghívása teljes mértékben futási időben történik. Megjegyzendő, hogy fordítási időben semmilyen ellenőrzés sem történik a példányosítandó osztályra nézve, így ha elgépeltünk valamit az bizony kivételt fog dobni futáskor (*System.ArgumentNullException*).
Az *InvokeMember* első paramétere annak a konstruktornak/metódusnak/tulajdonságnak/… a neve amelyet meghívunk. A második paraméterrel jelezzük, hogy mit és hogyan fogunk hívni (a fenti példában egy metódust). Következő a sorban a *binder* paraméter, ezzel beállíthatjuk, hogy az öröklött/túlterhelt tagokat hogyan hívjuk meg. Ezután maga a típus amin a hívást elvégezzük, végül a hívás paraméterei (ha vannak)).

33. Állománykezelés

Ebben a fejezetben megtanulunk file –okat írni/olvasni és a könyvtárstruktúra állapotának lekérdezését illetve módosítását.

33.1 Olvasás/írás file –ból/file –ba

A .NET számos osztályt biztosít számunkra, amelyekkel file –okat udunk kezelni, ebben a fejezetben a leggyakrabban használtakkal ismerkedünk meg.
Kezdjük egy file megnyitásával és tartalmának a képernyőre írásával. Legyen pl. a szöveges file tartalma a következő:

```
alma
korte
dio
csakany
konyv
penz
```

A program pedig:

```
using System;
using System.IO;

class Program
{
    static public void Main()
    {
        FileStream fs = new FileStream("Text.txt", FileMode.Open);
        StreamReader rs = new StreamReader(fs);

        string s = rs.ReadLine();
        while(s != null)
        {
            Console.WriteLine(s);
            s = rs.ReadLine();
        }

        rs.Close();
        fs.Close();

        Console.ReadKey();
    }
}
```

Az IO osztályok *a System.IO* névtérben vannak.
A C# ún. stream –eket, adatfolyamokat használ az IO műveletek végzéséhez. Az első sorban megnyitottunk egy ilyen folyamot és azt is megmondtuk, hogy mit akarunk csinálni vele.
A *FileStream* konstruktorának első paramétere a file neve. Ha nem adunk meg teljes elérési útat, akkor automatikusan a saját könyvtárában fogja keresni a program. Ha külső könyvtárból szeretnénk megnyitni az állományt, akkor azt a következőképpen tehetjük meg:

```
FileStream fs = new FileStream("C:\\Egymasikkonyvtar\\Masikkonyvtar\\text.txt",
FileMode.Open);
```

Azért használunk dupla backslash –t (\), mert az egy ún. escape karakter, magában nem lehetne használni (persze ez nem azt jelenti, hogy minden ilyen karaktert kettőzve kellene írni, minden ilyen esetben a backslash –t kell használnunk).

Egy másik lehetőség, hogy az „at” jelet (@) használjuk az elérési út előtt, ekkor nincs szükség dupla karakterekre, mivel minden karaktert normális karakterként fog értelmezni:

```
FileStream fs = new FileStream(@"C:\Egymasikkonyvtar\Masikkonyvtar\text.txt",
FileMode.Open);
```

A *FileMode* enum –nak a következő értékei lehetnek:

Create	Létrehoz egy új file –t, ha már létezik a tartalmát kitörli
CreateNew	Ugyanaz mint az előző, de ha már létezik a file, akkor kivételt dob.
Open	Megnyit egy file –t, ha nem létezik kivételt dob.
OpenOrCreate	Ugyanaz mint az előző, de ha nem létezik akkor létrehozza a file –t.
Append	Megnyit egy file –t és automatikusan a végére pozícionál. Ha nem létezik létrehozza.
Truncate	Megnyit egy létező file –t és törli a tartalmát. Ebben a módban a file tartalmát nem lehet olvasni (egyébként kivételt dob).

A *FileStream* konstruktorának további két paramétere is lehet amelyek érdekesek számunkra (tulajdonképpen 15 féle konstruktora van), mindkettő enum típusú. Az első a *FileAccess*, amellyel beállítjuk, hogy pontosan mit akarunk csinálni az állománnyal:

Read	Olvasásra nyitja meg.
Write	Írásra nyitja meg.
ReadWrite	Olvasásra és írásra nyitja meg

A fenti példát így is írhattuk volna:

```
FileStream fs = new FileStream("text.txt", FileMode.Open, FileAccess.Read);
```

Végül a *FileShare* –rel azt állítjuk be, ahogy más folyamatok férnek hozzá a file –hoz:

None	Más folyamat nem férhet hozzá a file –hoz, amíg azt be nem zárjuk.
Read	Más folyamat olvashatja a file –t.
Write	Más folyamat írhatja a file –t.
ReadWrite	Más folyamat írhatja és olvashatja is a file –t.
Delete	Más folyamat törölhet a file –ból (de nem magát a file –t).
Inheritable	A gyermek processzek is hozzáférhetnek a file –hoz.

Most írjunk is a file –ba. Erre a feladatra a *StreamReader* helyett a *StreamWriter* osztályt fogjuk használni:

```
using System;
using System.IO;

class Program
{
    static public void Main()
    {
        FileStream fs = new FileStream("Text.txt", FileMode.Open, FileAccess.Write,
FileShare.None);
        StreamWriter sw = new StreamWriter(fs);

        Random r = new Random();
        for(int i = 0;i < 10;++i)
        {
            sw.Write(r.Next().ToString());
            sw.Write("\n");
        }

        sw.Close();
        fs.Close();

        Console.WriteLine("Operation success!");

        Console.ReadKey();
    }
}
```

Házi feladat következik: módosítsuk a programot, hogy ne dobja el az eredeti tartalmát a file –nak, és az írás után azonnal írjuk ki a képernyőre az új tartalmát.

Bináris file –ok kezeléséhez a *BinaryReader*/*BinaryWriter* osztályokat használhatjuk:

```
using System;
using System.IO;

class Program
{
    static public void Main()
    {
        BinaryWriter bw = new BinaryWriter(File.Create("file.bin"));
```

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

            bw.Close();

            BinaryReader br = new BinaryReader(File.Open("file.bin", FileMode.Open));

            while(br.PeekChar() != -1)
            {
                  Console.WriteLine(br.ReadInt32());
            }

            br.Close();

            Console.ReadKey();
      }
}
```

Készítettünk egy bináris file –t, és kiírtuk belé a számokat egytől százig. Ezután megnyitottuk a már létező file –t és elkezdjük kiovasni a tartalmát. A *PeekChar()* metódus a soron következő karaktert (byte –ot) adja vissza, illetve *-1* –et, ha elértük a file végét. A folyambeli aktuális pozíciót nem változtatja meg.
A cikluson belül van a trükkös rész. A *ReadValami* metódus a megfelelő típusú adatot adja vissza, de vigyázni kell vele, mert ha nem megfelelő méretű a beolvasandó adat, akkor hibázni fog. A fenti példában, ha a *ReadString* metódust használtuk volna akkor kivétel (*EndOfStreamException*) keletkezik, mivel a kettő nem ugyanakkor mennyiségű adatot olvas be. Az integer –eket beolvasó metódus nyílvná működni fog, hiszen tudjuk, hogy számokat írtunk ki.

33.2 Könyvtárstruktúra kezelése

A file –o kkezelése mellett a .NET a könyvtárstruktúra kezelését is széleskörűen támogatja. A *System.IO* névtér ebből a szempontból két részre oszlik: információs- és operációs eszközökre. Előbbiek (ahogyan a nevük is sugallja) információt szolgáltatnak, míg az utóbbiak (többségükben statikus metódusok) bizonyos műveleteket (új könyvtár létrehozása, törlése, stb.) végeznek a filerendszeren.
Első példánkban írjuk ki mondjuk a C meghajtó gyökerének könyvtárait:

```
using System;
using System.IO;

class Program
{
      static public void Main()
      {
            foreach(string s in Directory.GetDirectories("C:\\"))
            {
                  Console.WriteLine(s);
            }

            Console.ReadKey();
```

```
    }
}
```

Természetesen nem csak a könyvtárakra, de a file –okra is kíváncsiak lehetünk. A programunk módosított változata némi plusz információval együtt ezeket is kiírja nekünk:

```
using System;
using System.IO;

class Program
{
    static public void PrintFileSystemInfo(FileSystemInfo fsi)
    {
        if((fsi.Attributes & FileAttributes.Directory) != 0)
        {
            DirectoryInfo di = fsi as DirectoryInfo;
            Console.WriteLine("Directory {0}, created at {1}", di.FullName,
di.CreationTime);
        }
        else
        {
            FileInfo fi = fsi as FileInfo;
            Console.WriteLine("File {0} created at {1}", fi.FullName,
fi.CreationTime);
        }
    }

    static public void Main()
    {
        foreach(string s in Directory.GetDirectories("C:\\"))
        {
            PrintFileSystemInfo(new DirectoryInfo(s));
        }

        foreach(string s in Directory.GetFiles("C:\\"))
        {
            PrintFileSystemInfo(new FileInfo(s));
        }
        Console.ReadKey();
    }
}
```

Elsőként a mappákon, majd a file –okon megyünk végig. Ugyanazzal a metódussal írjuk ki az információkat kihasználva azt, hogy *DirectoryInfo* és a *FileInfo* is egy közös őstől a *FileSystemInfo* –ból származik (mindkettő konstruktora a vizsgált alany elérési útját várja paraméterként), így a metódusban csak meg kell vizsgálni, hogy éppen melyikkel van dolgunk és átkonvertálni a megfelelő típusra. A vizsgálatnál egy „bitenkénti és” műveletet hajtottunk végre, hogy ez miért és hogyan működik, annak meggondolása az olvasó feladata.

Eddig csak információkat kértünk le, most megtanuljuk módosítani is a könyvtárstruktúrát:

```
using System;
using System.IO;

class Program
{
    static public void Main()
    {
        string dirPath = "C:\\test";
        string filePath = dirPath + "\\file.txt";

        //ha nem létezik a könyvtár
        if(Directory.Exists(dirPath) == false)
        {
            //akkor megcsináljuk
            Directory.CreateDirectory(dirPath);
        }

        FileInfo fi = new FileInfo(filePath);

        //ha nem létezik a file
        if(fi.Exists == false)
        {
            //akkor elkészítjük és írunk bele
            StreamWriter sw = fi.CreateText();
            sw.WriteLine("Dio");
            sw.WriteLine("Alma");
            sw.Close();
        }

        Console.ReadKey();
    }
}
```

A *FileInfo CreateText* metódusa egy *StreamWriter* objektumot ad vissza, amellyel írhatunk egy szöveges file –ba.

33.3 In – memory stream –ek

A .NET a MemoryStream osztályt biztosítja számunkra, amellyel memóriabeli adatfolyamokat írhatunk/olvashatunk. Mire is jók ezek a folyamok? Gyakran van szükségünk arra, hogy összegyűjtsünk nagy mennyiségű adatot, amelyeket majd a folyamat végén ki akarunk írni a merevlemezre. Nyílván egy tömb vagy egy lista nem nyújtja a megfelelő szolgáltatásokat, előbbi rugalmatlan, utóbbi memóriaigényes, ezért a legegyszerűbb, ha közvetlenül a memóriában tároljuk el az adatokat. A *MemoryStream* osztály jelentősen megkönnyíti a dolgunkat, mivel lehetőséget ad közvetlenül file –ba írni a tartalmát. A következő program erre mutat példát:

```
using System;
using System.IO;

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

```
            MemoryStream mstream = new MemoryStream();
            StreamWriter sw = new StreamWriter(mstream);

            for(int i = 0;i < 1000;++i)
            {
                  sw.WriteLine(i);
            }

            sw.Flush();

            FileStream fs = File.Create("inmem.txt");
            mstream.WriteTo(fs);

            sw.Close();
            fs.Close();
            mstream.Close();

            Console.ReadKey();
      }
}
```

A *MemoryStream* –re „ráállítottunk” egy *StreamWriter* –t. Miután minden adatot a memóriába írtunk a *Flush()* metódussal, amely egyúttal kiűríti a *StreamWriter* –t is. Ezután létrehoztunk egy file -t és a *MemoryStream WriteTo* metódusával kiírtuk belé azadatokat.

34. Grafikus felületű alkalmazások fejlesztése

Az eddigi fejezetekben konzolalkalmazásokat készítettünk, így arra tudtunk koncentrálni amire kell, vagy is a nyelv elemeinek megértésére. Azonban a C# (és a .NET) igazi erejét olyan alkalmazások készítésével sajátíthatjuk el amelyek sokkal jobban kihasználják a platform képességeit.
A jegyzet hátralévő részében megismerkedünk a Windows Forms és a Windows Presentation Foundation alapelemeivel, megtanulunk adatokat és adatbázisokat kezelni a .NET 2.0 és a 3.5 (LINQ, Entity Framework) eszközeivel.
Elsőként a Windows Forms –ot vesszük górcső alá (lévén ő inkább kap helyet a felsőoktatásban mint fiatalabb társa), utána pedig a WPF következik.
A következő két alfejezet egy-egy egyszerű grafikus felületű alkalmazás készítését mutatja be a két technológia segítségével, a következő fejezetekben ez a párhuzamosság már nem fog helyet kapni.

34.1 Windows Forms - Bevezető

A Windows Forms a .NET első grafikus interfész API (Application Programming Interface) –jának neve. A WF az eredeti Windows API metódusati hívja meg menedzselt környezetben.
Készítsük hát el első ablakos programunkat. Most még parancssorból fogunk fordítani és a Visual Studio beépített tervezőjét sem fogjuk használni, ezért ez az egyszerű program is bonyolultnak fog tűnni elsőre, de a későbbiekben már jobb lesz a helyzet. A feladat a következő: az ablakban (hívjuk mostantól formnak) legyen egy gomb és rákattintva jelenjen meg egy *MessageBox* valamilyen üzenettel:

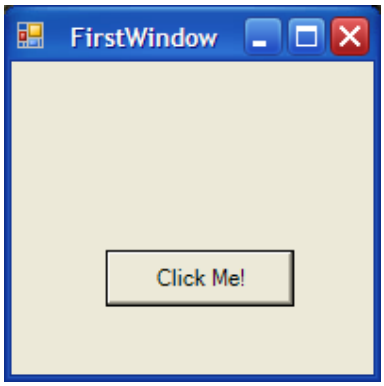
```
using System;
using System.Windows.Forms;

class FirstWindow : Form
{
    private Button button;

    public FirstWindow()
    {
        this.Text = "FirstWindow"
        this.Size = new System.Drawing.Size(200, 200);

        this.button = new Button();
        this.button.Text = "Click Me!";
        this.button.Size = new System.Drawing.Size(100, 30);
        this.button.Location = new System.Drawing.Point(50, 100);
        this.button.Click += new EventHandler(Button_Click);

        this.Controls.Add(button);
    }

    protected void Button_Click(object sender, EventArgs e)
    {
        MessageBox.Show("Hello WinForms!");
    }

    public static void Main()
```

```
    {
        Application.Run(new FirstWindow());
    }
}
```

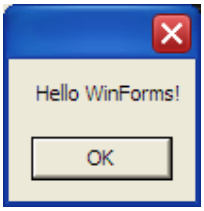
A programot a következőképpen fordíthatjuk le:

```
csc /target:winexe file.cs
```

A kapcsoló nélkül is lefordul, de a futáskor egy konzolablak is megjelenik.
Az eredmény ez lesz:



Ha pedig rákattintunk a gombra:



Jöjjön a magyarázat: a *FirstWindow* lesz az ablakunk osztálya, ezt a *System.Windows.Forms.Form* osztályból származtattuk. Az ablakon lesz egy gomb (*System.Windows.Forms.Button*), őt elneveztük *button* –nak. A form konstruktorában beállítjuk az ablak címsorát és méretét, ezután példányosítjuk a gombot, beállítjuk a szövegét (*Text*), a méretét (*Size*), az ablakon belüli helyzetét (*Location*), majd a kattintás (*Click*) eseményéhez hozzárendeltük az eseménykezelőt, amely majd a *MessageBox* –ot fogja megjeleníteni a képernyőn.
Végül hozzáadtuk a gombot a formhoz (a this szócska végig az osztályra azaz a formra mutat).
Az eseménykezelő metódusok visszatérési típusa mindig *void* és két paraméterük van, az első az eseményt kiváltó objektum (most a gomb), ez mindig object típusú, a második pedig egy az esemény információit tároló változó. Ez utóbbinak több specializált változata van (pl.: külön a billentyűzet és egér által kiváltott eseményekhez), de ha nincs különösebb szükségünk ezekre az adatokra nyugodtan használjuk az ősosztályt (*EventArgs*).

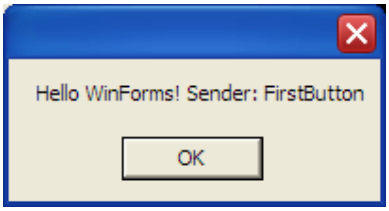
A *sender* paraméter segítségével felhasználhatjuk a küldő objektum adatait, mondjuk írjuk ki a küldő nevét is. Ehhez először nevet kell adnunk a gombnak, írjuk a konstruktorba:

```
this.button.Name = "FirstButton";
```

Az eseménykezelőben típuskonverziót kell végrehajtanunk, hogy elérjük a gomb tulajdonságait:

```
protected void Button_Click(object sender, EventArgs e)
{
    Button mp = (Button)sender;
    MessageBox.Show("Hello WinForms! Sender: " + mp.Name);
}
```

Most ez lesz a kimenet:

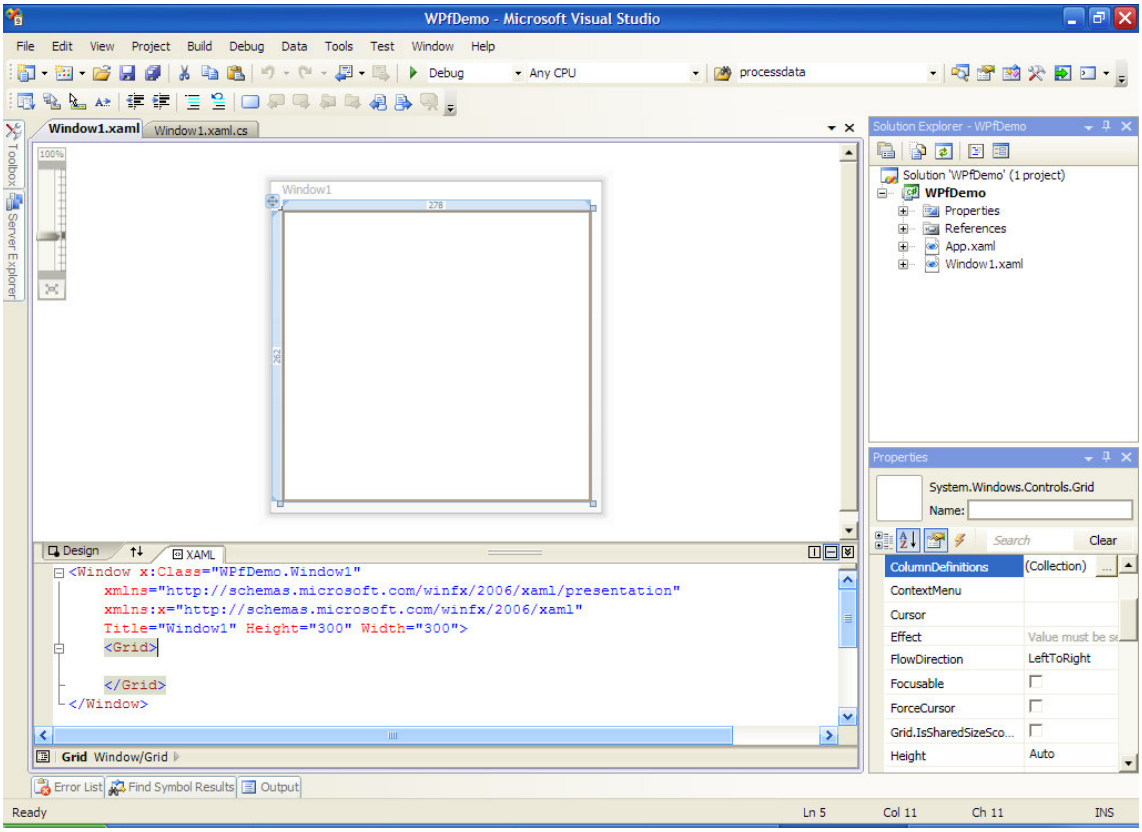


A *Main* függvény lesz a program belépési pontja, itt jelenítjük meg a formot, mégpedig az *Application* osztály *Run* metódusával amely paramétereként egy példányt kap a form –ból. Ez a metódus elindít egy programot az aktuális programszálban és irányítja a form és WinAPI közti kommunikációt.

34.2 Windows Presentation Foundation - Bevezető

A WPF (kódneve Avalon) a .NET 3.0 verziójában megjelent grafikus alrendszer. A WinForms –tól eltérően a WPF a WinAPI helyett közvetlenül a DirectX –xel kommunikál az ún. Media Integration Layer –en (MIL) keresztül, amely egy natív könyvtár. A WPF és a MIL közti menedzselt réteget (amelyet a .NET képes használni) PresentationCore –nak hívják, ez egyrészt hídat képez a .NET és a MIL közt, másrészt definiálja a WPF működését. Az i –re a pontot a PresentationFramework teszi fel, ami az animációkat, adatkötéseket, stb. definiálja.
A WPF a grafikus felület, az animációk, az adatkötések (stb…) létrehozásához bevezeti az ún. XAML (eXtensive Application Markup Language) nyelvet, amely az XML –en alapul. Az XAML segítségével deklaratív úton (vagyis hagyományos programozás nélkül) állíthatjuk be az egyes komponenesek kinézetét, viselkedését. Egyúttal azt is lehetővé teszi, hogy specifikus programozói készség nélkül lehessen definiálni a felületet, vagyis egy designer is elkészítheti azt.
Ugyanakkor nem az XAML az egyetlen útja a WPF programozásának, hiszen forráskódból is gond nélkül létrehozhatunk vezérlőket a már ismert úton (és persze az XAML is hagyományos CLR kódra fordul le).
Azt sem árt tudni, hogy az XAML –t bár a WPF vezeti be, mégsem kötődik hozzá szorosan, gyakorlatilag bármely környezet átveheti azt.

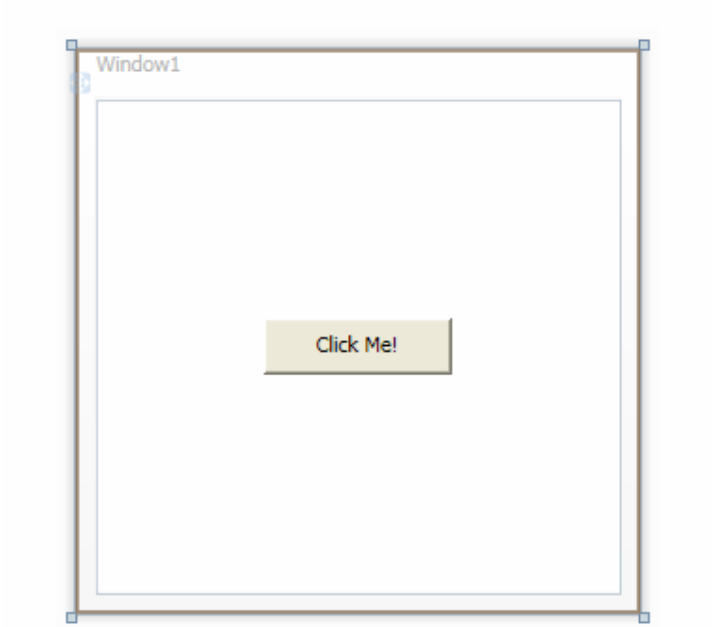
Készítsük el első WPF programunkat, ehhez már szükség van a Visual Studio –ra (lehetőleg 2008 legyen, akár Express is). File menü, New -> Project, a Windows fülnél válasszuk ki a WPF Application sablont, adjunk nevet a projectnek, elérési útat és OK. A következőt kell látnunk:



Láthatjuk az ablakot (illetve még nem, mivel nincs rajta semmi), alatta pedig ott a hozzá tartozó XAML kód. A feladat ugyanaz lesz, mint amit a Winforms –nál csimáltunk, tehát először szükségünk van egy gombra. Most még mindent kézzel írunk be, navigáljunk a két *Grid* tag közé és készítsük el. Az IntelliSense segítségével könnyű dolgunk lesz:

```
<Window x:Class="WPfDemo.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="300">
  <Grid>
    <Button x:Name="firstWPFButton"
            Width="100" Height="30"
            Content="Click Me!" Click="firstWPFButton_Click" />
  </Grid>
</Window>
```

A Click eseményhez tartozó kezelőt kérésre létrehozza nekünk. A gombnak a tervező nézetben is meg kell jelennie:



Most váltsunk át a Windows1.xaml.cs file –ra, itt már benne kell legyen az eseménykezelő (ha kértük, hogy készítse el):

```
private void firstWPFButton_Click(object sender, RoutedEventArgs e)
{

}
```

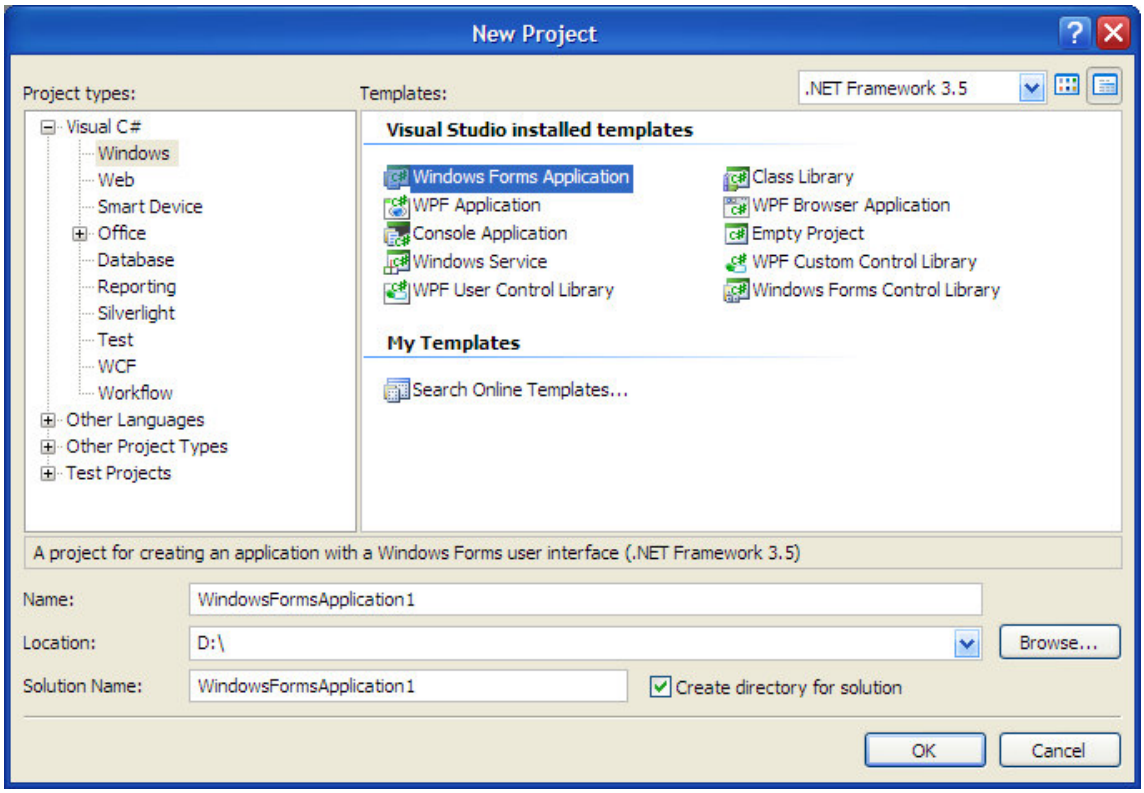
A MessageBox megjelenítése pontosan úgy zajlik, mint amit már láttunk, ennek a résznek a megírása az olvasó feladata. Futtatni az F5 gomb megnyomásával tudjuk a kész programot.

35. Windows Forms

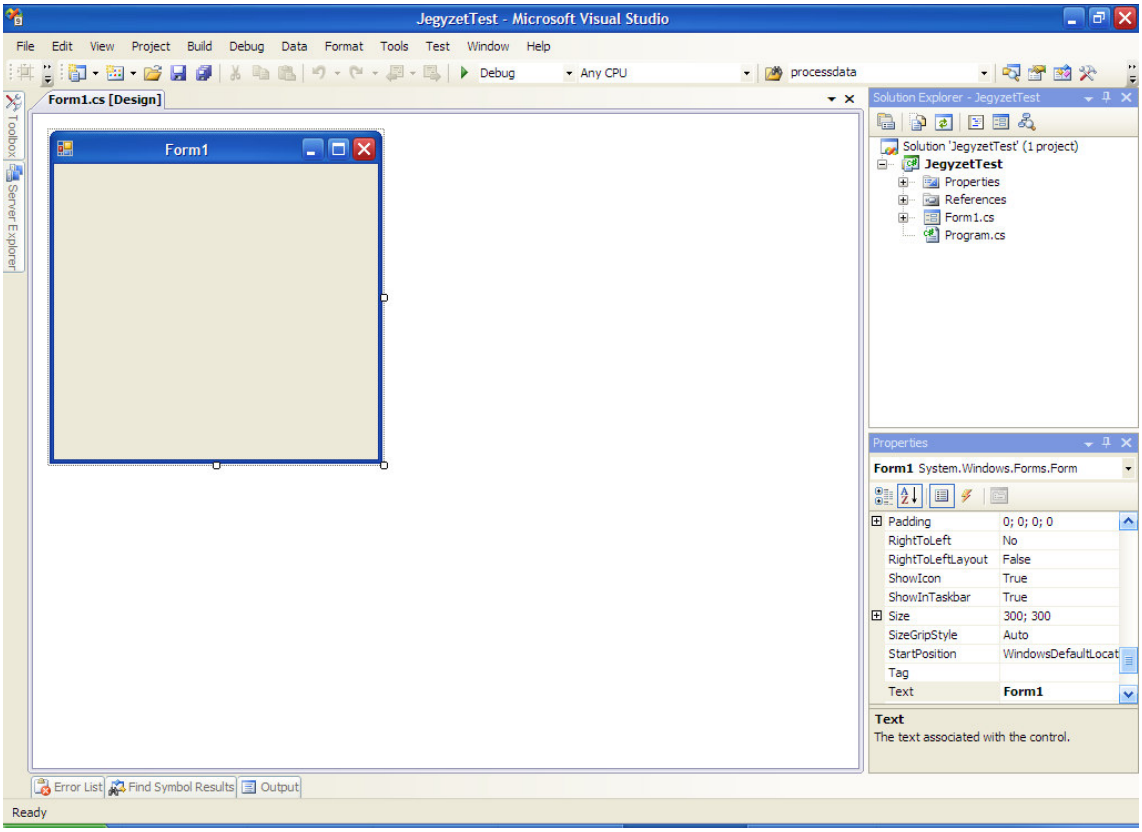
Az előző részben már elkészítettük első WF alkalmazásunkat, ebben a fejezetben pedig mélyebbre ássuk magunkat a témában. Ebben a részben megtanulunk bánni a Visual Studio –val, megismerkedünk a leggyakrabban használt vezérlőkkel, készítünk saját vezérlőt, végül pedig egy összetett alkalmazást fogunk elkészíteni.

A Windows Forms fejlesztéshez a leggyakrabban a System.Windows.Forms névteret fogjuk használni. A legtöbb osztály amit felhasználunk majd, a névtér *Controls* osztályából származik, így rengeteg olyan tulajdonság/esemény/stb… lehet, amelyet „közösen használnak”.

A következőkben már szükség lesz a Visual Studio –ra (2008 vagy 2005 (ebben nem használható a C# 3.0), Expressek is). File menü, New -> Project, itt a Visual C# fülnél válasszuk ki a Windows Forms Application sablont:



Adjunk nevet a projectnek, elérési utat, majd kattintsunk az OK gombra (a VS 2008 ezután biztonsági okokból felajánlja – alapértelmezés szerint, hogy csak „megnézni” nyissuk meg a projektet, ekkor válasszuk a „Load Project Normally” lehetőséget). Ezután a designer fog megjelenni, rajta az ablakkal:



A jobb fölső sarokban a Solution Explorer található, itt vannak a project file –jai. Alatta pedig egy sokkal érdekesebb dolog: a Properties ablak. Itt az adott vezérlő tulajdonságai vannak (ezek a valóságban is tulajdonságok). Az ablak felső sorában a vezérlő neve (*Form1*) és típusa (*System.Windows.Forms.Form*) van. Nézzük a legfontosabb tulajdonságokat, ezek a legtöbb vezérlő esetében használhatóak:

- *Name*: ez lesz a vezérlő neve, a programban ezzel az azonosítóval hivatkozunk majd rá.
- *Enabled*: azt jelöli, hogy a vezérlő aktív –e.
- *Location*: a vezérlő helyezete az öt tartalmazó vezérlőben, a bal fölső sarok koordinátái: (0;0).
- *Opacity*: átlátszóság, minél kisebb az érték annál átlátszóbb a vezérlő.
- *Size*: a vezérlő mérete.
- *Text*: a vezérlő felirata.

Egy *Form* esetében a *Location* értékei csak akkor érvényesek, ha a *StartPosition* tulajdonságot *Manual* –ra állítjuk.
Most nézzünk egy kicsit a dolgok mélyére. Kattintsunk a jobb egérgombbal a designerre és válasszuk ki a View Code –ot. Ekkor elénk tárul az ún. code-behind file. Ez nálam a következőképpen néz ki (VS 2008):

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

```
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace JegyzetTest
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }
   }
}
```

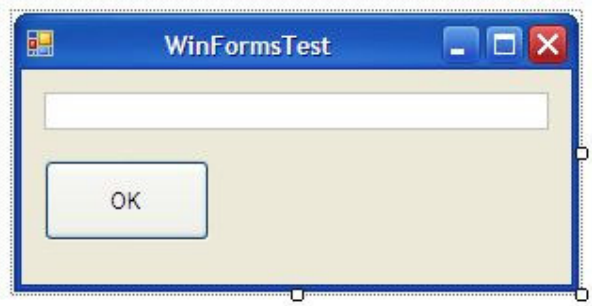
Két dolgot kell észrevennünk: az első, hogy az osztály a *Form* osztályból származik, a második pedig az, hogy ez egy parciális osztály, hogy ez miért van, azért nézzük meg a konstruktort. Ez egy *InitializeComponent* nevű metódust hív meg, a definícióját azonban itt nem látjuk. Ezt a metódust a Visual Studio automatikusan generálja, a feladata pedig az, hogy beállítsa a vezértlők kezdeti tulajdonságait. Nézzük meg, ehhez jobb klikk a metódushíváson és válasszuk ki a Go To Definition-t. Ha megnézzük a forráskódot akkor látható, hogy ez a parciális osztályunk másik fele, az *InitializeComponent* valahogy így kell kinézzen:

```
private void InitializeComponent()
{
   this.SuspendLayout();
   //
   // Form1
   //
   this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
   this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
   this.ClientSize = new System.Drawing.Size(292, 266);
   this.Name = "Form1";
   this.StartPosition = System.Windows.Forms.FormStartPosition.Manual;
   this.Text = "Form1";
   this.ResumeLayout(false);

}
```

Amit fontos megjegyeznünk, az az, hogy ezt a metódust ne nagyon piszkáljuk (minden ami itt beállításra kerül beállítható a Properties ablakban), ezt a Visual Studio fordításkor minden esetben újragenerálja, az esetleges általunk eszközölt módosítások elvesznek.

Most már készen állunk, hogy elkészítsük a második WF programunkat. Térjünk vissza a tervező nézethez. A jobb oldalon található a Toolbox, amely a felhasználható vezérlőket tartalmazza (alapértelmezés szerint ott kell lennie, ha nincs, akkor a View menüből válasszuk ki). Nyissuk ki az All Windows Forms feliratú fülecskét (ha még nincs nyitva), ezután húzzunk a formra egy TextBox –ot és egy Buttont. Előbbit nevezzük el (a Properties ablakban) MessageTextBox –nak, utóbbit pedig OkButton –nak. A gomb felirata legyen: „OK”. Végül a form feliratát is állítsuk

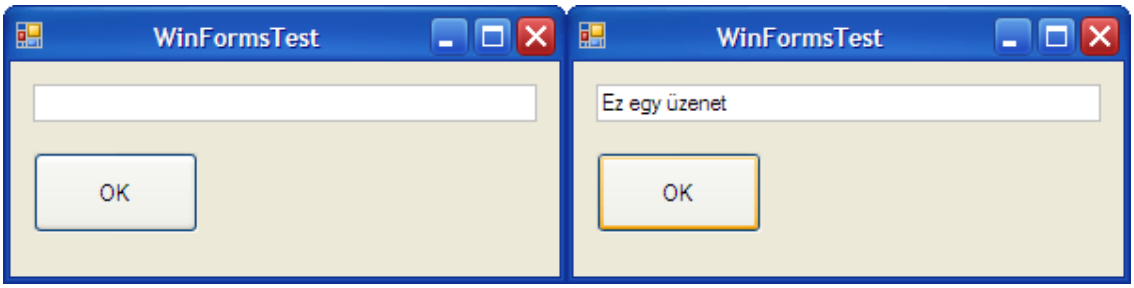
át, mondjuk WinFormsTest –re. Most valahogy így kell kinézzen (a formot átméreteztem):



A feladat a következő: rákattintunk a gombra, ekkor valamilyen üzenet jelenjen meg a Textbox –ban. Ehhez a feldathoz a gomb már ismert Click eseményét fogjuk használni. Az ehhez tartozó eseménykezelőt háromféleképpen tudjuk létrehozni, első és legegyszerűbb módszer, hogy megírjuk, ezután tervező nézetben válasszuk ki a gombot és a Properties ablakban kattintsunk a kis villámjelre, keressük meg a Click eseményt és írjuk be a kezelő nevét. A második módszer, hogy szintén a villámjelnél duplán kattintunk az eseméyn nevére, ekkor a VS lérehozza a metódust. A legutólsó lehetőségünk pedig az, hogy tervező nézetben duplán kattintunk egy vezérlőn, ekkor az adott vezérlő alapértelmezett eseményéhez (ez a gomb esetében a Click) létrejön a kezelő. Akárhogy is, a kezelő metódusba írjuk a következőt:

```
private void OkButton_Click(object sender, EventArgs e)
{
   MessageTextBox.Text = "Ez egy üzenet";
}
```

A *Text* tulajdonság egy string –et vár (vagy ad vissza, létezik setter is), amely a *TextBox* szövegét fogja beállítani.
A program futtatásához nyomjuk le az F5 gombot, ezután így fog működni:



36. Windows Forms – Vezérlők

Vezérlők alatt azokat az elemeket értjük, amelyekből egy grafikus alkalmazás felépül. Minden egyes vezérlőnek vannak egyedi tulajdonságai, eseményei, amelyek alkalmassá teszik őket egy adott feladatra. A következőkben megnézünk néhány vezérlőt.

Néhány vezérlő nem kap önálló fejezetet, mivel csak egy másik vezérlővel együtt játszik szerepet.

36.1 Form

Ez az osztály fogja képezni a felhasználói felület alapját. A *Form* osztály a *ContainerClass* osztályból származik, így tartalmazhat más vezérlőket is. Egy *Form* lehet modális ablak is, azaz olyan párbeszédablak amely megköveteli a felhasználótól, hogy használja, mielőtt visszatér az eredeti ablakhoz (ilyen pl. a *MessageBox* is). Ezt a hatást a S*howDialog* metódussal tudjuk elérni. Helyezzünk egy formra egy gombot, és a *Click* eseményében jelenítsünk meg egy modális ablakot:

```
private void button1_Click(object sender, EventArgs e)
{
   Form modalForm = new Form();
   modalForm.ShowDialog();
}
```

Alapértelmezetten egy form címsorában megjelennek a tálcára helyezéshez, illetve a teljes méretre nagyításnak a gombjai. Ezeket a *MinimizeBox* illetve a *MaximizeBox* tulajdonságokkal tudjuk tetszés szerint megjeleníteni/elrejteni. Az ablak méretezésének letiltását a *FormBorderStyle* tulajdonság valamelyik *Fixed* előtaggal rendeljező értékével tudjuk jelezni.
A *Form* alapértelmezett eseménye a *Load*, amely akkor fut le mielőtt a *Form* először megjelenik.

Mielőtt és miután a *Form* és gyermekei megjelennek számos esemény fut le. Ezeket összefoglaló néven a Form életciklusának nevezzük. Ennek az állomásai a következőek:

- HandleCreated: egy kezelő jön létre a formhoz, ez gyakorlatilag egy hivatkozás
- Load: a form megjelenése előtt fut le, ez az a hely ahol a legcélszerűbb a form által használt erőforrások lefoglalása, illetve az adattagok kezdeti beállítása.
- Shown esemény: az első alkalommal, amikor a form megjelenik fut le.
- Activated esemény: a form és vezérlői megjelennek és a formrakerül a fókusz (vagyis ő van az előtérben.
- Closing: akkor fut le, amikor bezárjuk az ablakot.
- Closed: miután bezártuk az ablakot fut le.
- Deactivated: az ablakon nincs rajta a fókusz.
- HandleDestroyes: az ablakhoz tartozó kezelő megsemmisül.

Készítsünk egy programot ezeknek a folyamatoknak a szemléltetésére! A Formunk egyik adattagja legyen egy Form, a fő-Formra pedig húzzunk egy gombot.

```
public partial class Form1 : Form
{
   Form form = null;

   public Form1()
   {
      InitializeComponent();
   }
}
```

A *Click* eseményéhez pedig írjuk ezt:

```
private void button1_Click(object sender, EventArgs e)
{
   this.form = new Form();
   form.HandleCreated += new EventHandler(form_HandleCreated);
   form.Load += new EventHandler(form_Load);
   form.Activated += new EventHandler(form_Activated);
   form.Shown += new EventHandler(form_Shown);
   form.Show();
}
```

Az egyes eseményeket módosítsuk úgy, hogy lássuk mi történik (pl. egy MessageBox –ban egy üzenet, vagy a fő – formon egy ListBox, stb…).

Az eseménykezelők hozzáadásánál a += operátor után nyomjunk dupla TAB –ot, ekkor automaikusan kiegészíti a sort és létrehozza a kezelőt is.

Egy Form –hoz kétféleképpen adhatunk hozzá vezérlőt (tulajdonképpen a kettő egy és ugyanaz), vagy a tervezőt használjuk és egyszerűen ráhúzzuk, vagy pedig futás közben dinamikusan. Utóbbihoz tudni kell, hogy minden *ContainerClass* leszármazott rendelkezik egy *Controls* nevű tulajdonsággal, amely egy *ControlCollection* típusú objektumot ad vissza. A *ControlCollection* megvalósítja az *ICollection* és *IEnumerable* interfészeket is, vagyis tömbként kezelhető, valamint foreach ciklussal is használható. Ehhez tudjuk majd hozzáadni a vezérlőket:

```
Button b = new Button();
b.Size = new Size(100, 30);
b.Location = new Point(10, 10);
b.Text = "Gomb";
this.Controls.Add(b);
```

Itt a this egy formra vonatkozik. A *Controls* tulajdonság használható foreach ciklussal.

36.2 Button

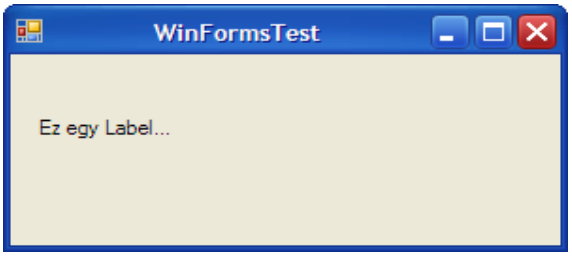
A gombot már ismerjük, elsődlegesen arra használjuk, hogy elindítsunk valamilyen folyamatot (pl. adatok elküldése, beolvasása, stb.). Alapértelmezett eseménye a kattintás.

36.3 Label

Szöveget jelenítünk meg vele, amelyet a felhasználó nem módosíthat. A leggyakrabban iránymutatásra használjuk. Értéket a Text tulajdonságán keresztül adhatunk neki, a példánkban ezt a form konstrukorában tesszük meg:

```
public partial class Form1 : Form
{
   public Form1()
   {
      InitializeComponent();

      label1.Text = "Ez egy Label...";
   }
}
```

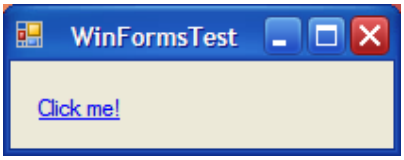
És az eredmény:



A *Label* egy specializált változata a *LinkLabel*, amely hasonlóan működik mint egy weboldalon található hivatkozás. Ennek alapértelmezett eseménye a *LinkClicked*, ekkor kell átirányítanunk a felhasználót a megfelelő oldalra. A *LinkLabel* szövegét szintén a *Text* tulajdonsággal tudjuk beállítani:

```
private void linkLabel1_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
{
   System.Diagnostics.Process.Start("http://www.microsoft.com");
}
```

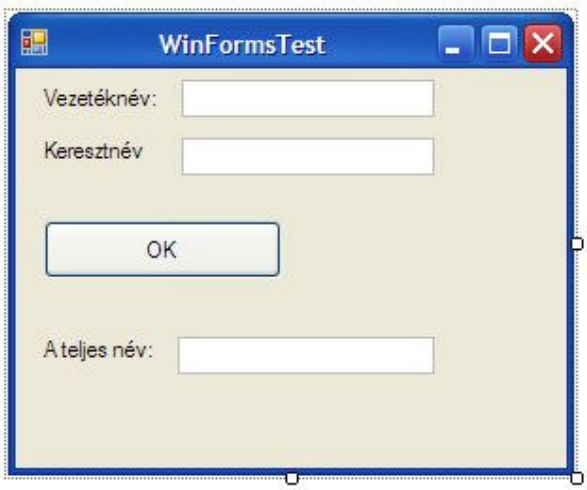
A Form pedig így néz ki:



36.4 TextBox

Szintén egy szöveges vezérlő, de ez már képes fogadni a felhasználótól érkező adatokat. Készítsünk egy programot, amely felhasználja az eddig megismert három

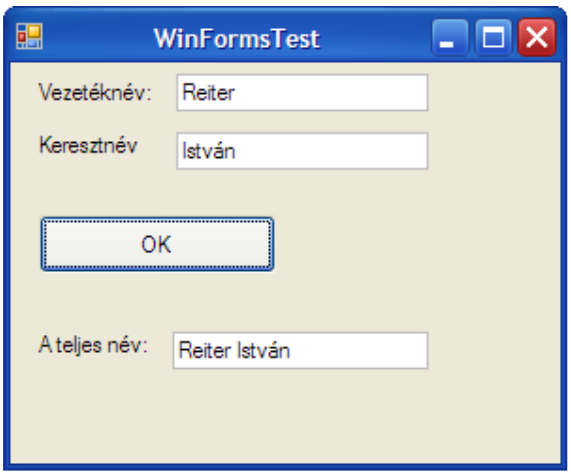
vezérlőt. Kérjük be a felhasználó kereszt- és vezetéknevét külön *TextBox* –ban, majd egy harmadikba írjuk ki a teljes nevet, amikor lenyomunk egy gombot. Húzzunk a formra három *TextBox* -ot, három *Label* –t és egy *Button* –t. A *TextBox* –ok nevei legyenek pl. LastNameText, FirstNameText és FullNameText, a gombot pedig nevezzük OKButton –nak. Most valahogy így kellene kinéznie a formnak:



Kezeljük a gomb kattintás eseményét:

```
private void OKButton_Click(object sender, EventArgs e)
{
   if (LastNameText.Text == "" || FirstNameText.Text == "")
   {
      MessageBox.Show("Kérem töltse ki az összes mezőt!");
   }
   else
   {
      FullNameText.Text = LastNameText.Text + " " + FirstNameText.Text;
   }
}
```

Egy kis hibakezelést is belevittünk a dologba. Most már működnie kell a programnak, méghozzá így:



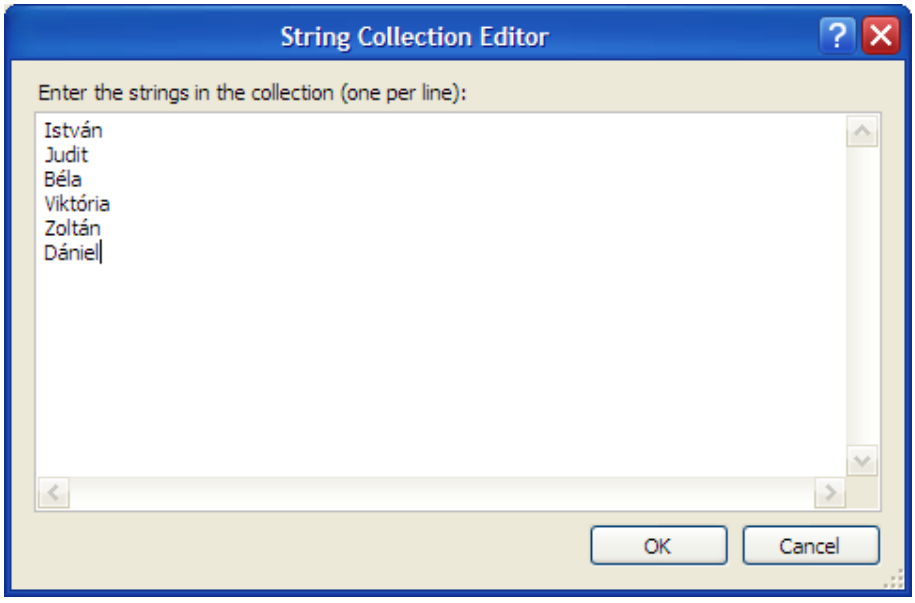
A *TextBox MultiLine* tulajdonságát igazra állítva több sorba is írhatunk. Egy másik érdekes tulajdonság a *PasswordChar*, amelynek megadva a kívánt karaktert ezután minden beírt karakter helyett a megadott karaktert jeleníti meg, azaz úgy viselkedik, mint egy jelszómező:

A *TextBox* alapértelmezett eseménye a *TextChanged*, ez akkor váltódik ki, amikor a *TextBox* tartalma megváltozik. Készítsünk egy programot ennek szemléltetésére! Legyen a formunkon két *TextBox*, ha az elsőben írunk valamit, az jelenjen meg a másikban. Legyen a két *TextBox* neve FirstTextBox és SecondTextBox. Generáljunk a FirstTextBox –hoz egy *TextChanged* eseménykezelőt:

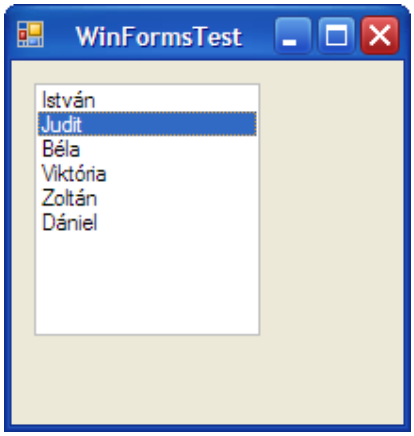
```
private void FirstTextBox_TextChanged(object sender, EventArgs e)
{
   SecondTextBox.Text = FirstTextBox.Text;
}
```

36.5 ListBox

A ListBox egy olyan vezérlő, amelyben adatokat tárolunk, jellemzően egy lista stílusában. Egy ListBox –ot kétféleképpen tölthetünk fel adatokkal, kezdjük az egyszerűbbel. A Properties ablakban keressük meg az Items tulajdonságot, kattintsunk a pontocskákra és írjuk be az adatokat, egy sorba egyet:



Ha futtatjuk a programot, akkor az eredmény a következő lesz:



Ennek a módszernek a hátránya az, hogy előre tudnunk kell, hogy milyen adatokat akarunk. A dinamikus feltöltéshez bevezetjük az adatkötések (databinding) fogalmát. Ez tulajdonképpen annyit tesz, hogy néhány (adatokat tároló) vezérlőnek megadhatunk adatforrásokat (egy tömb, egy lekérdezés eredménye, stb…), amelyekből feltölti magát.
Írjuk át a fenti programot, hogy adatkötést használjon. A *ListBoxot* a *Form Load* eseményében fogjuk feltölteni, tehát először generáljuk le azt, majd írjuk:

```
private void Form1_Load(object sender, EventArgs e)
{
   string[] t = new string[]
   {
      "István", "Judit", "Béla", "Viktória", "Dániel"
   };

   listBox1.DataSource = t;
}
```

Mi van azonban akkor, ha nem stringeket, hanem egy saját osztály példányait akarom megjeleníteni:

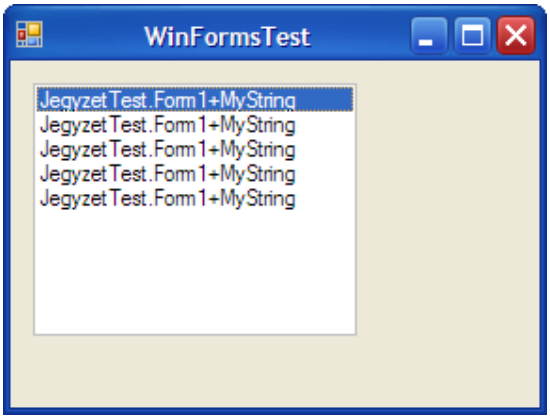
```
class MyString
{
   public string data;

   public MyString(string s)
   {
      data = s;
   }

   public string Data
   {
      get { return data; }
   }
}
```

Ha egy ilyen elemekből álló tömböt kötünk a *ListBox* –ra, akkor vajon mi történik?

```
private void Form1_Load(object sender, EventArgs e)
```

```
{
   MyString[] t = new MyString[]
   {
      new MyString("István"),
      new MyString("Judit"),
      new MyString("Béla"),
      new MyString("Viktória"),
      new MyString("Dániel")
   };

   listBox1.DataSource = t;
}
```

Hát ez:



A *ListBox* a rákötött elemeinek a *ToString()* metódus által nyert adatát jeleníti meg, mi viszont nem terheltük túl ezt a metódust, ezért az eredeti változat hívódik meg, ami pont a fent látható csúfságot adja vissza. A megoldás tehát az, hogy elkészítjük a metódust a saját osztályunkhoz is:

```
class MyString
{
   public string data;

   public MyString(string s)
   {
      data = s;
   }

   public override string ToString()
   {
      return data;
   }
}
```

Most már kifogástalanul kell működnie.

A *ListBox* alapértelmezett eseménye a *SelectedIndexChanged*, amelyet az vált ki, ha megváltozik a kiválasztott elem. Módosítsuk a programot, hogy a kiválasztott elemre kattintva jelenjen meg egy *MessageBox* a megfelelő szöveggel:

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
   ListBox lb = (ListBox)sender;
   MessageBox.Show("A kiválasztott elem: " + lb.SelectedItem.ToString());
}
```

A küldő objektumon explicit típuskonverziót hajtottunk végre, hogy hozzáférjünk az eredeti adatokhoz. Fontos tudni, hogy ilyenkor nem jön létre új objektum, csak egy már létezőre mutató referencia.
A *ListBox* elemeihez az *Items* tulajdonságon keresztül is hozzáférhetünk, ez egy gyűjteményt ad vissza, vagyis használhatjuk az indexelő operátor, illetve az *Add* metódust. Az *Items* gyűjtemény megcalósítja az *IEnumerable* és *IEnumerator* interfészeket így foreach ciklust is használhatunk.
Házi feladat következik: húzzunk a formra egy másik *ListBox* -ot és a form *Load* eseménykezelőjében töltsük fel ezt a vezérlőt is a másik lista elemeivel. A használandó eszközök: *Items* tulajdonság, foreach ciklus.

A *ListBox SelectionMode* tulajdonságát megváltoztatva több elemet is kijelölhetünk egyszerre. A *SelectionMode* egy enum típus, értékei lehetnek *None* (nem lehet kiválasztani egy elemet sem), *MultiSimple* (a Ctrl billentyű nyomva tartásával lehet több elemet kiválasztani) illetve *MultiExtended* (az egér nyomva tartása mellett lehet „kijelölni” az elemeket).

A Sorted tulajdonság beállításával a *ListBox* elemei ABC sorrendbe rendeződnek.

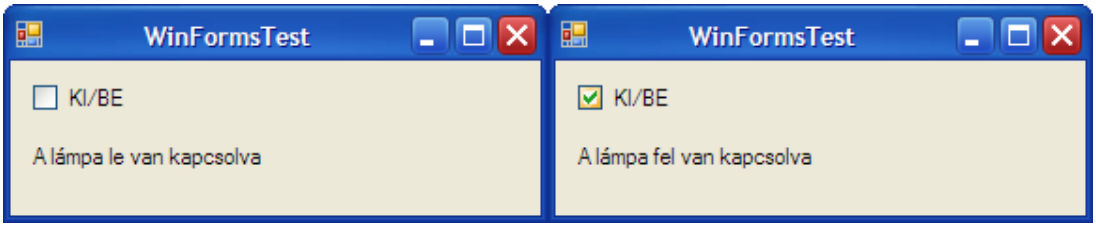
A *ListBox BeginUpdate* és *EndUpdate* metódusai segítségével anélkül tölthetjük fel a lsitát futásidőben, hogy minden egyes alkalommal újrarajzolná azt:

```
listBox1.BeginUpdate();

for(int i = 0;i < 100;++i)
{
     listBox1.Items.Add(i.ToString());
}

listBox1.EndUpdate();
```

Ez a metódus a hasonló vezérlőknél (CheckedListBox, ComboBox, …) is használható.

36.6 CheckBox

A *CheckBox* vezérlő a választás lehetőségét kínálja fel. Bizonyára mindenki találkozott már olyan beléptetőrendszerrel, amely felajánlja, hogy legközelebb felismer minket és azonnal beléptet. Nos, azt a választást egy *CheckBox* –xal implementálták (természetesen ilyen más környezetekben is van).
A *CheckBox* alapértelmezett eseménye a *CheckedChanged*, ezt a bejelölés/visszavonás váltja ki. Készítsünk egy egyszerű programot, amellyel egy lámpát szimulálunk. A felhasználói felület nagyon egyszerű lesz:



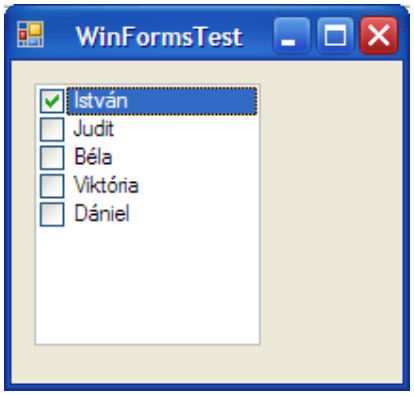
Az eseménykezelő pedig a következő:

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
   if (checkBox1.Checked)
   {
      label1.Text = "A lámpa fel van kapcsolva";
   }
   else label1.Text = "A lámpa le van kapcsolva";
}
```

36.7 CheckedListBox

Ez a vezérlő a *ListBox* és a *CheckBox* házasságából született, az elemeit egy – egy *CheckBox* –xal lehet kijelölni. Ennél a vezérlőnél nem használhatunk adatkötést, egyszerre több elemet az *Items* tulajdonságának *AddRange* metódusával tudunk átadni neki (illetve egy elemet az *Add* metódussal):

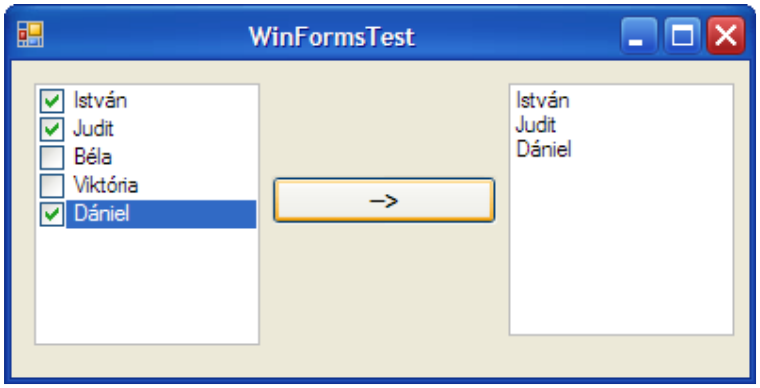
```
private void Form1_Load(object sender, EventArgs e)
{
   string[] t = new string[]
   {
      "István", "Judit", "Béla", "Viktória", "Dániel"
   };

   checkedListBox1.Items.AddRange(t);
}
```



Alapértelmezetten dupla kattintással tudunk megjelölni egy elemet, ezt a *CheckOnClick* tulajdonság *True* értékre állításával szimpla kattintásra változtathatjuk. A kiválasztott elemekhez a *CheckedItems* tulajdonságon keresztűl férhetünk hozzá, ez egy gyűjteményt ad vissza, amelyen használhatjuk pl. a foreach ciklus.

Készítsünk egy programot, amely tartalmaz egy *CheckedListBox* –ot, egy gombot és egy *ListBox* –ot. A gomb megnyomásakor a kiválasztott elemek jelenjenek meg a *ListBox* –ban. A gomb *Click* eseményében fogjuk megoldani:

```
private void button1_Click(object sender, EventArgs e)
{
   foreach (object s in checkedListBox1.CheckedItems)
   {
      listBox1.Items.Add(s);
   }
}
```



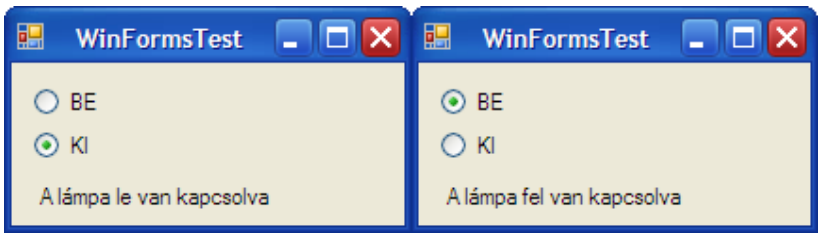
Hasonlóan a *ListBox* –hoz, a *CheckedListBox* alapértelmezett eseménye is a *SelectedIndexChanged.*

36.8 RadioButton

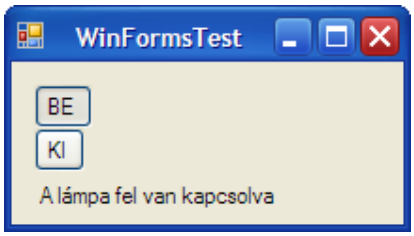
A *RadioButton* hasonló funkcionalitást kínál, mint a *CheckBox*, a különbség annyi, hogy minden *RadioButton* egy adott tárolón belül (pl. egy form) egy csoportot alkot, közülük pedig egy időben csak egyet lehet kiválasztani.
Vegyük elő a jól bevált lámpakapcsoló programunkat és alakítsuk át, hogy használja ezt a vezérlőt. A *RadioButton* alapértelmezett eseménye a *CheckedChanged*, ezért erre fogunk ráülni. A két *RadioButton* használja ugyanazt az eseményt (mindkettő kijelöl, utána *Properties* ablak villámjel és ott állítsuk be):

```
private void radioButton2_CheckedChanged(object sender, EventArgs e)
{
   RadioButton rb = (RadioButton)sender;
   if (rb.Name == "radioButton1")
   {
      label1.Text = "A lámpa fel van kapcsolva";
   }
   else label1.Text = "A lámpa le van kapcsolva";
}
```

A form *Load* eseményében is állítsuk be a Label kezdeti szövegét.
A *RadioButton* –nak tervezési időben megadhatjuk, hogy be legyen –e jelölve, ehhez a *Checked* tulajdonságot kell megváltoztatni (itt is érvényes az egy időben egy lehet bejelölve szabály).
A lámpakapcsolgató program:



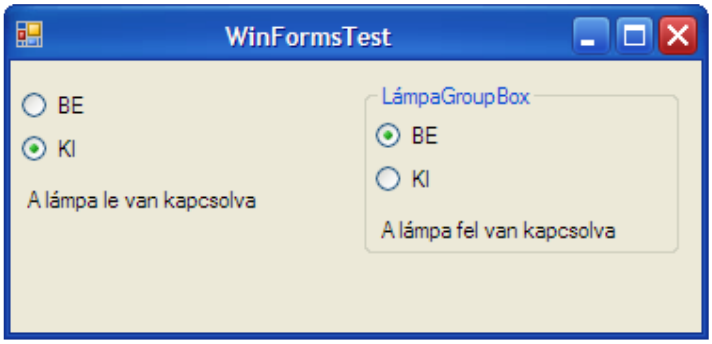
A megjelenését átállíthatjuk gombszerűvé az *Appearance* tulajdonsággal:



Nyílván egy formon belül előfordulhatnak különálló *RadioButton* csoportok, ekkor, hogy ne zavarják egymást el kell különíteni őket valahogy. Erre a célra két vezérlőt használhatunk, a *Panel* –t és a *GroupBox* –ot. A kettő közt a különbségek a következők: a *Panel* scrollozható, nincs felirata sem állandó kerete, a *GroupBox* pedig nem scrollozható van felirata és kerete is.
Alakítsuk át a progrmunkat, hogy két külön lámpa legyen rajta, az egyik csoport egy *Panel* –ban legyen a másik egy *GroupBox* –ban. A négy RadioButton ugyanazt az eseménykezelőt fogja használni, így meg kell tudnunk különböztetni, hogy melyik csapat küldte a jelzést. Nincs nehéz dolgunk, mivel minden vezérlő rendelkezik egy *Parent* nevű tulajdonsággal, amely az őt tartalmazó vezérlőt adja vissza. Egy egyszerű típuskonverzióval fogjuk eldönteni a kérdést:

```
private void radioButton2_CheckedChanged(object sender, EventArgs e)
{
   RadioButton rb = (RadioButton)sender;

   Panel p = rb.Parent as Panel;

   if (p == null)
   {
      if (rb.Name == "radioButton3")
      {
         label2.Text = "A lámpa fel van kapcsolva";
      }
      else label2.Text = "A lámpa le van kapcsolva";
   }
   else
   {
      if (rb.Name == "radioButton1")
      {
         label1.Text = "A lámpa fel van kapcsolva";
      }
      else label1.Text = "A lámpa le van kapcsolva";
   }
}
```

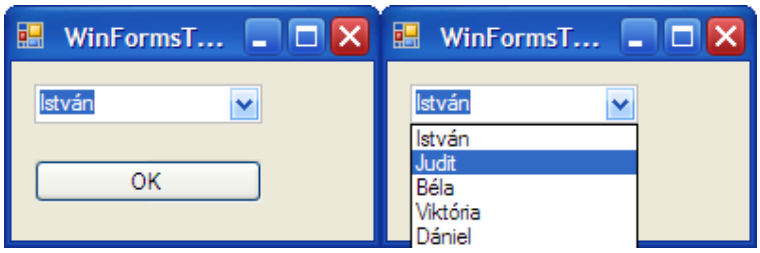
Kihasználtuk azt, hogy az *as* operátor nullértéket ad vissza, ha a konverzió sikertelen volt, így biztosan tudjuk, hogy melyik csapattól érkezett a jelzés.
Ugyanakkor meg kell jegyeznünk, hogy a szöveg módosítása nem igazán elegáns, ezért az olvasó feladata lesz, hogy egy szebb megoldást találjon.
A program most már szépen fut:



A *Panel* –nek alapértelmezetten nincs körvonala, de ezt beállíthatjuk a *BorderStyle* tulajdonsággal.

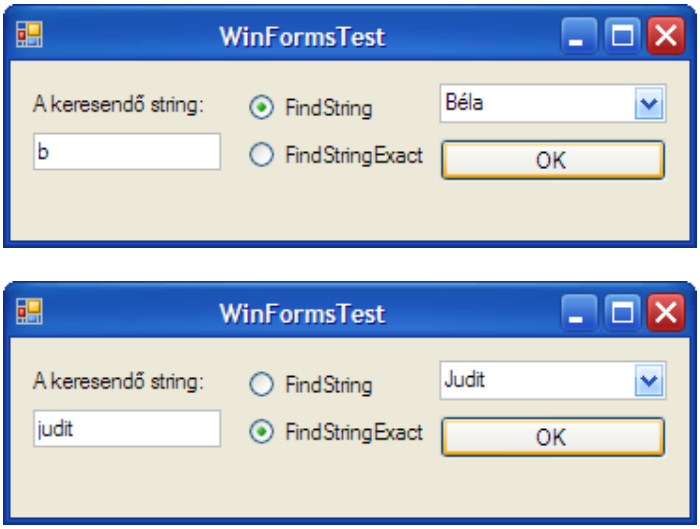
36.9 ComboBox

A ComboBox a TextBox és a ListBox keresztezéséből jött létre, a beírt szöveg alapján kiválaszthatunk egy elemet egy előre megadott listából. A ComboBox –xal használhatunk adatkötést is. Készítsünk egy programot, amely egy ComboBox –ból és egy gombból áll, a gombot megnyomva megjelenik egy MessageBox, amely a kiválasztott elemet jeleníti meg.



A *ComboBox* –ot a már ismert módon töltöttük fel a form *Load* eseményében.

A *FindString* és *FindExactString* metódusokkal a *ComboBox* elemei között kereshetünk. Előbbi a legelső olyan elem indexét adja vissza, amely a paramétereként megadott stringgel kezdődik, utóbbi pedig pontos egyezést keres.
Készítsünk egy alkalmazást ennek szemléltetésére! Húzzunk a fomra egy *TextBoxot*, ez fogja tartalmazni a keresendő szöveget, illetve két *RadioButton* –t, amelyek a használandó metódusokat jelölik. A gomb funkcionalitása most megváltozik, a keresést fogja elindítani:



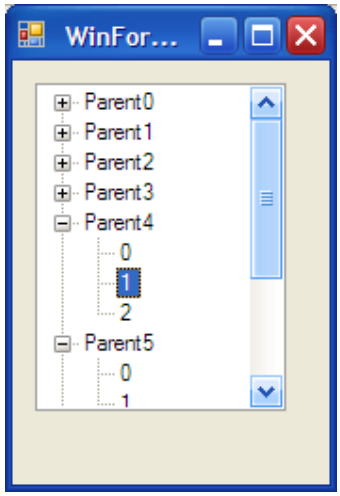
A *Click* esemény kódja pedig ez lesz:

```
private void button1_Click(object sender, EventArgs e)
{
   comboBox1.SelectedIndex =
      radioButton1.Checked == true
      ?
      comboBox1.FindString(textBox1.Text)
      :
      comboBox1.FindStringExact(textBox1.Text);
}
```

Ezek a metódusok a többi hasonló vezérlővel (*ListBox*, stb…) is használhatóak.

36.10 TreeView

A TreeView vezérlő adatok hierarchikus megjelenítésére szolgál (pl. egy filerendszer elemei). A fa egyes elemei *TreeNode* típusuak.



A vezérlőt a form *Load* eseményében töltöttük fel adatokkal:

```
private void Form1_Load(object sender, EventArgs e)
{
   treeView1.BeginUpdate();

   for (int i = 0; i < 10; ++i)
   {
      treeView1.Nodes.Add("Parent" + i.ToString());
      for (int j = 0; j < 3; ++j)
      {
         treeView1.Nodes[i].Nodes.Add(j.ToString());
      }
   }

   treeView1.EndUpdate();
}
```

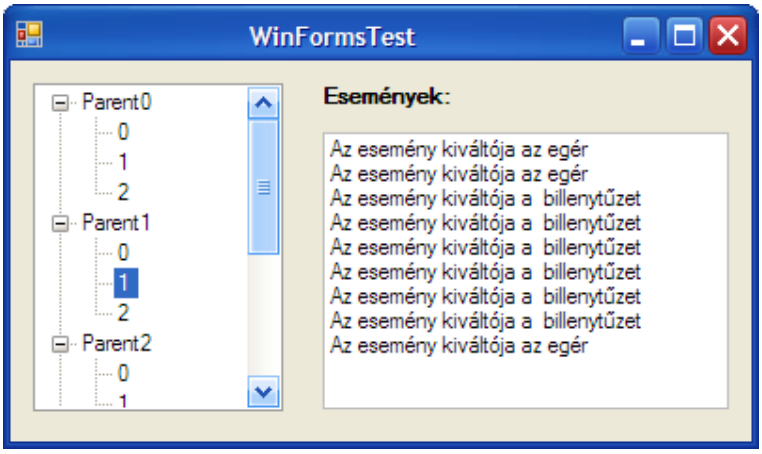
A *TreeView* alapértelmezett eseménye az *AfterSelect*, ez akkor fut le, ha valamelyik *Node* –ot kiválasztjuk. Az eseménykezelő a következőképpen néz ki:

```
private void treeView1_AfterSelect(object sender, TreeViewEventArgs e)
{

}
```

A *TreeViewEventArgs* a kiválasztás módjáról ad tájékoztatást. Készítsünk egy programot, amely ezekről az eseményekről ad információt. Húzzunk a formra egy *ListBoxot*, ebben fogjuk az eseményeket megjeleníteni. Ezután az eseménykezelő kódja:

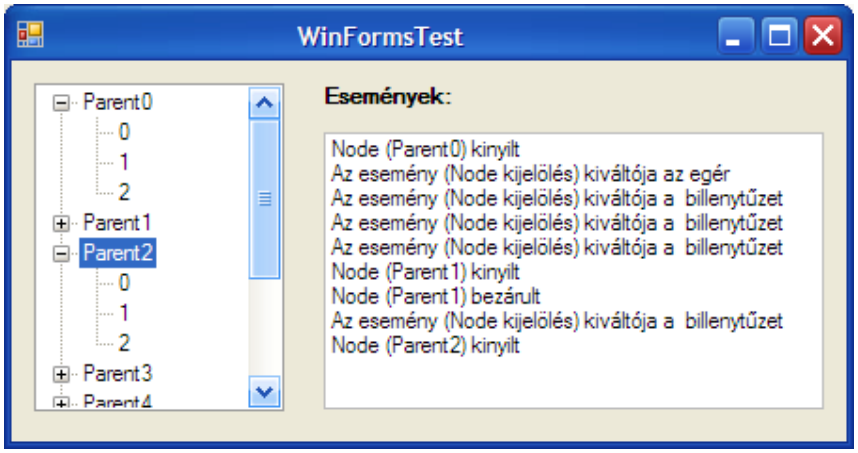
```
private void treeView1_AfterSelect(object sender, TreeViewEventArgs e)
{
   switch (e.Action)
   {
      case TreeViewAction.ByKeyboard:
         listBox1.Items.Add("Az esemény kiváltója a  billenytűzet");
         break;
      case TreeViewAction.ByMouse:
         listBox1.Items.Add("Az esemény kiváltója az egér");
         break;
      default:
         break;
   }
}
```

A program pedig:



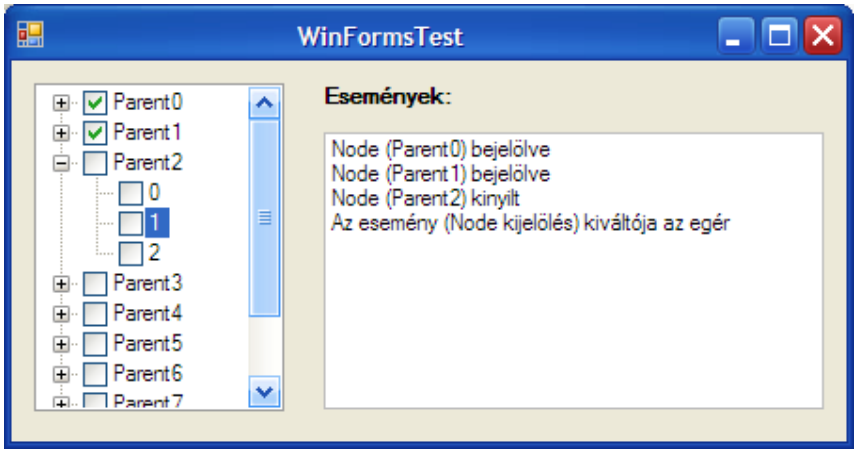
Ez az eseménykezelő azonban csak a *Node* kiválasztását figyeli, de mi esetleg szeretnénk a kinyit/becsuk eseményre is feliratkozni, ezek az *AfterExpand* és *AfterCollapse* események:

```
private void treeView1_AfterCollapse(object sender, TreeViewEventArgs e)
{
   listBox1.Items.Add("Node (" + e.Node.Text + ") bezárult");
}

private void treeView1_AfterExpand(object sender, TreeViewEventArgs e)
{
   listBox1.Items.Add("Node (" + e.Node.Text + ") kinyilt");
}
```



A *CheckBoxes* tulajdonság bejelölésével minden egyes *Node* –hoz egy *CheckBox* is megjelenik. A bejelölést az *AfterCheck* eseménnyel tudjuk kezelni:

```
private void treeView1_AfterCheck(object sender, TreeViewEventArgs e)
{
   listBox1.Items.Add("Node (" + e.Node.Text + ") bejelölve");
}
```



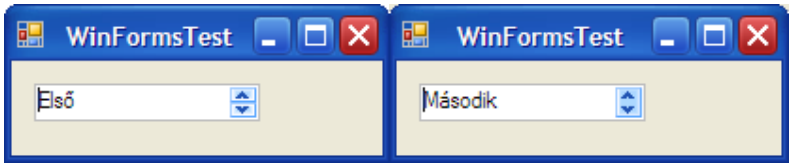
Egy *Node* szülőjére a *Parent* tulajdonságával tudunk hivatkozni. A *TreeView Nodes* tulajdonsága egy *TreeNodeCollection* típusú gyűjteményt ad vissza, amely megvalósítja az *IEnumerable* és *IEnumerator* interfészeket, azaz végigiterálható egy foreach ciklussal.
Házi feladat 1.: gombnyomásra listázzuk ki (egy másik *ListBox* –ban, *TextBox* –ban, akárhol) a bejelölt *Node* –ok nevét és szülőjét (a gyökérelemek *Parent* tulajdonsága null –t ad vissza, mivel nekik nincs szülőjük). A bejelölést a *Node Checked* tulajdonságával ellenőrizhetjük.
Házi feladat 2.: egy *Node* bejelölése nem jár együtt a gyermekei bejelölésével. Készítsünk programot, amely ezt megoldja (tipp: a T*reeView* –on meghívott *Nodes[x]* tulajdonság az x –edik gyökérelemet fogja visszaadni, ennek pedig van *Nodes* tulajdonsága, tipp2: akár az egész lerendezhető két egymásba ágyazott foreach ciklussal (kulcsszó: *TreeNodeCollection*)).

36.11 DomainUpDown

A *DomainUpDown* egy stringérték megjelenítésére szolgál, amelyet egy listából választhatunk ki.

```
private void Form1_Load(object sender, EventArgs e)
{
   domainUpDown1.Items.Add("Első");
   domainUpDown1.Items.Add("Második");
   domainUpDown1.Items.Add("Harmadik");
}
```

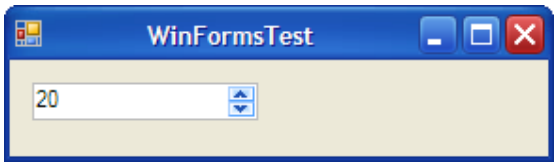
A *Text* tulajdonságot nem árt kitisztítani, vagy tetszés szerint beállítani, ellenkező esetben indításnál a *domainUpDown1* felirat jelenik meg.



A *Wrap* tulajdonság igaz értékre állításával ha elértük a lista végét akkor a következő elemnél automatikusan a lista elejére ugrik.

36.12 NumericUpDown

Hasonló mint az előző vezérlő, de ezúttal számokat jelenítünk meg. A Minimum és Maximum tulajdonságokkal a legkisebb és legnagyobb lehetséges értéket adjuk meg, míg az Increment tulajdonság a lépték méretét adja meg.



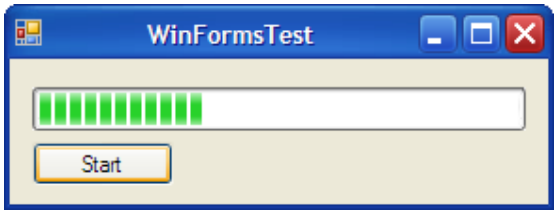
Alapértelmezett eseménye a ValueChanged, amely akkor indul el, amikor megváltozik a mutatott érték:

```
private void numericUpDown1_ValueChanged(object sender, EventArgs e)
{
   MessageBox.Show(((NumericUpDown)sender).Value.ToString());
}
```

Ebben az esetben a típuskonverzióhoz nem hoztunk létre új azonosítót, a megfelelő zárójelezéssel is elérhetőek a vezérlő adatai.

36.13 ProgressBar

A ProgressBar vezérlőt általában egy folyamat állapotának jelölésére használjuk:



A *Minimum* és *Maximum* tulajdonságai a felvehető értékek intervallumát jelölik ki, ezek alapértelmezetten 0 és 100. A *Style* tulajdonsággal a vezérlő kinézete (blokkos, egy növekvő téglalap illetve egyetlen mozgó blokk) állítható be.
A fenti program elszámol százig és ezt a folyamatot egy *ProgressBar* –on jelzi. A gomb *Click* eseményében indítjuk el a folyamatot:

```
private void button1_Click(object sender, EventArgs e)
{
   for (int i = 0; i < 100; ++i)
   {
      progressBar1.PerformStep();
      System.Threading.Thread.Sleep(100);
   }

   MessageBox.Show("100%");
}
```

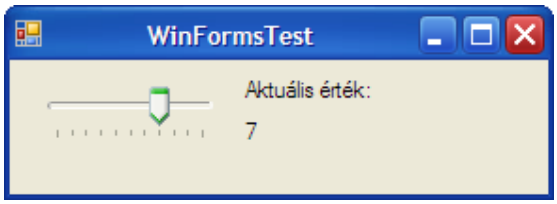
Minden lépésnél pihentetjük kicsit a főszálat, hogy látványosabb legyen az eredmény és a folyamat végét egy *MessageBox* –xal jelezzük. A *ProgressBar* léptetését a *PerformStep* metódussal végeztük el, amely a Step tulajdonság értéke szerint lépteti az aktuális értéket (a *Step* alapértelmezetten 1 –re van állítva). Az értéket kézzel is állíthatjuk a *Value* tulajdonsággal:

```
progressBar1.Value += 10;
```

Természetesen ekkor is előre/vissza lép a vezérlő.

36.14 TrackBar

Ezzel a vezérlővel a felhasználó ki tud választani egy értéket egy intervallumból. A *Minimum* és *Maximum* tulajdonságokkal az alsó és felső határt lehet állítani, a TickFrequency a vezérlő beosztását módosítja, a *Large*- illetve *SmallChange* pedig azt az értéket, amikor az egérrel vagy a billentyűzettek módosítjuk az aktuális értéket:



A programban a *TrackBar* alapértelmezett eseményére ültünk rá, ez a Scroll, amely akkor aktiválódik, amikor megváltozik a vezérlő értéke (vagyis elmozdul a mutató):

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
   label2.Text = ((TrackBar)sender).Value.ToString();
}
```

Az *Orientation* tulajdonsággal beállíthatjuk, hogy a vezérlő vízszintesen (*Horizontal*) vagy függőlegesen (*Vertical*) jelenjen meg.

36.15 PictureBox

Ez a vezérlő – ahogyan azt a neve is mutatja – képek megjelenítésére illetve grafikus feladatokra való. Az Image tulajdonságát kiválasztva egy ablak jelenik meg és két lehetőséget kínál nekünk. Ha a képet egy resource állomány tartalmazza, akkor a kép belefordul a futtatható állományba (vagy osztálykönyvtárba), csakis a programon belül érhetjük el. Alapértelmezetten nincs kép hozzárendelve a programhoz, ezt az Import gombra kattintva tehetjük meg. A fordítás után a Solution Explorerben megjeleni egy Resources.resx nevű tag, ez fogja tartalmazni az erőforrásokat (most a képet). Ennek a file –nak is vannak beállítási, válasszuk ki és nyissuk meg a Properties ablakot. Most be tudjuk állítani, hogy hogyan kezelje az erőforrásokat (beágyazva, átmásolva a saját könyvtárba, stb…).
A másik lehetőség, hogy a Local Resource –t választjuk, ekkor a kép csak simán belefordul a kimenetbe.

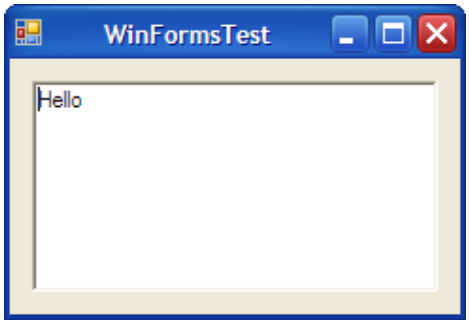
Természetesen arra is van lehetőség, hogy a forráskódból állítsuk be a képet, ekkor a kép elérési útját kell megadnunk (ha a futtatható állománnyal egy helyen van, akkor csak a nevét):

```
private void Form1_Load(object sender, EventArgs e)
{
   Image img = new Bitmap("Naplemente.jpg");
   pictureBox1.Image = img;
}
```

Az *Image* egy absztrakt osztály, a *Bitmap* pedig egy leszármazottja. Utóbbi számos konstruktorral (12 –vel) rendelkezik, a fent bemutatott a legegyszerűbb.
A P*ictureBox Image* tulajdonsága pedig egy *Image* leszármazottat vár.

36.16 RichTextBox

A *RichTextBox* szöveg bevitelére és manipulálására alkalmas, de jóval rugalmasabb és több szolgáltatást nyújt, mint a *TextBox*. Emellett kapacitásában is túlmutat rajta. Szöveget rendelhetünk hozzá a *Text* tulajdonságán keresztűl, egyszerű szöveges fileból vagy RTF (Rich Text File) formátumú szövegből:




```
private void Form1_Load(object sender, EventArgs e)
{
   richTextBox1.Text = "Hello";
}
```

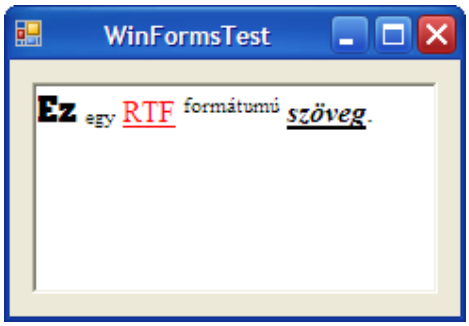
Egy másik módja a szöveggel való feltöltésnek az *AppendText* metódus:

```
private void Form1_Load(object sender, EventArgs e)
{
   richTextBox1.AppendText("Ez a hozzáfűzött szöveg");
}
```

Ekkor a paraméterként átadott karaktersorozat a vezérlő tartalmának végéhez fűződik hozzá (gyakorlatilag a *Text += „szoveg”* rövidített formája).

RTF vagy szöveges állományt olvashatunk be a *LoadFile* metódusával:

```
private void Form1_Load(object sender, EventArgs e)
{
   richTextBox1.LoadFile("test.rtf");
}
```



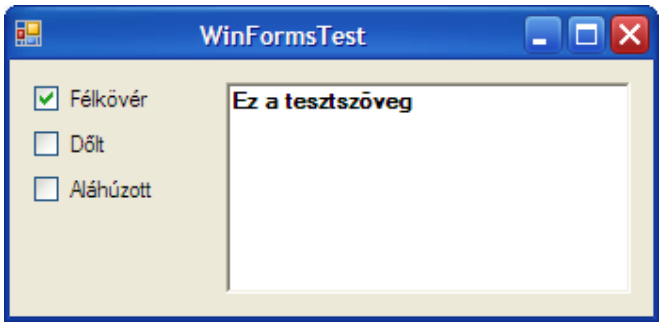
A *LoadFile* párja a *SaveFile*, amellyel RTF –be menthetjük a vezérlő aktuális tartalmát (a példában egy gombhoz rendeltük a mentést):

```
private void button1_Click(object sender, EventArgs e)
{
   richTextBox1.SaveFile("savedfile.rtf");
}
```

A *RichTextBox* rengeteg metódust állít rendelkezésünkre a szöveg formázásához. Ehhez először ki kell jelölni a szöveget, ezt vagy kézzel tesszük meg, vagy a *Find* metódussal. Húzzunk a formra három *CheckBox* –ot, ezekkel fogjuk beállítani, hogy félkövér, dőlt vagy aláhúzott legyen a szöveg. A három vezérlő használja ugyanazt az eseményt:

```
private void checkBox3_CheckedChanged(object sender, EventArgs e)
{
   FontStyle style = FontStyle.Regular;
   switch(((CheckBox)sender).Name)
   {
      case "checkBox1":
         style = FontStyle.Bold;
         break;
      case "checkBox2":
         style = FontStyle.Italic;
         break;
      case "checkBox3":
         style = FontStyle.Underline;
```

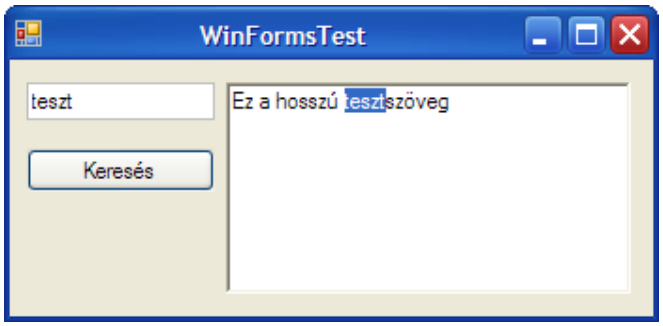
```
            break;
        default:
            break;
    };
    richTextBox1.SelectionFont = new Font(richTextBox1.Font, style);
}
```



Hasonlóan kell kezelni a *SelectionColor*, *SelectedIndent*, stb. tulajdonságokat is.
A *Find* metódussal megkereshetünk és kijelölhetünk egy adott karaktersorozatot. Húzzunk egy gombot és egy *TextBox* –ot a formra, a gomb *Click* eseményéhez pedig írjuk a következőt:

```
private void button1_Click(object sender, EventArgs e)
{
    int idx = richTextBox1.Find(textBox1.Text);
    richTextBox1.Focus();
}
```

Vissza kell adnunk a fókuszt a vezérlőnek, különben nem látszana az eredmény:

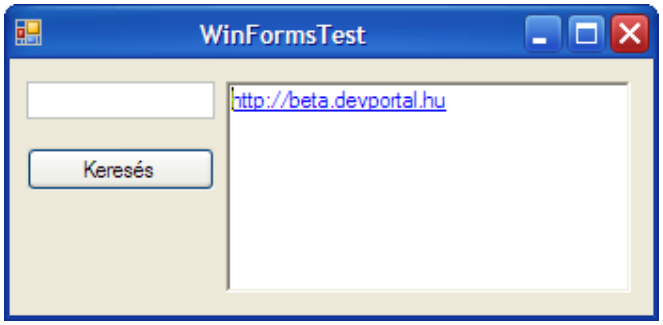


Kijelölni a *Select* metódussal is tudunk, mindössze meg kell adnunk a kezdőindexet és a hosszt:

```
richTextBox1.Select(0, 10);
```

Ekkor a szöveg elejétől kezdve (nulladik index) tíz karakter hosszan jelöljük ki a vezérlő tartalmát.

Ha a vezérlő tartalmaz egy internetes hivatkozást, akkor azt automatikusan felismeri és megjelöli:



Ahhoz azonban, hogy a link működjön is kezelnünk kell a *RichTextBox LinkClicked* eseményét:

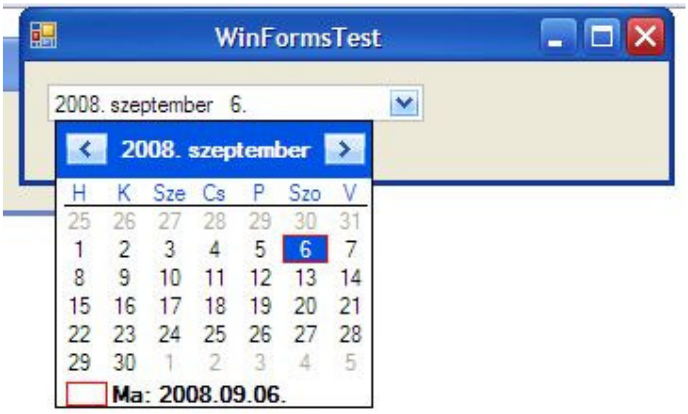
```
private void richTextBox1_LinkClicked(object sender, LinkClickedEventArgs e)
{
   System.Diagnostics.Process.Start(e.LinkText);
}
```

A *LinkClickedEventArgs* a hivatkozásról tartalmaz információkat, *LinkText* tulajdonsága a címet adja vissza.

Hasonlóan a *TextBox* –hoz a *RichTextBox* alapértelmezett eseménye is a *TextChanged*.

36.17 DateTimePicker

Ez a vezérlő lehetővé teszi a felhasználó számára, hogy kiválasszon egy adott dátumot:



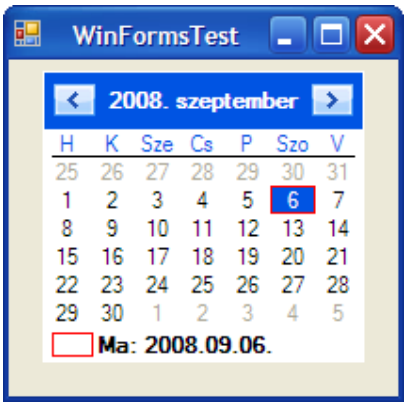
Számos tulajdonság ál lrendelkezésünkre, amelyekkel megváltoztathatjuk a vezérlő megjelenítését, pl. a *CalendarForeColor*, *CalendarFont*, stb…
A *MinDate* és *MaxDate* tulajdonságaival beállíthatjuk, hogy milyen intervallumban választhat a felhasználó.
Alapértelmezett eseménye a *ValueChanged*, amely a kiválasztott dátum megváltozásakor aktivizálódik. Írjunk egy programot, amely egy *MessageBox* –ban megjeleníti a kiválasztott dátumot:

```
private void dateTimePicker1_ValueChanged(object sender, EventArgs e)
```

```
{
   MessageBox.Show(dateTimePicker1.Value.ToShortDateString());
}
```

A *Value* tulajdonság egy *DateTime* típusú értéket ad vissza. A fenti példában a *ToShortDateString* metódust használtuk, ez csak a napra lebontott dátumot adja vissza (vagyis év-hónap-nap), amennyiben másodpercre pontos értéket akarunk használjuk a sima *ToString* metódust.

A *DateTimePicker* tulajdonképpen két vezérlőből áll, egy *TextBox* leszármazottból illetve a dátum kiválasztásához egy *MonthCalendar* –ból, ez utóbbi önállóan is felhasználható:



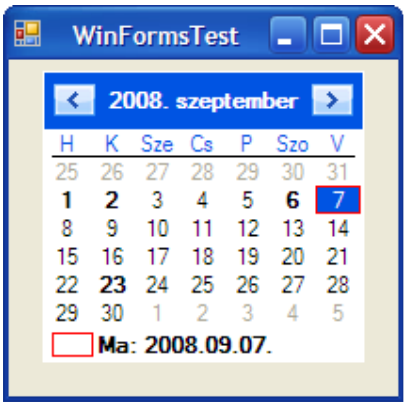
A *FirstDayOfWeek* tulajodnságával a hetek kezdőnapját adhatjuk meg, ez egyes országokban eltérő lehet, alapértelmezetten az operációs rendszer nyelve alapján kap kezdőértéket.

Kijelölhetünk napokat, amelyek a többitől kiemelkednek, ezt a BodledDates, AnnuallyBoldedDates illetve MonthlyBoldedDates tulajdonságokkal érhetjük el, mindegyikük egy DateTime típusú tömböt vár:

```
private void Form1_Load(object sender, EventArgs e)
{
   monthCalendar1.BoldedDates = new DateTime[]
   {
      new DateTime(2008, 9, 2)
   };

   monthCalendar1.AnnuallyBoldedDates = new DateTime[]
   {
      new DateTime(2008, 9, 6),
      new DateTime(2008, 9, 1),
```

```
        new DateTime(2008, 9, 23)
    };
}
```

Az eredmény pedig ez lesz:



Két fontos eseménye van, az első a *DateChanged*, amely akkor aktivizálódik, amikor kiválasztunk az egérrel egy dátumot:

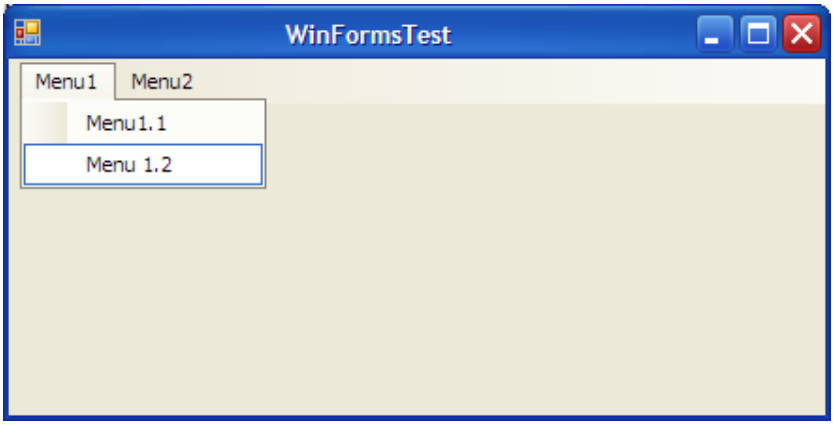
```
private void monthCalendar1_DateChanged(object sender, DateRangeEventArgs e)
{
    MessageBox.Show(e.End.ToShortDateString());
}
```

A *DataRangeEventArgs End* illetve *Start* tulajdonságai a felhasználó által utolsóként és először kiválasztott dátumokat adják vissza. Természetesen ebben az esetben csak egy időpontot tudunk visszaadni, a második paramétert igazából a másik *DateSelected* eseményben tudjuk felhasználni (és ez a vezérlő alapértelmezett tulajdonsága is). Az egérrel kijelölhetünk több dátumot és ekkor már fel tudjuk használni az *End* és *Start* tulajdonságokat:

```
private void monthCalendar1_DateSelected(object sender, DateRangeEventArgs e)
{
    MessageBox.Show(e.Start.ToShortDateString() + " " + e.End.ToShortDateString());
}
```

36.18 MenuStrip

Ezzel a vezérlővel a legtöbb programban megszokott menüsort tudunk létrehozni:



Kétféleképpen tudjuk feltölteni a menüpontokat, a legegyszerűbb a Visual Studio tervező nézetét használni, ekkor csak szimplán be kell írni a szöveget. A másik lehetőség, hogy kódból hozzuk létre, ez a következőképpen néz ki:

```
private void Form1_Load(object sender, EventArgs e)
{
   for (int i = 0; i < 3; ++i)
   {
      ToolStripMenuItem item = new ToolStripMenuItem("Menu" + i.ToString());
      item.DropDownItems.Add("Menu" + i.ToString() + ".1");
      menuStrip1.Items.Add(item);
   }
}
```

Látható, hogy a *MenuStrip* minden egyes menüpontja egy *ToolStripItem* típusú objektum. A *MenuStrip Items* és a *ToolStripMenuItem DropDownItems* tulajdonsága is egy *ToolStripItemCollection* típusú listát ad vissza, amelynek tagjai mind *ToolStripMenuItem* típusuak. A fenti kód kétféle módját mutatja be a menühöz való hozzáadásnak: vagy a hagyományos úton létrehozzuk az objektumot és hozzáadjuk, vagy helyben az *Add* metódus paramétereként megadott string értékkel. Utóbbi esetben automatikusan meghívódik a *ToolStripMenuItem* konstruktora.
A menü megvan, most kellene valami funkcionalitást is hozzáadnunk, ezt két módon tudjuk megoldani. Kezelhetjük a *MenuStrip* alapértelmezett eseményét, az *ItemClick* –edet, amely értelemszerűen akkor sül el, amikor rákattintunk valamelyik menüpontra. Azt is tudnunk kell ekkor, hogy melyik menüpont volt a „bűnös”. Nézzük meg az eseményt:

```
private void menuStrip1_ItemClicked(object sender, ToolStripItemClickedEventArgs e)
{

}
```

A *ToolStripEventArgs ClickedItem* tulajdonsága visszaadja azt a *ToolStripMenuItem* objektumot, amelyre kattintottuk. Viszont van egy probléma, méghozzá az, hogy ez a *MenuStrip* eseménye, nem pedig a gyermek objektumoké. A legfelsőbb szintű elemekre kiváltódik, de az almenükre nem. Ilyenkor a *ToolStripMenuItem ItemClicked* eseményét (ez bizony ugyanaz, vajon miért?) fogjuk használni. Mielőtt továbbmegyünk gondolkodjunk egy kicsit: tegyük fel, hogy a *MenuStrip* eseményét

használjuk, és el kell dönteni, hogy az egyes menüpontok kiválasztásakor mit tegyünk. Írhatunk mondjuk valami ilyesmit:

```
private void menuStrip1_ItemClicked(object sender, ToolStripItemClickedEventArgs e)
{
   switch (e.ClickedItem.Name)
   {
      case "elso":
         //itt csinálunk valamit
         break;
      case "masodik":
         //itt is csinálunk valamit
         break;
         /*...*/
      default:
         break;
   };
}
```

Ez a megoldás nem valami szép. Átláthatatlan és nehezen módosítható. Olyan esetekben azonban jó, amikor a menü csak egyszintű.
Bonyolultabb helyzetben viszont sokkal egyszerűbb az egyes menüpontok *ItemClicked* eseményét használni, ehhez az adott menüponton kattintsunk duplán:

```
private void menu2ToolStripMenuItem_Click(object sender, EventArgs e)
{

}
```

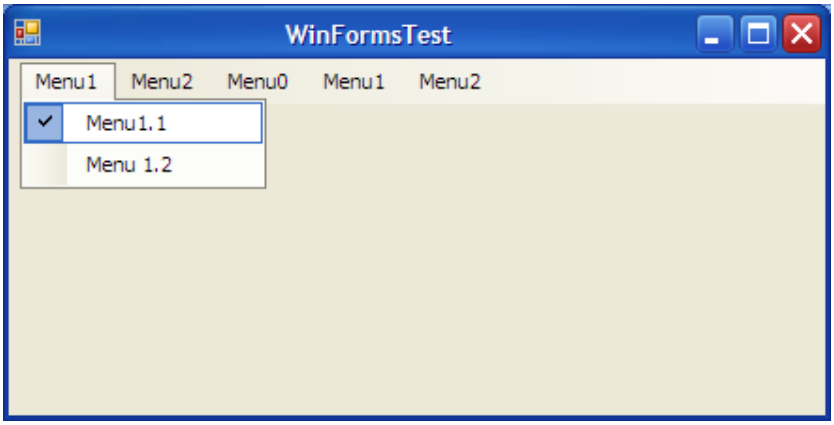
Itt már nem jár a speciális eseményargumentum, mivel ilyen esetekben nem érdekel minket, hogy ki küldte a parancsot, csak az utasítás a lényeg.
Amennyiben a menüpontot kódból hoztuk létre, akkor rögtön meg is adhatjuk neki az eseménykezelőjét:

```
ToolStripMenuItem item = new ToolStripMenuItem("Menu", null, new
EventHandler(Click_EventHandler));
```

A konstruktor második paramétere a menüponthoz rendelt kép lenne, de ezt most nem adtuk meg.
Módosítsuk az eseménykezelőt, hogy egy *MessageBox* –ban jelenítse meg a menüpont nevét:

```
private void menu11ToolStripMenuItem_Click(object sender, EventArgs e)
{
   MessageBox.Show(((ToolStripMenuItem)sender).Text);
}
```

A menüpontok *CheckOnlick* tulajdonságának igazra állításával beállíthatjuk, hogy kattintásra változtassa az állapotát (ez olyankor jön jól, amikor egy kétállású tulajdonságot akarunk implementálni):

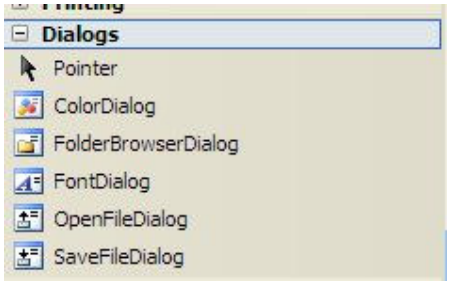


A *Checked* tulajdonság visszadja, hogy a menüpont be van –e jelölve, vagy sem:

```
private void menu11ToolStripMenuItem_Click(object sender, EventArgs e)
{
   MessageBox.Show(((ToolStripMenuItem)sender).Checked.ToString());
}
```

36.19 Általános párbeszédablakok

A .NET –ben lehetőségünk van előre megírt dialógusablakok használatára, mint pl. amivel megnyithatunk egy szöveges file –t. Ezeket az előre legyártott elemeket a Visual Studio tervezőnézetében a ToolBox ablak Dialogs füle alatt találjuk, összesen hat darab van (a *PrintDialog* a Printing fül alatt lakik).

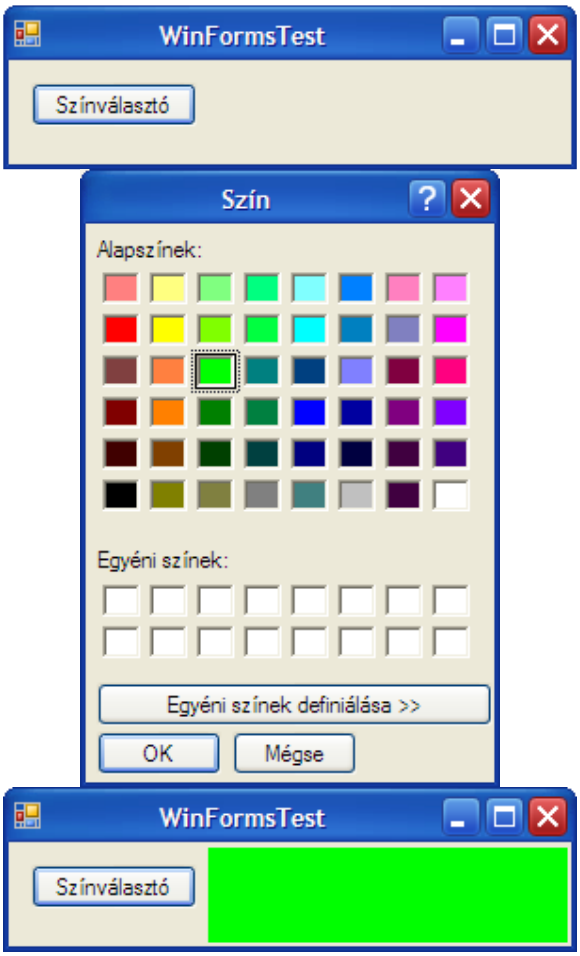


Elsőként a legegyszerűbbet a *ColorDialog* –ot fogjuk megvizsgálni. Egy ilyen dialógusablakot kétféleképpen használhatunk fel. Vagy a tervezőnézetben ráhúzzuk a formra (ekkor alul megjelenik egy külön sávban), vagy a kódból hozzuk létre, ha szükség van rá. Az előbbi esetben egy (alapértelmezetten) *private* elérésű *ColorDialog* típusú adattag jön létre, azaz a form teljes élettartama alatt ugyanazt az ablakot használjuk. Ez olyankor jön jól, ha gyakran van szükségünk az adott ablakra. Az ellenkező esetben, amikor csak ritkán használjuk a dialógust elegánsabb, ha lokális változóként hozzuk létre.
Jöjjön egy gyakorlati példa: húzzunk a formra egy gombot és egy *PictureBox* –ot, illetve egy *ColorDialog* –ot. A gomb *Click* eseményét fogjuk használni:

```
private void button1_Click(object sender, EventArgs e)
{
   if (colorDialog1.ShowDialog() == DialogResult.OK)
   {
```

```
        pictureBox1.BackColor = colorDialog1.Color;
    }
}
```

A *ShowDialog* metódus egy *DialogResult* típusú felsorolt értéket ad vissza, attól függően, hogy a felhasználó mit választott (az IntelliSense mutatja a többi lehetőséget is, kísérletezzünk). A fenti példában, ha az eredmény *OK*, akkor tudjuk, hogy a felhasználó választott valamilyen színt, ezt pedig lekérdezhetjük a *Color* tulajdonsággal.



Ha az ablakot tisztán kódból akarjuk létrehozni, akkor a *Click* esemény így néz ki:

```
private void button1_Click(object sender, EventArgs e)
{
    ColorDialog colorDialog1 = new ColorDialog();

    if (colorDialog1.ShowDialog() == DialogResult.OK)
    {
        pictureBox1.BackColor = colorDialog1.Color;
    }
}
```

Ekkor viszont minden egyes alkalommal új objektum jön létre, mivel a lokális változók hatóköre a metódus végén lejár.

Következő a sorban az *OpenFileDialog*, ez – mint azt a neve is mutatja – arra való, hogy file –okat nyissunk meg. Húzzunk egy formra egy *RichTextBox* –ot illetve egy gombot, ez utóbbit arra használjuk, hogy megnyissunk egy szöveges állományt, amit majd a *RichTextBox* –ban megjelenítünk.
Az *OpenFileDialog* –nak rengeteg érdekes tulajdonsága van, most azonban csak a *Filter* –t vizsgáljuk. Ezzel megadhatjuk, hogy milyen típusú file –okat akarunk megnyitni, pl.: *Text files|*.txt*.
A *Filter* tulajdonság kódból is megadható:

```
openFileDialog1.Filter = "Text files|*.txt|";
```

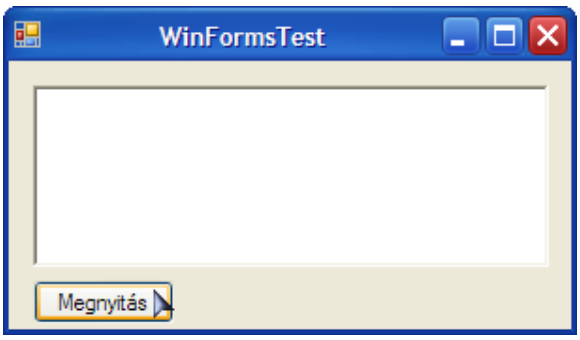
Akár többféle szűrőt is megadhatunk:

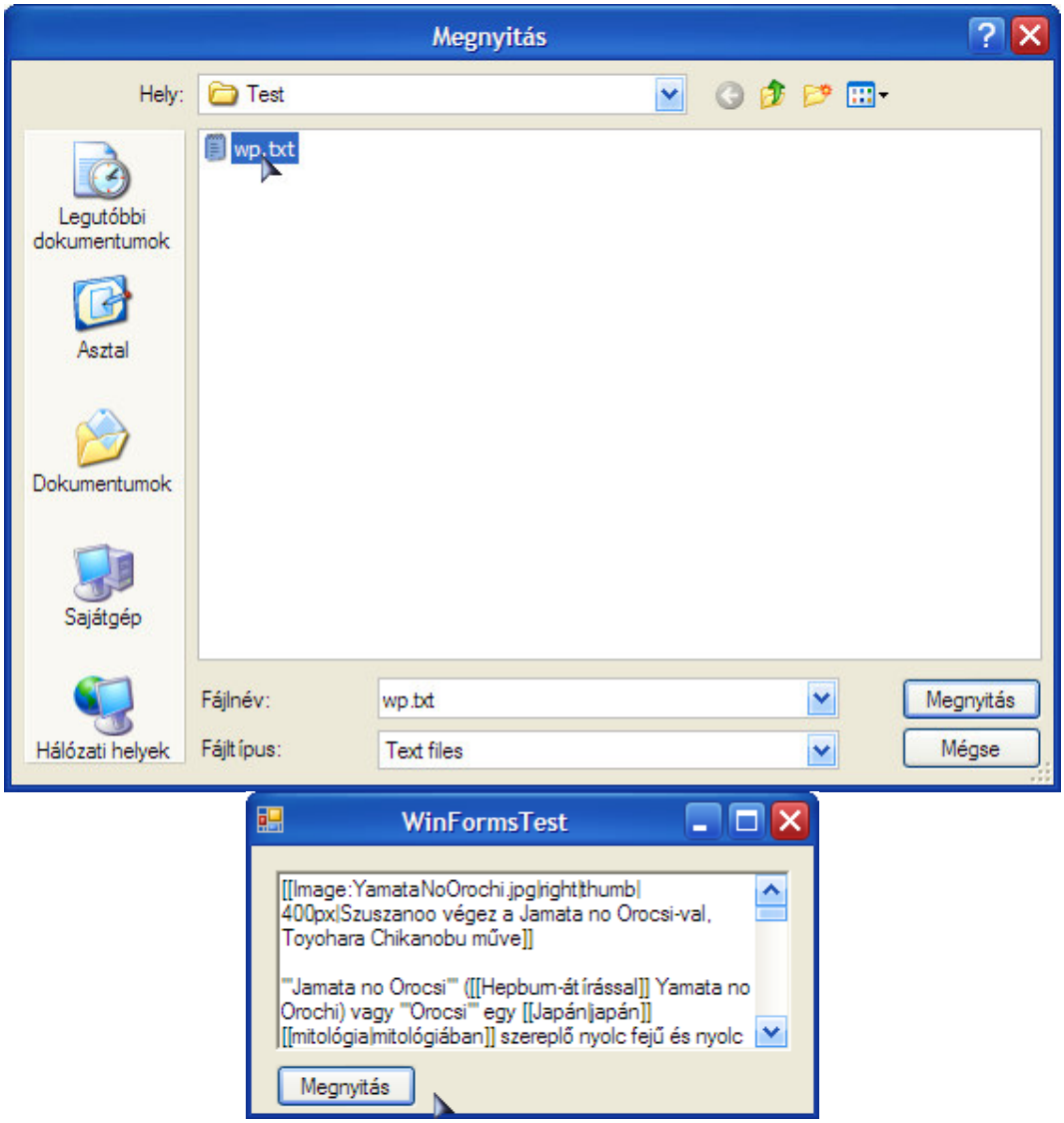
```
openFileDialog1.Filter = "Text files|*.txt|All Files|*.*";
```

A programunkban gomb *Click* eseménye:

```
private void button1_Click(object sender, EventArgs e)
{
   if (openFileDialog1.ShowDialog() == DialogResult.OK)
   {
      richTextBox1.LoadFile(openFileDialog1.FileName,
RichTextBoxStreamType.PlainText);
   }
}
```

Ebben az esetben a *LoadFile* metódusban meg kell adnunk, hogy milyen típusú file –t akarunk a *RichTextBox* –ba beolvasni, mivel az alapértelmezetten RTF formátumúakat olvas. A dialógus *FileName* tulajdonsága a file elérési útját adja vissza.





Az *OpenFile* metódus megnyit egy stream –et csakis olvasásra a kiválasztott file –ra:

```
private void button1_Click(object sender, EventArgs e)
{
   openFileDialog1.Filter = "Text files|*.txt|All Files|*.*";

   if (openFileDialog1.ShowDialog() == DialogResult.OK)
   {
      richTextBox1.LoadFile(openFileDialog1.OpenFile(),
RichTextBoxStreamType.PlainText);
   }
}
```

Itt kihasználtuk a *RichTextBox* azon tulajdonságát, hogy stream –ből is képes olvasni.

Természetesen a *Filter* tulajdonságot elegánsabb és ésszerűbb a konstruktorban vagy a form *Load* eseményében beállítani, a fenti megoldás az érthetőség miatt született.

Az *OpenFileDialog* párja a *SaveFileDialog*, amely file –ok elmentésére szolgál. Az előző példát fogjuk használni, de most a gomb lenyomásakor elmentjük a *RichTextBox* tartalmát:

```
private void button1_Click(object sender, EventArgs e)
{
   if (saveFileDialog1.ShowDialog() == DialogResult.OK)
   {
      richTextBox1.SaveFile(saveFileDialog1.OpenFile(),
RichTextBoxStreamType.PlainText);
   }
}
```

Az *OpenFile* metódus egy írható/olvasható stream –et nyit meg, amelybe a *RichTextBox SaveFile* metódusával írjuk ki az adatokat.

Következő a sorban a *FontDialog*, amely betűtípus beállítására használandó. A példa ugyanaz lesz itt is, de gombnyomásra előjön az ablak és megváltoztatjuk a *RichTextBox* betűtípusát:

```
private void button1_Click(object sender, EventArgs e)
{
   if (fontDialog1.ShowDialog() == DialogResult.OK)
   {
      richTextBox1.Font = fontDialog1.Font;
   }
}
```

A *PrintDialog* –gal kinyomtathatunk egy dokumentumot, még mindig a példánknál maradva a *RichTextBox* tartalmát. Ehhez szükségünk lesz egy *PrintDocument* típusú objektumra, amely a nyomtatandó adatokról tartalmaz megfelelő formátumú információkat. Ezt az objektumot akár a ToolBox –ból is a formra húzhatjuk, ez lesz a legegyszerűbb.

Mielőtt továbbmegyünk érdemes a teszteléshez beszerezni valamilyen PDF nyomtatót, pl. az ingyenes PDF995 –öt.

A tényleges nyomtatást a *PrintDocument* fogja végezni, a *PrintDialog* a beállítások miatt kell. Most is gombnyomásra fogjuk elindítani a folyamatot:

```
private void button1_Click(object sender, EventArgs e)
{
   if (printDialog1.ShowDialog() == DialogResult.OK)
   {
      printDocument1.Print();
   }
}
```

Amikor meghívjuk a *Print* metódust kiváltódik a *PrintDocument PrintPage* eseménye, ahol a tényleges nyomtatást fogjuk megcsinálni:

```
private void printDocument1_PrintPage(object sender,
System.Drawing.Printing.PrintPageEventArgs e)
{
   Font font = new Font("Arial", 10);
   float maxLine = e.MarginBounds.Height / font.GetHeight(e.Graphics);

   int i = 0;
   while (i < maxLine && i < richTextBox1.Lines.Length)
   {
      float y = i * font.GetHeight(e.Graphics) + e.MarginBounds.Top;
      e.Graphics.DrawString(richTextBox1.Lines[i], font, Brushes.Black,
e.MarginBounds.Left, y, new StringFormat());
      ++i;
   }

   e.HasMorePages = i < richTextBox1.Lines.Length;
}
```

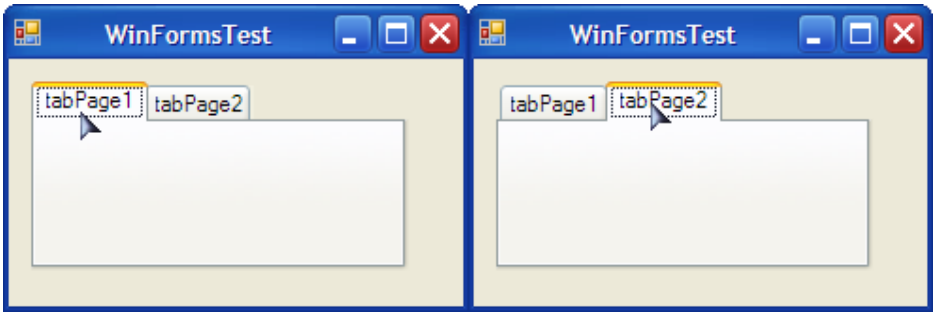
Az esemény paramétere a nyomtatásról tartalmaz információkat, pl margó, de ezt fogjuk használni a tényleges nyomtatáshoz is.
Elsőként létrehoztunk egy *Font* típusú objektumot, ez a betűtípust fogja jelenteni. Ezután kiszámoljuk, hogy a beállított betűtípus és méret mellett hány sor fér el egy oldalra, vagyis elosztjuk a lap magasságát a betűk magasságával. Most egy while ciklust indítunk, amely vagy addig megy amíg teleírja alapot, vagy az adatforrás végéig. Itt kihasználtuk, hogy a *RichTextBox Lines* tulajdonsága az egyes sorokat adja vissza egy stringtömb formájában. A ciklusban kiszámoljuk, hogy hol kezdődik az adott sor (vagyis az *y* koordinátát): megszorozzuk a betűk magasságát az aktuális sor számával, ezt az értéket pedig hozzáadjuk a lap tetejének értékéhez. A következő lépésben az esemény paraméterét használjuk, annak is a *DrawString* metódusát, ennek első paramétere a nyomtatandó sor, a második a betűtípus, a harmadik a betűk színe, a negyedik és ötödik a kezdés *x* illetve *y* koordinátája, végül pedig a formátumstring, amelyet üresen hagytunk.
Az esemény legvégén beállítjuk a *HasMorePages* logikai (*bool* típusú) tulajdonságot, vagyis azt, hogy van –e még nyomtatandó szöveg. Ha van, akkor az esemény automatikusan újrakezdődik (pontosabban folytatódik).

Az előre megírt párbeszédablakok közül az utolsó a *FolderBrowserDialog*, amelynek segítségével kiválaszthatunk a könyvtárstruktúrából egy mappát. Készítsün kegy programot, amely gombnyomásra megjelenít egy *FolderBrowserDialog* –ot, és miután a felhasználó kiválasztott egy mappát egy *TextBox* –ban jelenítsük meg a mappa elérési útját. A gomb *Click* eseménye következő lesz:

```
private void button1_Click(object sender, EventArgs e)
{
   if (folderBrowserDialog1.ShowDialog() == DialogResult.OK)
   {
      textBox1.Text = folderBrowserDialog1.SelectedPath;
   }
}
```

36.20 TabControl

Ezzel a vezérlővel több párhuzamosan létező részre oszthatjuk a formot, amely részekből egyszerre csak egy látszik:



Az egyes lapok *TabPage* típusúak, amelyek lekérhetőek a *TabControl TabPages* tulajdonságával, amely egy *TabPageCollection* típusú listával tér vissza (ez felhasználható egy foreach ciklusban, a lista egyes elemei nyílván *TabPage* típusúak).
A *TabControl* –nak nincs alapértelmezett eseménye, ehelyett az adott lap *Click* eseményét kezeljük, ha duplán kattintunk a vezérlőre (ezt minden lapnál külön kell megtennünk).

36.21 ContextMenuStrip

Biztosan mindenki ismeri a jobb egérgombra előugró helyi menüt, ez a vezérlő pontosan erről szól. Bármelyik vezérlőhöz hozzáköthetjük, ha beállítjuk a *ContextMenu* tulajdonságát. A *ContextMenuStrip* tulajdonképpen a már ismert *MenuStrip* „mobilizált” változata, minden amit ott tanultunk itt is hasznosítható.

36.22 Gyakorló feladatok

1. Készítsünk programot, amely a „Misszionárius - kannibál” problémát modellezi. Egy folyó egyik partján 3 kannibál és 3 misszionárius áll. A cél az, hogy egy csónakban mindenkit átjuttasunk a túlpartra. A csónak kétszemélyes és semelyik parton nem lehet több kannibál, mint misszionárius, egyébként a kannibálok megeszik őket. A két partot jelezheti mondjuk két *ListBox*.

2. Készítsünk egyszerű szövegszerkesztőt a *RichTextBox* és a menüvezérlők segítségével. Lehessen menteni, betölteni és formázni is.

37.Windows Forms – Komponensek

A komponensek olyan osztályok, amelyek közvetlenül nem látszanak a formon, de speciálisan arra tervezték őket, hogy Windows Forms környezetben működjenek a legjobban. Ebben a fejezetben néhány fontosabb komponenssel ismerkedünk meg.

37.1 Timer

A *Timer* egy olyan komponens, amely lehetővé teszi, hogy bizonyos időközönként végrehajtsunk egy utasítást. Készítsünk egy programot, amely automatikusan mozgat egy vezérlőt (mondjuk egy gombot) egy formon. A ToolBox –ból húzzunk egy *Timer* –t a formra (a Component fül alatt található), és egy tetszőleges vezérlőt is (a példában gomb lesz). A *Timer* tulajdonságai közt találjuk az *Interval* –t, amely az eseményünk gyakoriságt fogja jelölni ezredmásodpercben (tehát ha az *Interval* 1000, akkor másodpercenként fog aktiválódni az esemény).
Az aktivált esemény pedig a *Timer* alapértelmezett eseménye a *Tick* lesz. A programunkban a *Tick* kódja ez lesz:

```
private void timer1_Tick(object sender, EventArgs e)
{
   if (button1.Location.X + button1.Width + 10 < this.Width)
   {
      button1.Location = new Point(button1.Location.X + 10, button1.Location.Y);
   }
   else
   {
      timer1.Stop();
   }
}
```

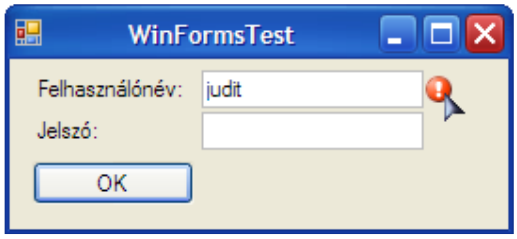
A gombot addig mozgatjuk balra, amíg eléri a form szélét, ha pedig odaért akkor megállítjuk a *Timer* –t. Valahol azonban el is kell indítanunk, erre pedig a legjobb hely a gomb *Click* eseménye:

```
private void button1_Click(object sender, EventArgs e)
{
   timer1.Start();
}
```

A *Timer* nincs felkészítve többszálú végrehajtásra, ha erre vágyunk, akkor használjuk a *System.Timers* névtér *Timer* osztályát.

37.2 ErrorProvider

Ezzel a komponenssel egy hibaüzenetet köthetünk egyes vezérlőkhöz, ez - alapértelmezetten – egy ikonnal jelenik meg:



A vezérlőhöz kötést az *ErrorProvider SetError* metódusával tudjuk megtenni, első paramétere a vezérlő amihez kötünk, a második pedig a hibaüzenet.
A fenti program egy - nagyon – egyszerű „beléptetőrendszer”. Húzzuk rá a formra a megfelelő vezérlőket és egy *ErrorProvider* –t is (ez alul fog megjelenni). A felhasználóneveket és jelszavakat egy *Dictionary* gyűjteményben fogjuk eltárolni:

```
private Dictionary<string, string> users = new Dictionary<string, string>();

private void Form1_Load(object sender, EventArgs e)
{
   users.Add("Istvan", "istvan");
   users.Add("Judit", "judit");
   users.Add("Ahmed", "ahmed");
}
```

Gombnyomásra fogjuk ellenőrizni a megadott adatok helyességét:

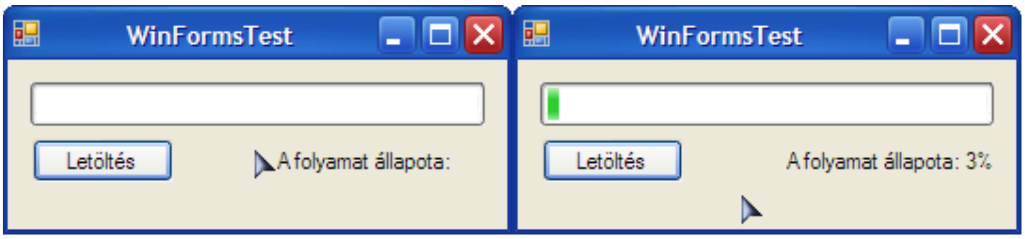
```
private void button1_Click(object sender, EventArgs e)
{
   if (users.ContainsKey(textBox1.Text))
   {
      if (users[textBox1.Text] == textBox2.Text)
      {
         MessageBox.Show("Helyes felhasználónév és jelszó!");
      }
      else
      {
         errorProvider1.SetError(this.textBox2, "Rossz jelszó!");
      }
   }
   else
   {
      errorProvider1.SetError(this.textBox1, "Nem létező felhasználó!");
   }
}
```

37.3 BackGroundWorker

Ezzel a komponenssel egy bizonyos feladatot egy háttérben futó szálban végezhetünk el. Ez olyankor hasznos, ha a feladat időigényes (pl. egy file letöltése) és nem engedhetjük meg, hogy addig a form ne tudja fogadni a felhasználó utasításait. Az utasítás végrehatásához létre kell hoznunk a komponens *DoWork* eseményének a kezelőjét. Ez az esemény a *BackGroundWorker RunWorkerAsync* metódusának meghívásakor váltódik ki (ennek túlterhelt változata paramétert is kaphat). A folyamat állapotát lekérdezhetjük a *ProgressChanged* esemény

kezelésével, illetve a feladat teljesítését a *RunWorkerCompleted* esemény kezeli le. Ahhoz, hogy használhassuk a *ProgressChanged* eseményt a komponens *WorkerReportsProgress* tulajdonságát igaz értékre kell állítanunk.
A *BackGroundWorker* nem támogatja a szálak közötti kommunikációt, így a felhasználói felület nem módosítható a *DoWork* eseményből, ellenkező esetben *InvalidOperationException* típusú kivételt kapunk. Ugyanígy nem lehetséges Application Domain-ek közötti kommunikációt sem végrehajtani.

A példaprogramban egy file letöltését fogjuk szimulálni. Húzzunk a formra egy *ProgressBar* –t, egy gombot és egy *Label* –t. A kész program így néz ki:



A „letöltést” gombnyomásra indítjuk:

```
private void button1_Click(object sender, EventArgs e)
{
   backgroundWorker1.RunWorkerAsync();
}
```

A *DoWork* eseménykezelő ez lesz:

```
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
   for (int i = 0; i < 100; ++i)
   {
      System.Threading.Thread.Sleep(200);
      backgroundWorker1.ReportProgress(i + 1);
   }
}
```

A *ReportProgress* metódus fogja kiváltani a *ProgressChanged* eseményt. Ez a metódus paramétereként a folyamat állapotát kapja százalékos értékben (a ciklusváltozó értékéhez azért adtunk egyet, hogy a valós értéket mutassa, hiszen nulláról indul). Ennek a metódusnak van egy túlterhelt változata is, amely második paramétereként egy *object* típusú változót vár, amely tetszőleges adatot tartalmazhat. Ezt a *ProgressChangedEventArgs UserState* tulajdonságával kérdezhetjük le.
Végül a *ProgressChanged* és a *RunWorkerCompleted* esemény:

```
private void backgroundWorker1_ProgressChanged(object sender,
ProgressChangedEventArgs e)
{
   ++progressBar1.Value;
   label1.Text = "A folyamat állapota: " + e.ProgressPercentage.ToString() + "%";
```

```
}
```

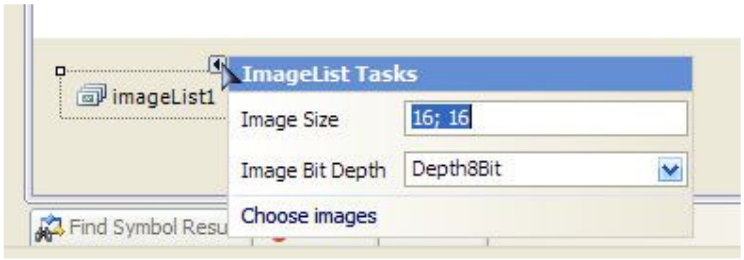
```
private void backgroundWorker1_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
{
   MessageBox.Show("A letöltés kész!");
}
```

37.4 Process

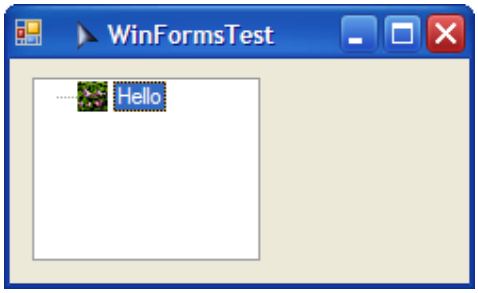
Helyi és távoli folyamatok vezérlésére ad lehetőséget ez a komponens. A többszálúságról szóló fejezetben már találkoztunk vele.

37.5 ImageList

Ennek a komponesnek a segítségével több képet tudunk kezelni, hasonlóan mint egy gyűjteménnyel. A komponensnek előre megadhatjuk az elemeit, ekkor azok „belefordulnak" a futtatható állományba. A komponensben beállíthatjuk, a képek méreteit és színmélységét is.
Egy *ImageList* –et pl. felhasználhatunk egy *TreeView* vezérlőhöz, megadva az egyes *Node* –okhoz rendelt ikont. Húzzunk a formra egy *ImageList* komponenenst és egy *TreeView* vezérlőt. Az *ImageList* „tetején" megjelenik egy kis nyilacska, erre kattintva beállíthatjuk a képek formátumát és itt adhatjuk hozzá őket:



A *TreeView* *ImageList* tulajdonságánál állíthatjuk be a használandó komponenst. Adjunk néhány elemet a vezérlőhöz, ezután az eredmény:



Természetesen nem csak egy képet rendelhetünk az összes *Node* –hoz, hanem az *ImageIndex* tulajdonságon keresztül egyesével is beállíthatjuk azt:

```
treeView1.Nodes[0].ImageIndex = 0;
```

37.6 Gyakorló feladatok

1. Készítsünk „reflexjátékot”, a *Timer* komponens segítségével! A formon egy vezérlő (mondjuk egy *PictureBox*, de teljesen mindegy) adott időközönként változtatja a helyét, a játékos feladata pedig, hogy rákattintson. A felhasználó állíthassa be, hogy milyen gyorsan mozogjon a vezérlő, de ne engedjük, hogy ez túl lassú legyen.

2. Készítsünk „teknősversenyt”! Az állatokat szimbolizáló vezérlők egyenes vonalban mozogjanak és a *Timer* minden *Tick* –jénél döntsük el, hogy mi történik: ha a teknős épp fáradt (ezt is mérjük), akkor az adott körből kimarad, ha nem, akkor sorsoljunk neki egy véletlenszámot. Ez a szám fogja jelezni, hogy hány mezőt léphet előre. Nehezítésképpen a játék elején lehessen fogadni a győztesre, illetve beilleszthetőek egyéb események is a játékmenetbe (pl. csapda).

3. Készítsünk „irodaszimulátort”! A játkos a főnök, aki elküldi az alkalmazottait különböző feladatok elvégzésére. Az alkalmazottak BackgroundWorker –ek legyenek, és jelezzük, ha elvégezték a dolgukat. Nehezítésképpen akár egy egyszerű gazdasági játékot is csinálhatunk.

38. Windows Forms - Új vezérlők létrehozása

Ha saját vezérlő létrehozásán törjük a fejünket két út áll előttünk. Vagy egy már létezőt bővítünk ki származtatással, vagy teljesen újat hozunk létre, egy ún. UserControl –t. Egy harmadik lehetőség, hogy közvetlenül a Control osztályból származtatunk, amely minden más vezérlő őse. Ezt a metódust ez a jegyzet nem részletezi.
Felmerülhet a kérdés, hogy melyik módszert mikor használjuk? Ha van olyan vezérlő, amely rendelkezik a szükséges képességek nagy részével, akkor a legjobb lesz, kibővítjük azt a szükséges metódusokkal, tulajdonságokkal.
Ha viszont olyan vezérlőt szeretnénk, ami több más vezérlő tulajdonságait olvasztja magába, akkor UserControl –t kell létrehoznunk.

38.1 Származtatás

Ebben az esetben rendkívül egyszerű dolgunk lesz, ugyanis egy szimpla származtatást kell végrehajtanunk. A (nagyon egyszerű) példaprogramban a *ListBox* vezérlőt fogjuk kibővíteni oly módon, hogy lekérdezhessük a leghosszabb elemét.
A Solution Explorer –ben kattintsunk jobb egérgombbal a Solution –ra, tehát ne a projectekre, hanem a legfelső „bejegyzésre". Válasszuk az Add menüpontot, azon belül pedig az New Project –et. A Windows fülnél válasszuk ki a Class Library sablont. Amikor létrejött az osztály kattintsunk jobb gombbal az újonnan létrejött projectre és válasszuk az Add Reference menüpontot, mivel hozzá kell adnunk a megfelelő könyvtárakat. A megjelenő ablak .NET feliratú fülén keressük meg a System.Windows.Forms assemblyt és adjuk hozzá.
Most már minden adott a fejlesztéshez. Az osztály most így néz ki (hozzáadtam a szükséges névtereket és elkezdtem írni az osztályt is):

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


namespace ExtendedListBox
{
   public class ExtendedListBox : ListBox
   {
      public ExtendedListBox()
         : base()
      {

      }
   }
}
```

Fontos, hogy a konstruktorban ne felejtsük el meghívni az ősosztály konstruktorát. A beépített vezérlők csakis paraméternélküli konstruktorral rendelkeznek, ezért az új vezérlőt is ennek szellemében fogjuk megírni.

Készítsük el a metódust:

```
public string MaxLengthItem()
{
   string max = "";
   foreach (string s in this.Items)
   {
      if (s.Length > max.Length)
      {
         max = s;
      }
   }
   return max;
}
```

Ez a metódus persze csakis az első leghosszabb elemet adja vissza, ha több ugyanolyan hosszú is van a *ListBox* –ban, akkor azt nem veszi figyelembe.

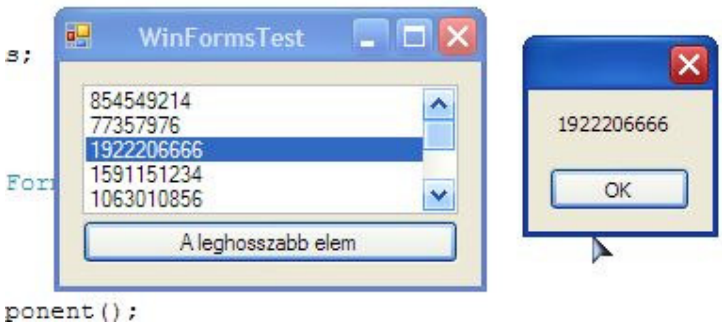
A vezérlő máris használható állapotban van, de egyelőre csakis futási időben adhatjuk hozzá a formhoz. Természetesen lehetőség van arra, hogy ezt tervezési időben tegyük meg, ehhez elsőként fordítsuk le az új vezérlő projectjét, a Build menüponttal, ekkor létrejön a vezérlőt tartalmazó assembly. Ezután navigáljunk el a form tervezőnézetéhez és kattintsunk jobb egérgombbal a projectre. Válasszuk az Add Reference menüpontot, és kattintsunk a megjelenő ablakban a Browser fülre. Menjünk el az új vezérlő könyvtárába, azon belül pedig a Bin mappába. Itt a fordítás „típusától" függően a Debug vagy Release mappában találjuk a megfelelő dll kiterjesztésű file –t. Adjuk hozzá a projecthez. Ezután jó eséllyel a ToolBox –ban megjelenik a vezérlő, ha mégsem, akkor jobb klikk a ToolBox –on Add Tab menüpont, nevezzük el valahogy, majd szintén jobb klikk az új fülre és Choose Items. A .NET Framework Components fül alatt keressük meg a vezérlőnket és pipáljuk ki (beírva a nevét gyorsabban megtaláljuk).
Ezután ugyanúgy használható a vezérlő, mint az összes többi, próbáljuk is ki:

```
private void Form1_Load(object sender, EventArgs e)
{
   Random r = new Random();
   for (int i = 0; i < 10; ++i)
   {
      extendedListBox1.Items.Add(r.Next().ToString());
   }
}
```

Húzzunk egy gombot a formra, ennek a *Click* eseményében keressük majd meg a leghosszabb elemet:

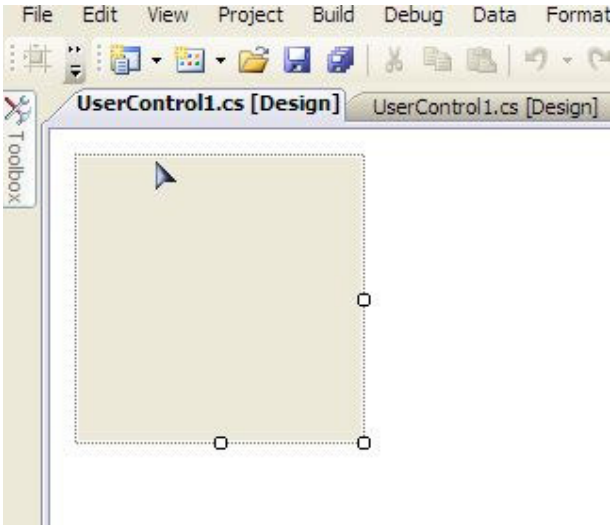
```
private void button1_Click(object sender, EventArgs e)
{
   MessageBox.Show(extendedListBox1.MaxLengthItem());
}
```

Az eredmény:

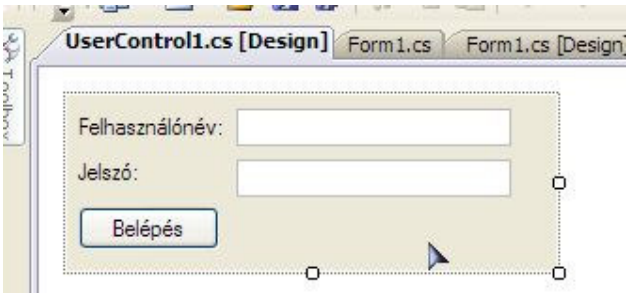


38.2 UserControl -ok

A UserControl példa egy „beléptetőrendszer” lesz. Adjunk egy új projectet a Solution –höz, de ezúttal válasszuk a Windows Forms Control Library sablont. Amikor létrejött a project, akkor egy üres alapot kell látnunk, erre húzhatjuk rá a vezérlőket:



A vezérlő valahogy így nézzen ki:



Ezenkívűl legyen egy *ErrorProvider* is, ami jelzi majd a hibákat. A vezérlő kódját megnyithatjuk, ha jobb egérgombbal kattintva a View Code menüpontot választjuk:

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

```
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsControlLibrary2
{
   public partial class UserControl1 : UserControl
   {
      public UserControl1()
      {
         InitializeComponent();
      }
   }
}
```

Látható, hogy az új osztály a *UserControl* –ból származik, ezenkívűl rendelkezik egy a tervező által generált metódussal, akárcsak a főprogramunk. A *UserControl* osztály a *ContainerControl* leszármazottja, így rendelkezik az összes olyan tulajdonsággal, amely ahhoz szükséges, hogy kezelni tudja a rajta („benne") elhelyezett vezérlőket (többek közt a *Form* osztály is az ő utódja).
A vezérlőt úgy fogjuk elkészíteni, hogy újrafelhasználható legyen, ezért nem égetjük bele a felhasználók adatait, hanem egy tulajdonságon keresztül adjuk meg azokat. A kész osztály így néz majd ki:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsControlLibrary2
{
   public partial class UserControl1 : UserControl
   {
      public UserControl1()
      {
         InitializeComponent();
      }
   }
}public partial class LoginControl : UserControl
{
   private Dictionary<string, string> loginData = null;

   public LoginControl()
   {
      InitializeComponent();
   }

   public Dictionary<string, string> LoginData
   {
      set { loginData = value; }
```

```
    }

    private void button1_Click(object sender, EventArgs e)
    {
        if (loginData.ContainsKey(textBox1.Text))
        {
            if (loginData[textBox1.Text] == textBox2.Text)
            {
                MessageBox.Show("Sikeres belépés!");
            }
            else errorProvider1.SetError(textBox2, "Rossz jelszó!");
        }
        else errorProvider1.SetError(textBox1, "Nem létezik a felhasználó!");
    }
}
```

A *LoginData* lesz az a tulajdonság, amelyen keresztül megadjuk a név – jelszó párokat.

Házi feladat: módosítsuk a programot, hogy tetszőleges metódust hívhasssunk sikeres belépés esetén. Kulcsszó: delegate.

A tervezési időben való támogatást pontosan ugyanúgy érhetjük el, mint a származtatott vezérlők esetében. Próbáljuk ki:

```
private void Form1_Load(object sender, EventArgs e)
{
    Dictionary<string, string> logData = new Dictionary<string, string>();
    logData.Add("Istvan", "istvan");
    logData.Add("Judit", "judit");
    logData.Add("Balazs", "balazs");
    loginControl1.LoginData = logData;
}
```

És az eredmény:

39. Rajzolás: GDI+

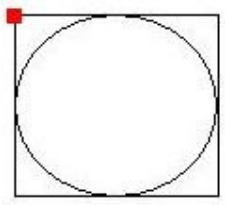
A GDI (Graphics Device Interface) a Microsoft Windows grafikus alrendszere, amelynek szolgáltatásait használja fel az operációs rendszer a felhasználói felület megjelenítéséhez. Használatához nem szükséges ismerni a hardvert, a GDI API automatikusan „kitalálja", hogy hogyan kell végrehajtania az utasításokat. A GDI+ a GDI Windows XP –ben megjelent utódja, ez már fejlettebb grafikus képességekkel rendelkezik.
A .NET Framework menedzselt felületet nyújt a GDI+ használatához, a *System.Drawing* névtér osztályain keresztül (korábban, amikor a *PrintDialog* –gal foglalkoztunk már találkoztunk ezekkel az osztályokkal).
Első programunkban rajzoljunk ki egy kört egy formra. Ehhez szükségünk lesz egy *Graphics* típusú objektumra (ún. *Device Context*, ez tartalmazza a megfelelő információkat az objektumról), amely kezeli az adott vezérlő grafikus megjelenését (ilyet bármelyik vezérlőtől kérhetünk). Ezt az objektumot a vezérlőn meghívott *CreateGraphics* metódussal kaphatjuk meg:

```
private void button1_Click(object sender, EventArgs e)
{
   Graphics g = this.CreateGraphics();
}
```

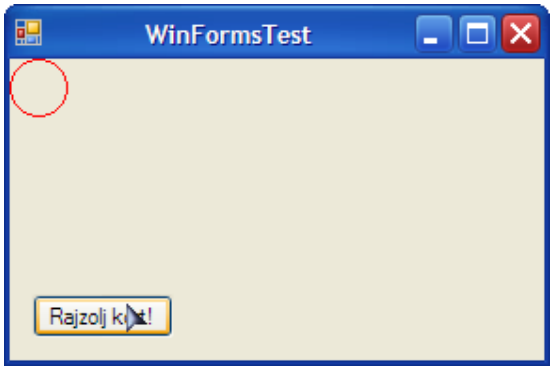
A rajzolást gombnyomásra indítjuk, a form *Load* eseményében nem lehet ezt megtenni (ha mégis „függetlenül" akarunk rajzolni, akkor válasszuk a *Paint* eseményt vagy a konstruktort).
A kört a *DrawEllipse* metódussal fogjuk megjeleníteni:

```
private void button1_Click(object sender, EventArgs e)
{
   Graphics g = this.CreateGraphics();
   g.DrawEllipse(new Pen(Brushes.Red), 0, 0, 30, 30);
}
```

Az első paraméterrel a színt állítjuk be, a második és harmadik pedig az alakzat kezdőkoordinátáját, ezek nem négyzetes alakzat esetén a síkidom köré rajzolt legszűkebb négyzet bal felső sarkát jelölik (az ábrán a pirossal jelzett pont):



Az utolsó két paraméterrel a szélességet és magasságot állítjuk be. A program eredménye:



A *Graphics* objektumot biztos felszabadítása érdekében érdemes *using* –gal együtt használni:

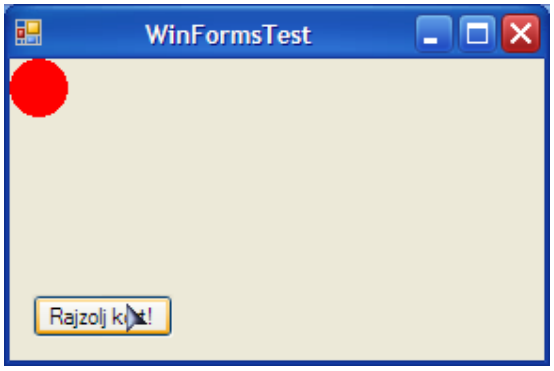
```
private void button1_Click(object sender, EventArgs e)
{
   using (Graphics g = this.CreateGraphics())
   {
      g.DrawEllipse(new Pen(Brushes.Red), 0, 0, 30, 30);
   }
}
```

A *DrawEllipse* (és más metódusok is) egy változata csak két paramétert vár, ahol a második egy *Rectangle* objektum, amely a fentebb már említett négyzetet jelképezi. A kört kirajzoló sort így is írhattuk volna:

```
g.DrawEllipse(new Pen(Brushes.Red), new Rectangle(0, 0, 30, 30));
```

Egy alakzatot ki is színezhetünk a megfelelő *Fill** metódussal. Tegyük ezt meg a körünkkel:

```
private void button1_Click(object sender, EventArgs e)
{
   using (Graphics g = this.CreateGraphics())
   {
      g.DrawEllipse(new Pen(Brushes.Red), new Rectangle(0, 0, 30, 30));
      g.FillEllipse(Brushes.Red, new Rectangle(0, 0, 30, 30));
   }
}
```



A *Fill** metódusok a kitöltendő területet várják második paraméterül.

39.1 Képek kezelése

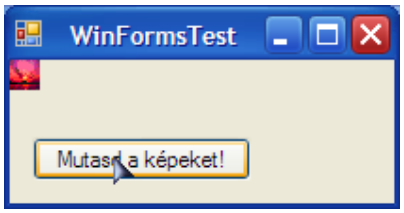
Képek beolvasására használhatjuk a *Bitmap* osztályt (amely a *System.Drawing* névtérben található). Húzzunk a fomra egy *PictureBox* –ot a form *Load* eseményébe pedig írjuk:

```
private void Form1_Load(object sender, EventArgs e)
{
   Bitmap bm = new Bitmap("Naplemente.jpg");
   pictureBox1.Image = bm;
}
```

A *Bitmap* elfogad stream –et is.

A GDI+ segítségével közvetlenül a fomra (vagy bármely vezérlőre) is kihelyezhetünk képeket, használhatjuk ehhez pl. az *ImageList* komponenst:

```
private void button1_Click(object sender, EventArgs e)
{
   using (Graphics g = this.CreateGraphics())
   {
      imageList1.Draw(g, new Point(0, 0), 0);
   }
}
```



A *Draw* metódus első paramétere a megfelelő *Graphics* objektum, második helyen áll a kezdőkoordináta végül pedig a kirajzolandó kép indexe.

Egy másik – függetlenebb – módja a közvetlen rajzolásnak az *Image* osztály használata:

```
private void button1_Click(object sender, EventArgs e)
{
   using (Graphics g = this.CreateGraphics())
   {
      Image img = Image.FromFile("Naplemente.jpg");
      g.DrawImageUnscaled(img, new Point(0, 0));
   }
}
```

Ez a metódus a képet eredeti nagyságában jeleníti meg, amennyiben ezt módosítani akarjuk használjuk a *DrawImage* metódust.

40. Drag and Drop

A feladat a következő: adott két *ListBox*, készítsünk programot, amely lehetővé teszi, hogy tetszés szerint húzzunk elemeket egyikből a másikba.
Az első dolog amit meg kell tennünk (persze a *ListBox* –ok formra helyezése után), hogy a vezérlők *AllowDrop* tulajdonságát igaz értékre állítjuk, ezzel engedélyezve a fejezet címében is szereplő metódust. Adjunk hozzá néhány elemet az első *ListBox* –hoz:

```
private void Form1_Load(object sender, EventArgs e)
{
   listBox1.Items.Add("Istvan");
   listBox1.Items.Add("Judit");
   listBox1.Items.Add("Balazs");
   listBox1.Items.Add("Viktoria");
}
```

A Drag and Drop műveletet akkor indítjuk el, amikor az egérgombot nyomva tartjuk egy elem felett, vagyis a *MouseDown* eseményben (mindkét vezérlő használhatja ugyanazt az eseményt).

```
private void listBox2_MouseDown(object sender, MouseEventArgs e)
{
   this.DoDragDrop(((ListBox)sender).SelectedItem.ToString(),
DragDropEffects.Move);
}
```

Az már látható, hogy a form fogja levezérelni a vonszolást. A *DoDragDrop* metódus első paramétere az az érték lesz, amit mozgatni akarunk, vagyis a kiválasztott elem, a második paraméter pedig a milyenséget fogja meghatározni. Ha most elindítjuk a programot, akkorr „megragadva” egy elemet az egérmutató megváltozik.
Tegyük fel, hogy jelezni akarjuk ha az éppen „húzásban” lévő elemmel egy vezérlő fölé érünk. Ezt az adott vezérlő *DragEnter* eseményében tehetjük meg (ez szintén egy olyan eseménykezelő lesz, amin a két vezérlő osztozhat). Mondjuk szinezzük át a hátteret. Azt se felejtsük el, hogy vissza kell állítani az eredeti színt, ezt a *DragLeave* eseményben fogjuk megcsinálni. A két metódus:

```
private void listBox2_DragEnter(object sender, DragEventArgs e)
{
   ((ListBox)sender).BackColor = Color.Red;
}

private void listBox2_DragLeave(object sender, EventArgs e)
{
   ((ListBox)sender).BackColor = Color.White;
}
```

A *DragOver* esemény fogja eldönteni, hogy engedélyezzük –e, hogy ledobhassuk az elemet:

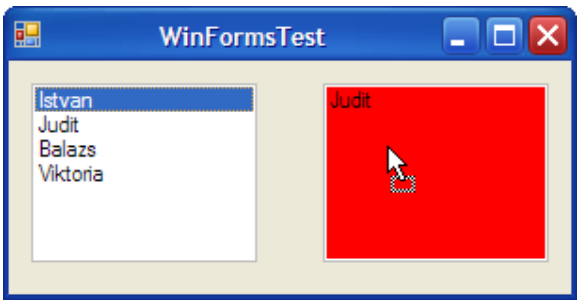
```
private void listBox2_DragOver(object sender, DragEventArgs e)
{
```

```
    e.Effect = DragDropEffects.Move;
}
```

Most már az egérmutató is megváltozik a *ListBox* –ok fölött. Egyetlen dolog van még hátra, méghozzá a dolog lényege, az információ elengedése. Amikor ezt megtesszük, akkor a *DragDrop* esemény lép működésbe:

```
private void listBox2_DragDrop(object sender, DragEventArgs e)
{
    ((ListBox)sender).Items.Add(e.Data.GetData(DataFormats.Text).ToString());
    ((ListBox)sender).BackColor = Color.White;
}
```

A háttérszínt vissza kell állítani, mert ebben az esetben a *DragLeave* esemény már nem következik be és piros maradna.



41. Windows Presentation Foundation

A WPF teljesen új alapokra helyezi a grafikus megjelenítést, míg eddig a Windows API függvényeit hívtuk meg és a GDI segítségével szoftveresen rajzoltuk ki az alkalmazások kezelőfelületét, most közvetlenül a DirectX fogja „hardverből" megtenni ezt. Ez a technológia egyúttal függetlenít minket (elvileg) az operációs rendszertől is.
Ez a váltás az elmúlt tizegynéhány év egyik legnagyobb jelentőségű fejlesztése, mivel teljesen új platformot kapunk. Eddig amikor Visual Basic, MFC stb. platformokon fejlesztettünk, ugyanazt tettük, csak éppen mindig fejlettebb környzetben. Ennek aztán meg is volt ára, hiszen a rendszer alapvető hiányosságait, gyengeségeit egyik sem tudta kiküszöbölni.
A WPF megjelenésével minden megváltozott, hiszen a DirectX egyetlen céllal jött világra, mégpedig villámgyors vizuális megjelenítésre.
Természetesen a WPF sem váltja meg a világot, például a felhasználó interaktivitását még mindig a Windows API kezeli, de a megjelenítést teljes egészében a WPF veszi át a GDI –től.

Egy fontos kérdés felmerülhet, mégpedig az, hogy mi van azokkal a számítógépekkel, amelyek nem rendelkeznek „erős" videókártyával? Nos, erre is van megoldás, a WPF képes szoftveresen is megjeleníteni az alkalmazást, természetesen ez kicsit lassabb lesz. A WPF arra törekszik, hogy a lehető legtöbb munkát bízza a célhardverre, de ezt nem erőlteti.

Ennek megfelelően a videókártya képességei alapján háromféle besorolása lehet egy rendszernek:

- Rendering Tier 0.: nincs hardveres gyorsítás, DirectX 7.0 vagy alatta.

- Rendering Tier 1.: alap (részleges) hardveres gyorsítás, DirectX 7.0 vagy felette, de 9.0 alatt.

- Rendering Tier 2.: teljes körű hardveres gyorsítás, a kártya mindent tud, amit a WPF elvár (Pixel Shader, Vertex Shader, stb.), DirectX 9.0 vagy fölötte.

Személtetésként lássuk a Rendering Tier 2. hardverigényét: DirectX 9.0 vagy fölötte, a videókártya legalább 120 MB memóriával rendelkezik, emellett támogatja a Pixel- és Vertex Shader 2.0 –t illetve legalább négy textúrázó egységgel bír.

További információkkal az MSDN megfelelő oldala szolgál:

- http://msdn.microsoft.com/en-us/library/ms742196.aspx

Nézzük meg ,mit kínál a megjelenítés mellett a WPF:

- Animációk
- Adatkötések
- Audio- és videó vezérlés
- Stílusok és sablonok
- Stb.

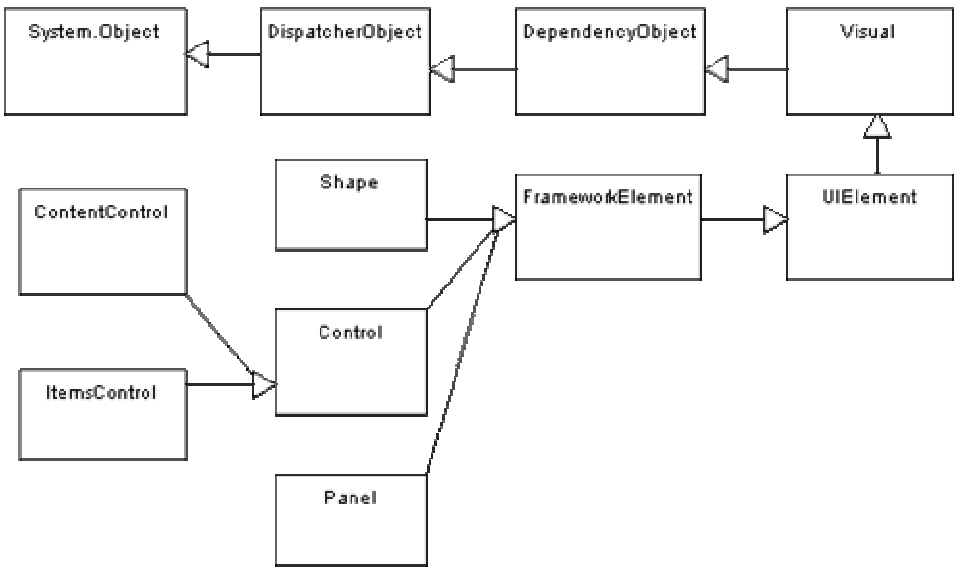
Még két a WPF –hez kötődő technológiáról is meg kell emlékeznünk, ezek az XBAP és a Silverlight. Első ránézésre mindkettő ugyanazt teszi: böngészőből érhetünk el egy felhasználói felületet. A megvalósítás azonban alapjaiban különbözik: az XBAP a WPF teljes funkcionalitását nyújtja, hiszen ugyanazt a .NET Framework –öt használja (ugyanakkor a nagytestvérhez képest vannak korlátai, mivel egy ún. sandbox –ban fut – később). Ezzel szemben a Silverlight a Flash ellenlábasaként egy miniatűr .NET FW telepítését igényli (~5MB), így a lehetőségek némileg korlátozottabbak, cserében platformfüggetlen (Macintosh és Windows rendszerek alá Silverlight néven érkezik, míg Linux alatt a Novell szolgáltatja az alternatívát, amelyet Moonlight –nak hívnak).

41.1 A WPF architectúra

Legfelsőbb szinten természetesen a C#/Visual Basic.NET/stb. nyelven megírt menedzselt kód tanyázik. Amikor egy ilyen porgramot futtatunk, akkor a kérések a WPF API –hoz futnak be, amelyet a számítógépen a PresentationFramework.dll, PresentationCore.dll és a WindowsBase.dll szerelvények jelképeznek. Ezek teljes egészében menedzselt kódon alapulnak.
Ők fogják továbbítani a kéréseket a Media Integration Layer –nek (MIL), amely az összekötő kapcsot jelképezi a .NET és a DirectX között. Ez a milcore.dll és WindowsCodecs.dll fileokban él (a milcore –t használja a Windows Vista is a grafikus felület megjelenítésére). A MIL már egy natív felületet biztosít és közvetlenül hívja a DirectX szolgáltatásait. Végül a WPF architechtúra részét képezi a Windows API is, amelyet korábban már említettünk.

41.2 A WPF osztályhierarchia

A WPF meglehetősen túlburjánzott osztályhierarchiával jött a világra, viszont ennek ismerete elengedhetetlen a későbbi fejlesztéshez, ezért álljon itt egy ismertető:



Természetesen mint minden mese ez is az *Object* osztállyal kezdődik, alatta eggyel pedig egy absztrakt osztály a *DispatcherObject* tartózkodik. Ez az osztály a

System.Threading névtérben található és ő fogja kontrollállni a szálak közti kommunikációt. Miért van erre szükség? A WPF az ún. Single-Thread Affinity (STA) modellel dolgozik, ami azt jelenti, hogy a felhasználói felületet egyetlen „főszál" irányítja (és ezért a vezérlőket is csak a saját szálukból lehet módosítani). Igen ám, de valahogyan a többi szálból érkező üzeneteket is fogadni kell pl. a billentyűzet vagy az egér megmozdulásait. És itt jön a képbe a *DispatcherObject*, amelynek szolgáltatásait meghívhatjuk más szálból is.

Következő a sorban a *DependencyObject*, amely az ún. Dependency Property –k lehetőségét adja hozzá a csomaghoz (erről hamarosan bővebben). A *DependencyObject* a *System.Windows* névtérben van.

A *System.Windows.Media* névtérbeli *Visual* absztrakt osztály felelős a tényleges megjelenítésért és a ő képezi a kapcsolatot a MIL –lel is.

A *Visual* –ból származó *System.Windows.UIElement* többek között az eseményekért felelős, de ő intézi a vezérlők elrendezését és a felhasználó interaktivitásának kezelését. is.

Az *UIElement* leszármazottja a *FrameworkElement*, amely ténylegesen megvalósítja azokat a szolgáltatásokat, amiket a WPF használ (pl.: az *UIElement* felelős az elrendezésért, de a *FrameworkElement* implementálja az ehhez szükséges tulajdonságokat, pl.: *Margin*).

A *Shape* és a *Panel* abszrakt osztályok, előbbi a primítív alakzatok (kör, négyszög, stb.) őse, utóbbi pedig azon vezérlők atyja, amelyek képesek gyermek vezérlőket kezelni (pl. *StackPanel*).

Most jön az érdekes rész, a *Control* osztály. A Windows Forms esetében megszokhattuk, hogy minden ami egy formon lehet, az vezérlő. A WPF világában ez nem így van, minden elemet *element* –nek nevezünk, néhány ezek közül pedig vezérlő (pl. minden vezérlő, amely fókuszba kerülhet illetve azok, amelyeket a felhasználó is használhat – ennek alapján amíg a Windows Forms –ban a *Label* vezérlő volt, itt már nem lenne az).
Ennek az osztálynak is van két leszármazottja, ezek közül a *ContentControl* az ami újdonság lesz: ez az osztály azokat a vezérlőket jelképezi, amelyeknek van „tartalma" (szöveg, kép, stb.).
Az *ItemsControl* azoknak a vezérlőknek az őse, amelyek maguk is több elemet tartalmaznak, pl. a *ListBox*.

41.3 XAML

Az XAML egy XML –ből elszármazott leírónyelv, amely segítségével definiálni tudjuk a WPF elemeinek elrendezését (illetve még más dolgokat is, de róluk a megfelelő időben).
Az XAML hasonló ahhoz, amit a Windows Forms esetében már láttunk, tehát a felhasználói felületet elválasztjuk a tényleges forráskódtól. Van azonban egy nagy különbség, mégpedig az, hogy ezúttal valóban különválasztva él a kettő, ellenben a WF megoldásától, ahol minden C#/VB:NET/stb. forráskódból állt, maximum azt nem kézzel írtuk, hanem a fejlesztőeszköz generálta..

XAML –t többféleképpen is létrehozhatunk, az első módszer, amit a jegyzet is használni fog az az, hogy egyszerűen kézzel begépelünk mindent (a Visual Studio nyújt IntelliSense támogatást ehhez). A másik kettő a Visual Studio tervező nézete (de csak az SP1) illetve a Microsoft Expression Blend nevű program, ezek grafikus felületet nyújtanak a tervezéshez.

41.3.1 Fordítás

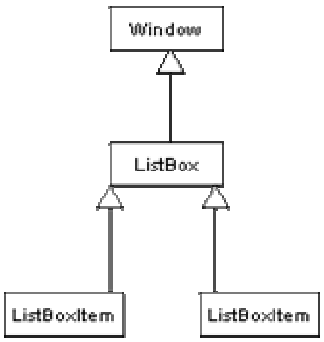
Mi történik vajon az XAML –lel, amikor lefordítunk egy WPF alkalmazást? Az XAML –t úgy tervezték, hogy jól olvasható és könnyen megérthető legyen, ennek megfelelően azonban egy kicsit „bőbeszédű". Hogy a programok futását felgyorsítsák valahogyan meg kell kurtítani ezt, ezért az XAML a fordítás után BAML –lé (Binary Application Markup Language) alakul, ami a forrás bináris reprezentációja. A BAML tokenizált, ami nagy vonalakban azt jelenti, hogy a hosszú utasításokat lerövidíti, így jóval gyorsabban tud betöltődni a file mint a nyers XAML.

41.3.2 Logikai- és Vizuális fa

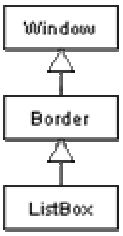
Vegyük az alábbi egyszerű XAML kódot:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="300">
  <Grid>
    <ListBox>
      <ListBoxItem Content="item1" />
      <ListBoxItem Content="item2" />
    </ListBox>
  </Grid>
</Window>
```

A formon létrehozunk egy *ListBox* –ot, amelyhez hozzáadunk két elemet. A logikai fa a következő lesz:



A magyarázat előtt lássuk a vizuális fa egy részletét is:



Hasonlítsuk össze a kettőt: az első esetben az alkalmazásnak azon elemeit láttuk, amelyek ténylegesen a szemünk előtt vannak, vagyis a logikai felépítést. A vizuális fa esetében már bejönnek a részletek is, azaz az egyes elemek kerete, színe, stb… (természetesen a fenti ábra messze nem teljes).

Fontos megjegyezni, hogy a két fa megléte nem igényli az XAML –t, akkor is léteznek, ha mindent forráskódból hozunk létre.

41.3.3 Felépítés

Nézzük meg még egyszer az előző kódot:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="300">
  <Grid>
    <ListBox>
      <ListBoxItem Content="item1" />
      <ListBoxItem Content="item2" />
    </ListBox>
  </Grid>
</Window>
```

Minden XAML dokumentum tetején egy olyan elem foglal helyet, ami alkalmas erre, egy ún. *top-level-element*. A WPF több ilyet is definiál:

- *Window*
- *Page*
- *Application*

Minden XAML dokumentum – ahogyan az XML dokumentumok is - egyetlen gyökérelemet engedélyez, tehát a top-level-element lezárása (</Window>) után már nem állhat semmi.

Lépjünk eggyel tovább. Az *x* névtér prefixxel a project számára biztosítunk egy sablont, a fenti esetben a code-behind osztályt adtuk meg. Ez egy nagyon érdekes dolog: a code-behind osztályt a fordító fogja generálni, de hála a parciális osztályok intézményének mi magunk is hozzáadhatjunk forráskódot, erről hamarosan.
Ezzel a prefixxel több más helyen is fogunk találkozni. Alapértelmezés szerint az elemek nem rendelkeznek külön azonosítóval (ahogy azt a Windows Forms esetében megszokhattuk). Ha szeretnénk őket megjelölni, akkor az *x:Name* attribútumot kell használnunk:

```
<ListBox x:Name="ListBoxWithName">
  <ListBoxItem Content="item1" />
  <ListBoxItem Content="item2" />
</ListBox>
```

Mostantól kezdve a „hagyományos” forráskódból is ezen a néven hivatkozhatunk a *ListBox* –ra. Ezenkívűl néhány osztály amely rendelkezik a *RunTimeNameProperty* attribútummal (például a *FrameworkElement* és így az összes leszármazotta) birtokol egy *Name* nevű tulajdonságot, amely segítségével futásidőben is beállíthatjuk az elem nevét:

```
ListBoxWithName.Name = "NewName";
```

Ekkor XAML –ből is használhatjuk ezt a tulajdonságot, tehát ez is egy helyes megoldás:

```
<ListBox Name=" ListBoxWithName " >
  <ListBoxItem Content="item1" />
  <ListBoxItem Content="item2" />
</ListBox>
```

Egy vezérlő tulajdonságait beállíthatjuk inline és explicit módon is. Lássunk mindkét esetre egy példát:

```
<TextBlock Text="Inline definíció" />

<TextBlock>
  <TextBlock.Text>
    Ez pedig az explicit definíció
  </TextBlock.Text>
</TextBlock>
```

Térjünk vissza a fejléchez. Az *xmlns* és *xmlns:x* attribútumok első ránézésre egy weboldalra mutatnak, de valójában nem. Ezek névterek, az első a WPF alapértelmezett névterére mutat, a második pedig az XAML kezeléshez szükséges dolgok névterére. Miért van ez így? Elsődlegesen azért, mert egy hagyományos XML dokumentum is így épül fel, ennek oka pedig az, hogy az egyes „gyártók” gond nélkül használhassák a saját elnevezésüket (a schemas.microsoft.com a Microsoft tulajdonában áll, és elég valószínűtlen, hogy más is használná).
A fordító ezekből a „címekből” tudni fogja, hogy hol keresse a számára szükséges osztályokat.

A gyökérelem rendelkezik még néhány attribútummal is, ezeknek a célját nem nehéz kitalálni. Ami fontos, az az, hogy egy XAML dokumentumban minden „element” egy létező WPF osztályra mutat, és minden attribútum az adott osztály egy tulajdonságára.

Következik egy *Grid* objektum, amely kiválóan alkalmas a vezérlők elrendezésére (ezt majd látni fogjuk). Ezenkívűl ez egy *Panel* leszármazott, így lehetnek gyermekei.

Rendben, most nézzük meg a code-behind kódot. Alapértelmezés szerint a Visual Studio ezt generálja:

```
namespace JegyzetWPF
{
   /// <summary>
   /// Interaction logic for Window1.xaml
   /// </summary>
   public partial class Window1 : Window
   {
      public Window1()
      {
         InitializeComponent();
      }
   }
}
```

Ez eléggé hasonlít arra, amit már láttunk akkor, amikor Windows Forms –szal dolgoztunk. Az *InitializeComponent* metódus azonban más funkcionalitással bír. Jobb egérgombbal klikkeljünk rajta és válasszuk a *Go To Definition* lehetőséget. Ami először feltűnhet, az az, hogy a vezérlőket nem itt adja hozzá, mint ahogy azt a WF tette. Ahhoz, hogy megtudjuk hogyan jön létre az alkalmazás vizuális felülete nézzük a következő sorokat:

```
System.Uri resourceLocater = new
System.Uri("/JegyzetWPF;component/window1.xaml", System.UriKind.Relative);

#line 1 "..\..\Window1.xaml"
System.Windows.Application.LoadComponent(this, resourceLocater);
```

Ezek az utasítások betöltik majd kibontják az erőforrásként beágyazott BAML –t, ebből pedig már fel tudja a WPF építeni a felhasználói felületet. Létrejön minden egyes vezérlő objektuma és hozzácsatolja az eseménykezelőket is.

42. WPF – események és tulajdonságok

Készítsünk egy egyszerű programot: egy *TextBox* és egy *Button* vezérlővel. A gomb megnyomásakor helyezzünk a *TextBox* –ba valamilyen szöveget. Az XAML a következő lesz:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="100" Width="300">
  <Grid>
    <TextBox
        x:Name="MessageTextBox"
        Width="100"
        Height="20"
        VerticalAlignment="Top" />

    <Button
        x:Name="MessageButton"
        Width="100"
        Height="30"
        Content="Click Me!"
        VerticalAlignment="Bottom"
        Click="MessageButton_Click" />
  </Grid>
</Window>
```

A *VerticalAlignment* tulajdonság a vertikális (függőleges) helyzetet állítja be, vagyis a *TextBox* felül lesz, a gomb pedig alul (létezik H*orizontalAlignment* is a vízszintes beállításokhoz). A későbbiekben megismerkedünk hatékonyabb rendezésekkel is, most ez is megteszi.
A gombhoz már hozzá is rendeltünk egy eseménykezelőt (ezt a Visual Studio kérésre létrehozza), mégpedig a *Click* eseményhez. Nézzük is meg:

```
private void MessageButton_Click(object sender, RoutedEventArgs e)
{
   MessageTextBox.Text = "Hello WPF!";
}
```

Gyakorlatilag nem különbözik attól, amit a Windows Forms esetében már láttunk, egy különbség azonban mégis van. A második paraméter típusa *RoutedEventArgs* lesz. Természetesen ez is egy *EventArgs* leszármazott, de utal valamire aminek sokkal nagyobb a jelentősége: eddig megszoktuk azt, hogy az üzenetet az az objektum küldi, amelyikkel történik valami. A WPF megváltoztatja ezt a helyzetet az ún. *routed events* bevezetésével, ami lehetővé teszi, hogy egy objektum egy eseményét - ha az közvetlenül nincs kezelve – „felutaztassuk” (vagy le) a vizuális fán, amíg valaki le nem kezeli. Nézzük meg a következő példát. Az XAML:

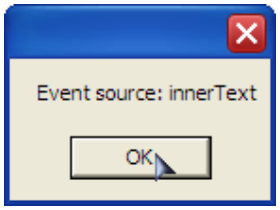
```
<Window x:Class="JegyzetTestWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="100" Width="300">
  <Grid>
```

```
    <Label Name="label1" MouseDown="label1_MouseDown">
      <TextBlock x:Name="innerText" Text="Hello" />
    </Label>
  </Grid>
</Window>
```

Mielőtt rátérnénk az eseménykezelésre vegyük észre, hogy a *Label* felépítését tetszés szerint alakíthatjuk, olyan módon, ahogyan azt a Windows Forms esetében csak usercontrol –ok segítségével érhettük el. Gyakorlatilag bármilyen vezérlőt beágyazhatunk ezen a módon, egyetlen feltétel van, mégpedig az, hogy az a vezérlő amibe beágyazunk *ContentControl* leszármazott legyen.
Most nézzük az eseményeket: az *UIElement* osztálytól minden leszármazottja örököl egy nagy csomó eseményt az egér és a billentyűzet kezeléséhez, ilyen a *MouseDown* is, amely az egérrel való klikkelést kívánja megszemélyesíteni. Nézzük az eseménykezelőt:

```
private void label1_MouseDown(object sender, MouseButtonEventArgs e)
{
   MessageBox.Show("Event source: " + ((FrameworkElement)e.Source).Name);
}
```

Ez is nagyon egyszerű, azt tudjuk, hogy minden vezérlő egyben *FrameworkElement* is és azt is, hogy annak pedig lekérdezhetjük a nevét. A *MouseButtonEventArgs* egy *RoutedEventArgs* leszármazott. A *Source* tulajdonság a logikai fa szerinti forrást adja vissza, míg az *OriginalSource* a vizuális fát veszi alapul. Amikor rákattintunk a *Label* –re, akkor felugrik egy *MessageBox*:



Igaz, hogy a *Label* eseményét kezeltük (volna) le, mégis a *TextBlock* volt a forrás. Nézzük meg egy kicsit közelebbről a routed event –eket. Amikor regisztrálunk egy ilyen eseményre, az választ egyet a három lehetséges stratégia közül:

- *Direct*: ez a legáltalánosabb, az esemény csakis a forrásobjektumban váltódik ki és csakis ő kezelheti.
- *Bubbling*: az esemény először a forrásobjektumban váltódik ki, majd elkezd felfelé vándorolni a vizuális fában, amíg talál egy kezelőt (vagy a gyökérhez nem ér).
- *Tunneling*: az esemény a gyökérelemben kezd és lefelé vándorol, amíg el nem éri a forrásobjektumot (vagy talál egy kezelőt).

Az esemény milyenségét a megfelelő *RoutedEventArgs* leszármazottól kérdezhetjük meg, a *RoutedEvent* tulajdonság *RoutedStrategy* tulajdonságán keresztül, amely egy felsorolt típusú érték (értelemszerűen, ha az eseménykezelő második paramétere *RoutedEventArgs* típusú, akkor közvetlenül hívhatjuk a *RoutedStrategy* –t).

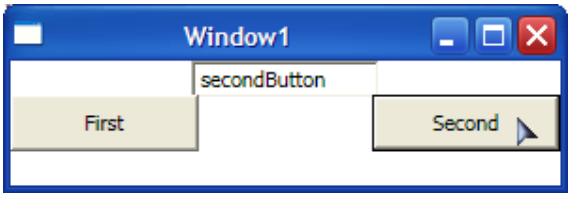
A példában a *TextBlock MouseDown* eseménye a *Bubbling* stratégiát használja, azaz elkezd felfelé vándorolni a vizuális fában, amíg talál egy barátságos eseménykezelőt (ez volt a *Label* –é).

Most nézzünk egy érdekesebb példát:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="100" Width="300" ButtonBase.Click="Window_Click">
  <Grid>
    <TextBox
        Name="messageTextBox"
        Width="100"
        Height="20"
        VerticalAlignment="Top" />

    <Button
        Name="firstButton"
        Width="100"
        Height="30"
        Content="First"
        HorizontalAlignment="Left"/>
    <Button
        Name="secondButton"
        Width="100"
        Height="30"
        Content="Second"
        HorizontalAlignment="Right"/>
  </Grid>
</Window>
```

A top-level-element *Window* minden olyan *Click* eseményt el fog kapni, amelyet egy *ButtonBase* leszármazott (*Button*, *CheckBox*, *RadioButton*) küld. Nézzük az eseménykezelőt:

```
private void Window_Click(object sender, RoutedEventArgs e)
{
   messageTextBox.Text = ((FrameworkElement)e.Source).Name;
}
```



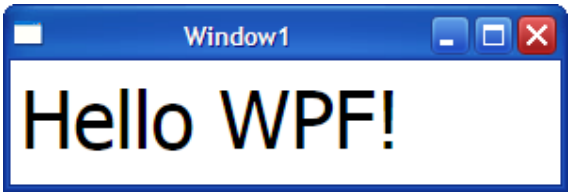
Mitől olyan érdekes ez a példa? Gondolkodjunk: a *Window* nem ismerheti a *Button* –t, mégis el tudja kapni a *Click* eseményt. Ez az ún. attached events (~ csatolt események) intézménye miatt van, ami lehetővé teszi egy vezérlőnek, hogy olyan eseményt kapjon el, amivel ő maga egyébként nem rendelkezik.
Attached event –et kódból is hozzáadhatunk egy vezérlőhöz:

```
window1.AddHandler(Button.ClickEvent, new RoutedEventHandler(Window_Click));
```

Egy másik fontos dolog a routed event –ek mellett a dependency property –k jelenléte. Ezek speciális tulajdonságok, amelyek hagyományos tulajdonságokba vannak becsomagolva, így olyan kód is kezelheti ezeket, amelyek nem ismerik a WPF –et.
A dependency property –k (~függőségi tulajdonság) valódi jelensége az, hogy a tulajdonság értéke függhet más tulajdonságoktól illetve a környezet sajátosságaitól (pl. az operációs rendszer). A hagyományos tulajdonságokat általában privát elérésű adattagok lekérdezésére és értékadására használjuk, vagyis amit betettünk a gépbe azt kapjuk vissza. A DP ezt a helyzetet megváltoztatja, mivel az aktuális érték több tényezőtől függ.
Nézzük a következő példát:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="100" Width="300">
  <Grid>
    <Label FontSize="42">
      Hello WPF!
    </Label>
  </Grid>
</Window>
```

Semmi különös, pontosan azt teszi, amit elvárunk, vagyis megjeleni a szöveg 42 –es betűmérettel:



Csavarjunk egyet a dolgon:

```
<Window x:Class="JegyzetTestWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="100" Width="300" FontSize="42">
  <Grid>
    <Label>
      Hello WPF!
    </Label>
  </Grid>
</Window>
```

Az egyetlen ami változott, az az, hogy a betűméretet a *Window* –nak adtuk meg. Ami érdekes, az pedig az, hogy a program pontosan ugyanazt csinálja, mint az előző.
Nézzük meg, hogy mi a titok. A *FontSize* tulajdonság így néz ki a *TextBlock* esetében:

```
[TypeConverter(typeof(FontSizeConverter)), Localizability(LocalizationCategory.None)]
public double FontSize
{
   get
   {
      return (double) base.GetValue(FontSizeProperty);
   }
   set
   {
      base.SetValue(FontSizeProperty, value);
   }
}
```

Koncentráljunk csak a getter/setter párosra. Elsőként, mi az a *FontSizeProperty*? A definíciója a következő:

```
[CommonDependencyProperty]
public static readonly DependencyProperty FontSizeProperty;
```

Amire még szükségünk van az az, hogy milyen osztályra hivatkozunk a *base* –zel? Ez pedig a *DependencyObject*.
Rendben, nézzük sorban, hogy mi is történik valójában a második példában: létrejön a *Label* és be kell állítania a betűméretet. Csakhogy ilyet mi nem adtunk meg, tehát amikor lekérdezné a *FontSize* tulajdonságot, akkor elvileg nem kapna semmit. Ekkor a tulajdonság az ős *DependencyObject* –től lekérdezi a tulajdonság aktuális értékét, ami vagy egy a fában nála magasabban elhelyezkedő (és őt magába foglaló) vezérlő megfelelő értéke lesz, vagy pedig a WPF alapértelmezése (vagy pedig, ha van általunk beállított értéke, akkor azt, ez az ún. lokális érték (local value)).
Ez fordítva is hasonlóan működik, ezt a következő példán láthatjuk is:

```
<Window x:Class="JegyzetTestWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="100" Width="300" FontSize="42">
  <Grid>
    <Label>
      Hello WPF!
    </Label>

    <Button x:Name="changeFontSize"
            Width="100"
            Height="30"
            Content="Change FontSize"
            Click="changeFontSize_Click"
            VerticalAlignment="Bottom" />
  </Grid>
</Window>
```

Semmi mást nem teszünk, minthogy megváltoztatjuk a *Window* betűméretét:

```
private void changeFontSize_Click(object sender, RoutedEventArgs e)
{
   window1.FontSize = 10;
```

```
}
```

Amikor a gombra kattintunk, egyúttal a *Label* betűmérete is megváltozik. Tehát a *SetValue* meghívásakor az alárendelt objektumok betűmérete is változik (amennyiben nem definiált sajátot).

Léteznek ún. megosztott tulajdonságok (shared property), amelyek bár különböző osztályokban definiáltak mégis ugyanarra a DP –re hivatkoznak. Ilyen például a szöveget megjeleníteni képes vezérlők *FontFamily* tulajdonsága is, amelyek tulajdonképpen mind a *TextElement* osztály DP –jét használják.

Hasonlóan a routed event –ekhez a dependency property –k is definiálnak egy attached property nevű képességet, amely lehetővé teszi, hogy egy másik vezérlő dependency property –jét használjuk. Ezzel leggyakrabban az elemek pozicionálásánál találkozunk majd:

```
<Window x:Class="JegyzetWPF.Window1"
        x:Name="window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="100" Width="300">
   <DockPanel>
      <Button DockPanel.Dock="Left" Width="100" Height="30" Content="Left" />
      <Button DockPanel.Dock="Right" Width="100" Height="30" Content="Right" />
   </DockPanel>
</Window>
```

Most a *Grid* helyett egy *DockPanel* –t használtunk, a gombok pozicióit pedig attached property –vel adtuk meg.

A későbbiekben még fogunk találkozni a fejezet mindkét szereplőjével, többek között megtanulunk majd saját WPF vezérlőt készíteni és hozzá a megfelelő DP –ket is.

43. WPF – Vezérlők elrendezése

A WPF a Windows Forms –tól eltérően egy merőben más módszert vezet be a vezérlők poziciónálásához. Előbbi a vezérlők koordinátáján alapult, míg a WPF egy leginkább a HTML –hez hasonló (de annál jóval összetettebb) megoldást kapott (ugyanakkor koordinátákon alapuló elrendezésre is van lehetőség, de nem az a standard).

A top-level-element (pl. a Window) egyetlen elemet képes magában hordozni, ami szükségessé teszi azt, hogy az az egy elem tudjon gyermek vezérlőket kezelni. Ezeket az elemeket konténernek (container) nevezzük és közös jellemzőjük, hogy a *Panel* absztrakt osztályból származnak (lásd: WPF osztályhierarchia). Hat darab ilyen osztályunk van, nézzük meg a jellemzőiket:

- *Grid*: a leggyakrabban használt, az elemeit sorok és oszlopok szerint osztja be.
- *UniformGrid*: minden elem ugyanolyan méretű cellákban foglal helyet, viszonylag ritkán használt.
- *StackPanel*: vízszintesen vagy függőlegesen helyezkednek el az elemei, általában egy másik konténerben foglal helyet, az alkalmazás kisebb szekcióinak kialakítására használjuk.
- *DockPanel*: az elemeit az egyes oldalaihoz rendelhetjük.
- *WrapPanel*: egyenes vonalban helyezkednek el az elemei (az Orientation tulajdonságán keresztül vízszintes vagy függőleges lehet a vonal).
- *Canvas*: koordináta alapú elhelyezkedést tesz lehetővé

Az egyes konténerek egymásba ágyazhatóak, vagyis készíthetünk egy *Grid* –et, ami tartalmaz egy *StackPanel* –t, ami tartalmaz egy *Canvas* –t, és így tovább.

Az egyes elemek elhelyezkedésére és méretére is van néhány szabály, amelyeket a WPF automatikusan alkalmaz (természetesen ezeket felülbírálhatjuk), ezek a következőek:

- Egy elem méretét nem szükséges explicit megadni, az minden esetben megfelelő nagyságú lesz a tartalmához (de beállíthatunk minimum és maximum méreteket is).

- A konténer elemek elosztják a gyermekeik közt a felhasználható teret.

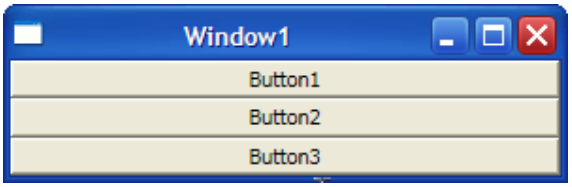
A következőkben megnézzük ezeknek a konténereknek a használatát.

43.1 StackPanel

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="100" Width="300">
  <StackPanel>
    <Button>Button1</Button>
```

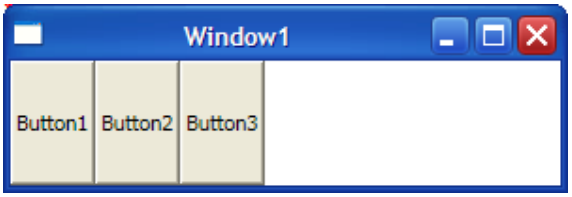
```
    <Button>Button2</Button>
    <Button>Button3</Button>
  </StackPanel>
</Window>
```

Az eredmény:



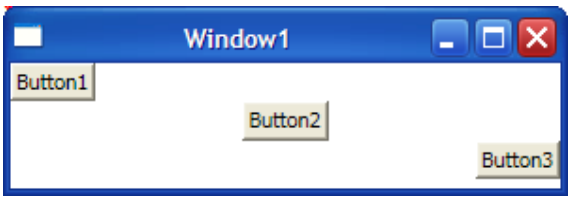
Az elemek függőlegesen követik egymást.
Az *Orientation* tulajdonság megváltoztatásával vízszintes elrendezést kapunk:

```
<StackPanel Orientation="Horizontal">
```



Az látható, hogy a vezérlők minden esetben teljes mértékben kitöltik a saját helyüket. Amennyiben megadjuk a *Horizontal/VerticalAlignment* tulajdonságot, az elemek a lehető legkisebb méretet veszik föl (vagyis igazodnak a tartalomhoz):

```
<StackPanel>
  <Button HorizontalAlignment="Left">Button1</Button>
  <Button HorizontalAlignment="Center">Button2</Button>
  <Button HorizontalAlignment="Right">Button3</Button>
</StackPanel>
```



Az Alignment tulajdonságokból az *Orient* tulajdonságnak megfelelően mindig csak egy használható fel.

A *StackPanel* és *WrapPanel* sajátja, hogy az elemeit ömlesztve, térköz nélkül helyezi el. Ezen a problémán segíthetünk „margók” beállításával.:

```
<Window x:Class="JegyzetWPF.Window1"
        x:Name="window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

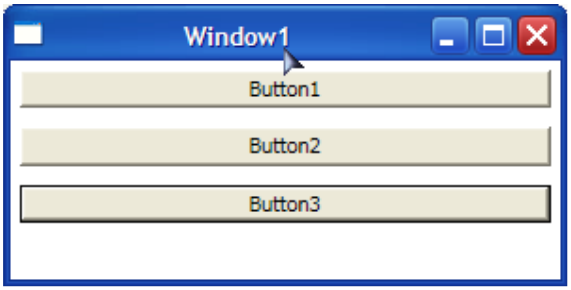
```
   Title="Window1" Height="150" Width="300">
  <StackPanel Orientation="Vertical">
    <Button Margin="5">Button1</Button>
    <Button Margin="5">Button2</Button>
    <Button Margin="5">Button3</Button>
  </StackPanel>
</Window>
```

A WPF a szélesség, magasság, margó, stb. meghatározására ún. device independent unit –okat használ (~rendszerfüggetlen egység). Egy DIU egy angol hüvelyknek az 1/96 –od része: ~0,2 mm (egy hüvelyk: 2,54 cm).
Azért ennyi, mert az alapértelmezett Windows DPI (dot-per-inch, az egy hüvelyken elhelyezkedő pixelek száma) éppen 96, vagyis ebben az esetben egy DIU pontosan egy pixelnek felel meg, mivel a WPF a következő képletet használja egy egység meghatározására:

Fizikai méret = DIU x DPI

Ahol a DPI az éppen aktuális rendszer DPI –nek felel meg.
Valójában a DPI szám a megjelenítő eszköztől függ, egy nagyfelbontású és nagyméretű monitornak sokkal nagyobb a DPI értéke.

A fenti példában a gombok körül 5 egység méretű térközt állítottunk be. Az eredmény:



A CSS –t ismerők ismersőnek fogják találni a helyzetet, ugyanis a fenti esetben a térköz nagysága mind a négy oldalra érvényes. Beállíthatjuk ezt persze egyenként is:

```
<Button Margin="5, 10, 5, 10" >Button1</Button>
```

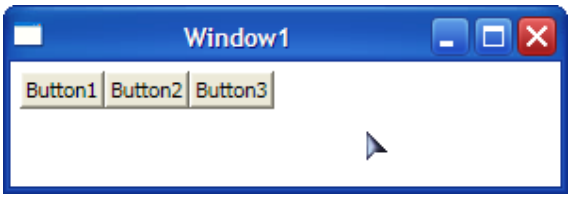
Az oldalak sorrendje balról jobbra: balra, fönt, jobbra, lent (ez eltér a CSS –től, ahol a sorrend: fönt, jobbra, lent, balra).

43.2 WrapPanel

A *WrapPanel* hasonlóan működik mint a *StackPanel*, csak itt egy sorban/oszlopban több elem is lehet:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="100" Width="300">
   <WrapPanel Margin="5">
     <Button>Button1</Button>
     <Button>Button2</Button>
     <Button>Button3</Button>
   </WrapPanel>
</Window>
```



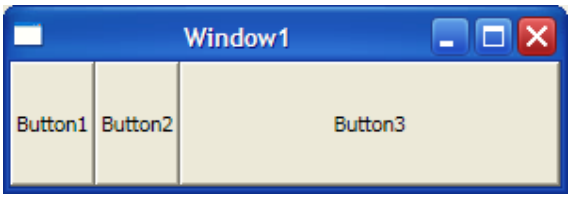
Hasonlóan mint a társánál az *Orientation* tulajdonságon keresztűl tudjuk a vízszintes/függőleges elrendezést állítani.

43.3 DockPanel

Vegyük a következő XAML –t:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="100" Width="300">
   <DockPanel>
     <Button>Button1</Button>
     <Button>Button2</Button>
     <Button>Button3</Button>
   </DockPanel>
</Window>
```

Vagyis egy *DockPanel* –ben elhelyeztünk három gombot, de semmi mást nem állítottunk. Ekkor ez történik:

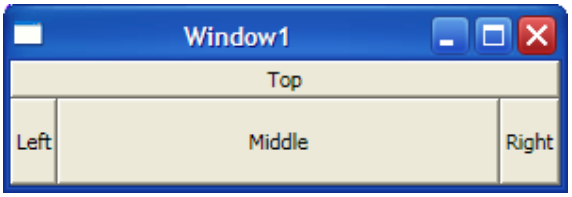


Az első két gomb függőlegesen kitölti a helyet, de vízszintesen a tartalomhoz igazodik a méretük. Nem így tesz az utolsó gomb, ami mindkét irányban elfoglalja a maradék helyet. Ez azért van, mert az elemei minden esetben ki kell töltsék a lehető legnagyobb helyet, hacsak mást nem adunk meg (tehát az első kettő gomb még normális, de a harmadikhoz érve látja, hogy van még hely és automatikusan megnyújtja).

Nézzünk megy egy másik példát is:

```
<DockPanel>
  <Button DockPanel.Dock="Top">Top</Button>
  <Button DockPanel.Dock="Left">Left</Button>
  <Button DockPanel.Dock="Right">Right</Button>
  <Button>Middle</Button>
</DockPanel>
```

Most egy - az előző fejezetben megismert - attached property –t használtunk, hogy meghatározzuk, hogy a gombok a *DockPanel* melyik oldalához igazodjanak.
Az eredmény ez lesz:

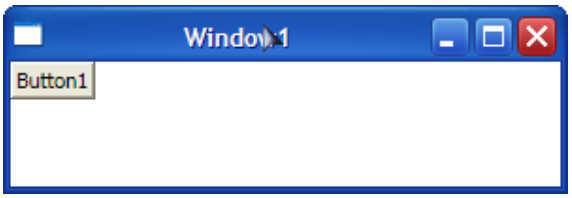


Most már könnyen meghatározhatjuk a viselkedését: az elemeket balra, fönt, jobbra, lent sorrendben rajzolja ki, és ha az utolsó elemhez érve még van hely, akkor azt maradéktalanul kitölti (persze, ha van beállítva pl. *Margin* tulajdonság, akkor azt figyelembe veszi).

43.4 Canvas

A *Canvas* lehetővé teszi, hogy a rajta elhelyezkedő vezérlőket egy koordinátarendszer segítségével pozicionáljuk:

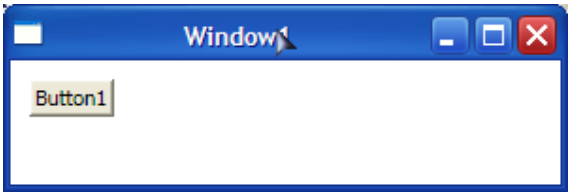
```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="100" Width="300">
  <Canvas>
    <Button>Button1</Button>
  </Canvas>
</Window>
```



Látható, hogy az alapértelmezett koordináták szerint az elem helye a bal felső sarok, vagyis a (0;0) pont. Szintén attached property –kel tudjuk megadni a pontos helyet:

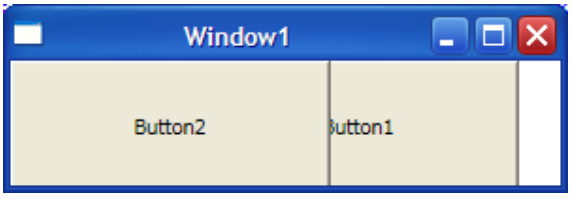
```
<Canvas>
  <Button Canvas.Left="10" Canvas.Top="10">Button1</Button>
</Canvas>
```

A példában a gombot a bal oldaltól és a felső résztől is tíz egységgel toltuk el (a viszonyítást mindig az adott vezérlő adott oldalához számoljuk):



Ha több egymást esetleg eltakaró elem van a *Canvas* –on, akkor a *ZIndex* tulajdonságát is beállíthatjuk. Ez alapértelmezetten minden elemnek nulla, és a WPF először mindig a legalacsonyabb z-indexű elemet rajzolja ki (tehát a magasabb indexűek eltakarják az alacsonyabbakat):

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="100" Width="300">
  <Canvas>
    <Button Canvas.Left="100" Width="170" Height="70">Button1</Button>
    <Button Panel.ZIndex="1" Width="170" Height="70">Button2</Button>
  </Canvas>
</Window>
```



A *ZIndex* –et a *Panel* –tól örökli, és csakis azon keresztűl érhetjük el.

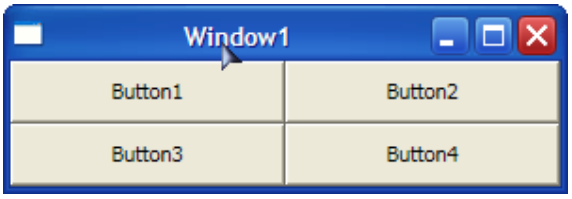
43.5 UniformGrid

Az *UniformGrid* előre definiált oszlopokba és sorokba (cellákba) rendezi a vezérlőit (valójában a méret megadására sincs szükség, automatikusan is elrendezi őket). Minden egyes vezérlő ugyanannyi helyet foglal el, pontosabban minden cella mérete egyenlő. Ha valamelyik elemnek explicit megadjuk a szélességét vagy magasságát, akkor az érvényesülni is fog, de ekkor két lehetőség van: vagy kisebb mint amekkora egy cella, ekkor a cella közepén foglal helyet és körülette megfelelő méretű térköz lesz, vagy pedig nagyobb mint egy cella, ekkor a *ZIndex* –e módosul és a „kilógó” része a többi vezérlőt alatt lesz. Nézzünk ezekre néhány pédát:

A „normális” eset:

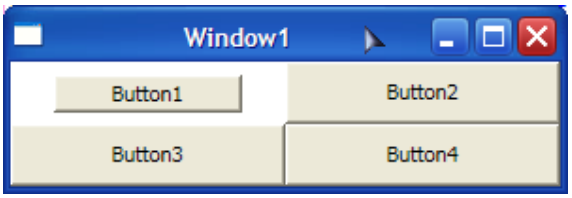
```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="100" Width="300">
   <UniformGrid>
     <Button>Button1</Button>
     <Button>Button2</Button>
     <Button>Button3</Button>
     <Button>Button4</Button>
   </UniformGrid>
</Window>
```



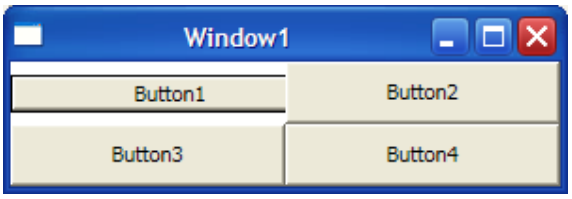
Explicit méretmegadás, de kissebb:

```
<UniformGrid>
   <Button Width="100" Height="20">Button1</Button>
   <Button>Button2</Button>
   <Button>Button3</Button>
   <Button>Button4</Button>
</UniformGrid>
```



Explicit megadás, de nagyobb mint kellene:

```
<UniformGrid>
   <Button Width="170" Height="20">Button1</Button>
   <Button>Button2</Button>
   <Button>Button3</Button>
   <Button>Button4</Button>
</UniformGrid>
```



43.6 Grid

A *Grid* a WPF legerőteljesebb (és leggyakrabban használt) container osztálya. Amit az előzőekkel meg tudunk csinálni, azt vele is reprodukálni tudjuk.

Hasonlóan a *UniformGrid* –hez itt is sorok és oszlopok szerint rendezzük a területet, de most megadhatjuk azok számát, méretét illetve egy adott cellába több elemet is tehetünk.
Egy *Grid* sorait illetve oszlopait a *RowDefinitions* és a *ColumnDefinitions* tulajdonságain keresztül tudjuk beállítani, amelyek *Row/ColumnDefinitionCollection* típusú gyűjteményeket kezelnek. Ezeket legyakrabban XAML –ből állítjuk be, a következő módon:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="100" Width="300">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="40" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="100" />
      <ColumnDefinition Width="150" />
      <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
  </Grid>
</Window>
```

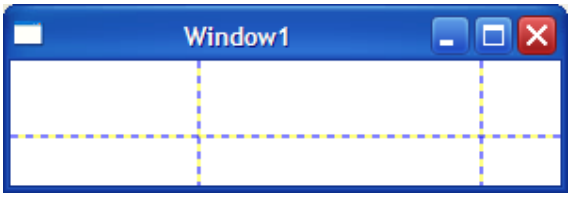
Két sort és három oszlopot készítettünk. Az nem meglepetés, hogy az egyes oszlopok/sorok definíciója *Column/RowDefinition* típusú.
Az oszlopoknál csakis a szélességet, a soroknál csakis a magasságot állíthatjuk. Ha ezeknek az értékeiknek a helyére csillagot írunk, az azt jelenti, hogy az adott sor/oszlop ki fogja tölteni a megmaradó összes helyet.
Viszont nem kötelező megani ezeket az értékeket, ekkor a *Grid* egyenlő nagyságú cellákra osztja a felületet, a sorok és oszlopok számának függvényében.
Ezenkívűl rábízhatjuk a *Grid* –re is a méret beállítását, ha a szélesség/magasság értéknek „auto” –t adunk, ekkor pontosan akkora lesz, amekkora kell (vagyis a benne lévő vezérlő méretéhez igazodik).

Ez utóbbi két esetben hasznosak lehetnek a *MinWidth/MaxWidth*, *MinHeight/MaxHeight* tulajdonások, amelyekkel a sorok/oszlopok mérethatárait állíthatjuk be.

Ha most elindítanánk a programot, akkor semmit sem látnák, ami nem meglepő, hiszen nem lenne túlságosan vonzó egy olyan alkalmazás, amelynek négyzetrácsos a felülete. Szerencsére a fejlesztés ellenőrzésének céljára lehetőségünk van bekapcsolni, hogy látszódjon ez a rács, a *Grid ShowGridLines* tulajdonságán keresztül:

```
<Grid ShowGridLines="True">
```

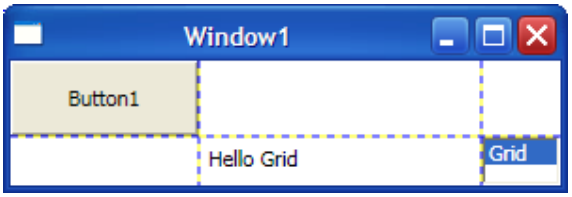
Ekkor az eredmény:



Helyezzünk el most néhány vezérlőt:

```
<Window x:Class="JegyzetWPF.Window1"
        x:Name="window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="100" Width="300">
   <Grid ShowGridLines="True">
      <Grid.RowDefinitions>
        <RowDefinition Height="40" />
        <RowDefinition Height="*" />
      </Grid.RowDefinitions>

      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="100" />
        <ColumnDefinition Width="150" />
        <ColumnDefinition Width="*" />
      </Grid.ColumnDefinitions>

      <Button Grid.Column="0" Grid.Row="0">Button1</Button>

      <Label Grid.Column="1" Grid.Row="1">Hello Grid</Label>

      <ListBox Grid.Column="2" Grid.Row="1">
        <ListBoxItem Content="Grid" />
      </ListBox>
   </Grid>
</Window>
```

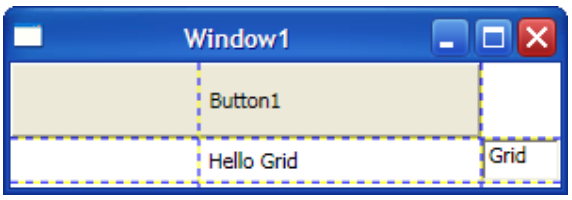
Ismét attached property –ket használtunk. Látható, hogy mind az oszlopok, mind a sorok számozása nullától kezdődik. Most így néz ki a programunk:



Fontos, hogy minden a *Grid* –en elhelyezni kívánt vezérlőnek be kell állítani a helyét, különben nem jelenik meg (de elég csak az egyiket – oszlop vagy sor - megadni, ekkor a WPF a hiányzó értékhez automatikusan nullát rendel).

A *Grid.RowSpan* és a *Grid.ColumnSpan* tulajdonságokkal azt adhatjuk meg, hogy a vezérlő hány oszlopot illetve sort foglaljon el:

```
<Button Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="2">Button1</Button>
```



Most a gomb két oszlopot kapott.

44. WPF – Vezérlők

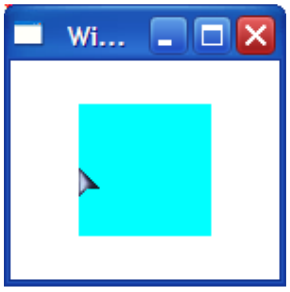
Ebben a fejezetben megismerkedünk a WPF vezérlőivel. Sokuk ismerős lesz a korábbi WinForms –szal foglalkozó fejezetből, legalábbis célját és működését tekintve (a használatuk, a létrehozásuk már nem biztos).
Már korábban megtanultuk, hogy nem minden elem vezérlő a WPF világában, hanem csak azok, amelyek képesek felhasználói interaktivitást kezelni.
Azt is tudjuk, hogy a vezérlők a *System.Windows.Control* osztályból származnak, ami felvértezi őket néhány alapértelmezett, mindegyikükre jellemző képességgel (ilyen pl. az, hogy formázhatjuk a vezérlő „feliratát", szöveges tartalmát).

A vezérlők bemutatásánál néhány elem kimarad, mivel valós haszna csak egy későbbi fejezet tudásanyagával lesz, így majd a megfelelő helyen ismerkedünk meg velük.

44.1 Megjelenés és szövegformázás

Megjelenés szempontjánból a WPF jóval flexibilisebb mint a WinForms. Ebben a fejezetben a vezérlők vizuális megjelenését fogjuk „kicsinosítani". Két általános vezérlőt fogunk ehhez használni, a *Button* –t és a *Label* –t (illetve a szinezés demonstrálásához egy egyszerű alakzatot a *Rectangle* –t).
Minden vezérlő magáévá teszi az előtér és a háttér fogalmát. Utóbbi a vezérlő tényleges felszíne, míg előbbi általában a szöveg. Windows Forms –ban ezeket a *Color* osztály segítségével állítottuk, de most rendelkezésünkre áll valami sokkal jobb, mégpedig a *Brush*. Ez egy absztrakt osztály, ami előrevetíti, hogy több gyermeke is van, amelyek mindegyike más-más lehetőséget biztosít számunkra.
A lehető legegyszerűbb esetben használhatjuk úgy a színeket, mint ahogy azt a *Color* osztály esetében tettük, csak most ugyanazt a funkciót a *Brushes* osztály tölti majd be. Vezérlők esetében a hátteret a *BackGround* tulajdonságon keresztűl állíthatjuk be, ugyanezt alakzatoknál a *Fill* tulajdonság fogaj szolgáltatni. A kettőben a közös, hogy *Brush* típusú (vagy leszármazott) objektumot várnak. Készítsünk egy négyzetet:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="150" Width="150">
  <Grid>
    <Rectangle x:Name="brushRectangle"
               Width="70"
               Height="70"
               VerticalAlignment="Center"
               Fill="Cyan"/>
  </Grid>
</Window>
```



Ugyanezt megtehetjük kódból is:

```
brushRectangle.Fill = Brushes.Cyan;
```

Az a *Brush* leszármazott, amelyet itt használtunk *SolidColorBrush* névre hallgat. Például írhatnánk ezt is a fenti helyett:

```
SolidColorBrush scb = new SolidColorBrush();
scb.Color = Color.FromRgb(100, 10, 100);
brushRectangle.Fill = scb;
```

Ugyanezt megtehetjük XAML –ből is:

```
<Rectangle x:Name="brushRectangle"
        Width="70"
        Height="70"
        VerticalAlignment="Center">
  <Rectangle.Fill>
    <SolidColorBrush>
      <SolidColorBrush.Color>
        <Color A="255" R="100" G="10" B="100" />
      </SolidColorBrush.Color>
    </SolidColorBrush>
  </Rectangle.Fill>
</Rectangle>
```

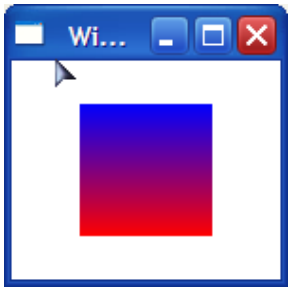
Itt csak arra kell figyelni, hogy ARGB értéket kell megadnunk, ahol A az átlátszóság mértékét jelzi (ha 0 akkor lesz teljesen átlátszó, a maximálisan felvehető értéke 255). Ha előre definiált színeket akarunk használni, akkor egyszerűbb a dolgunk:

```
<Rectangle x:Name="brushRectangle"
        Width="70"
        Height="70"
        VerticalAlignment="Center">
  <Rectangle.Fill>
    <SolidColorBrush Color="Cyan" />
  </Rectangle.Fill>
</Rectangle>
```

A *LinearGradientBrush* két (vagy több) színt „olvaszt” össze. A színnel feltöltött területeknek megadhatunk egy dőlészszöget (gradiens tengely) amely a színek haladási irányát jelöli majd.

```
LinearGradientBrush lgb = new LinearGradientBrush(
    Color.FromRgb(0, 0, 255),
    Color.FromRgb(255, 0, 0),
    90);
brushRectangle.Fill = lgb;
```

A harmadik paraméter a tengely dőlésszöge:



XAML:

```
<Rectangle.Fill>
  <LinearGradientBrush>
    <GradientStop Color="Blue" Offset="0.0" />
    <GradientStop Color="Red" Offset="0.5" />
  </LinearGradientBrush>
</Rectangle.Fill>
```

Az *Offset* tulajdonság a színek átmenetének a választóvonala, a bal felső sarkot jelöli a 0.0, míg a jobb alsót az 1.1. Alapértelmezetten a gradiens tengely ezt a két vonalat köti össze. A tengelyt a *StartPoint* illetve *EndPoint* tulajdonságokkal tudjuk beállítani:

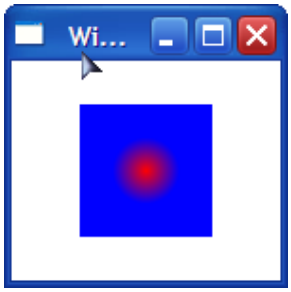
```
<Rectangle.Fill>
  <LinearGradientBrush StartPoint="0.5, 0" EndPoint="0.5, 1">
    <GradientStop Color="Blue" Offset="0.0" />
    <GradientStop Color="Red" Offset="0.5" />
  </LinearGradientBrush>
</Rectangle.Fill>
```

A *RadialGradientBrush* hasonló mint az előző, de ő a színeket kör alakzatban olvasztja össze. Hasonlóan mint a *LinearGradientBrush* –nál, itt is *GradientStop* objektumokkal tudjuk szabályozni a színek határait:

```
RadialGradientBrush rgb = new RadialGradientBrush();
rgb.GradientStops.Add(
    new GradientStop(Color.FromRgb(255, 0, 0),
      0.0));
rgb.GradientStops.Add(
    new GradientStop(Color.FromRgb(0, 0, 255),
      0.5));
brushRectangle.Fill = rgb;
```

Most egy kicsit másként adtuk hozzá színeket, közvetlenül a *GradientStops* tulajdonságot használva (eddig a konstruktorban adtuk meg őket). Ugyanez XAML –ben:

```
<Rectangle.Fill>
    <RadialGradientBrush>
       <GradientStop Color="Red" Offset="0.0" />
       <GradientStop Color="Blue" Offset="0.5" />
    </RadialGradientBrush>
</Rectangle.Fill>
```



Az *ImageBrush* –sal képeket tudunk kezelni:

```
ImageBrush ib = new ImageBrush();
ib.ImageSource = new BitmapImage(new Uri("test.jpg", UriKind.Relative));
brushRectangle.Fill = ib;
```

XAML:

```
<Rectangle.Fill>
  <ImageBrush ImageSource="test.jpg" />
</Rectangle.Fill>
```

Ha rosszul adtuk meg a kép nevét *XamlParseException* –t kapunk.

A vezérlőknek egyszerűbben állíthatjuk be a színezését, mivel ezek a *Control* osztálytól öröklik a *Background* és Foreground tulajdonságukat. Ezek valamilyen *Brush* leszármazottat várnak:

```
<Button Background="Blue" Foreground="Red" Width="100" Height="30"
Content="Click" />
```

A következő weboldalon több példát és leírást találunk:

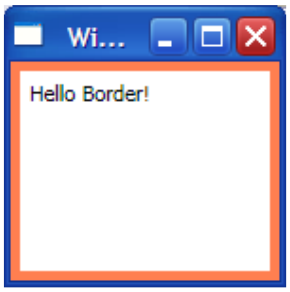
http://msdn.microsoft.com/en-us/library/aa970904.aspx

Nemcsak alakzatokat, vagy vezérlőket színezhetünk, hanem megadhatunk nekik keretet (Border) is, amely szintén kaphat egyedi színezést:

```
<Border BorderBrush="Coral" BorderThickness="5">
  <Rectangle x:Name="brushRectangle"
        Width="70"
        Height="70"
        VerticalAlignment="Center">
  </Rectangle>
</Border>
```

Ez a keret különnálló a *Rectangle* –től, mivel annak nincs saját kerete. Vannak vezérlők, amelyek viszont rendelkeznek ilyennel:

```
<Label
   BorderBrush="Coral"
   BorderThickness="5">
     Hello Border!
</Label>
```



A *BorderBrush* tulajdonság a keret színét, míg a *BorderThickness* a keret vastagságát jelöli.
Egy *Border* objektum csakis egyetlen gyermek vezérlővel rendelkezhet, ezért ha többet akarunk összefogni akkor (pl.) egy *Panel* –be tegyük őket és azt jelöljük meg kerettel.
Nem csak szögletes keretet adhatunk, meg, hanem a *CornerRadius* tulajdonsággal kerekíthatjük is:

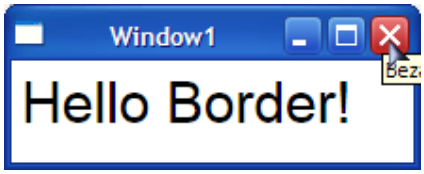
```
<Border BorderBrush="Black" BorderThickness="5" CornerRadius="45">
    <Button Width="100" Height="30" Content="Click" />
</Border>
```

Egyes vezérlők rendelkeznek saját kerettel is, a *Button* például ilyen:

```
<Button BorderBrush="Black" BorderThickness="5" Width="100" Height="30"
Content="Click" />
```

Hasonlóan Windows Forms –hoz a WPF is a jól ismert tulajdonságokat kínálja a szövegformázásokhoz:

```
<Label FontFamily="Arial" FontSize="30">
    Hello Border!
</Label>
```



44.2 Vezérlők

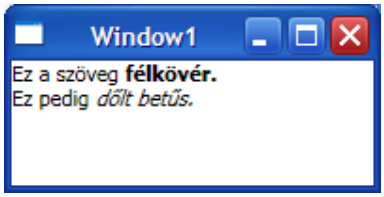
Ebben a fejezetben megismerkedünk az alapvető vezérlőkkel. Funkciójuk szinte semmit nem változott a WinForms –hoz tapasztaltakhoz képest, de tulajdonságaik illetve deklarációjuk igen. A példákban jobbára a külső megjelenéssel foglalkozunk, hiszen a vezérlők képességei nagyjából ugyanazok, mint amit már ismerünk.

44.2.1 TextBlock és Label

Ez a két vezérlő első ránézésre ugyanazt a célt szolgálja, de valójában nagy különbség van közöttük. Sőt, a *TextBlock* kakukktojás ebben a fejezetben, hiszen ő nem vezérlő, de a *Label* –hez való hasonlósága miatt itt a helye. A *TextBlock* –ot

arra találták ki, hogy kis mennyiségű szöveges (akár formázott) tartalmat jelenítsen meg:

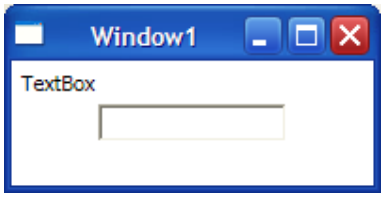
```
<Window x:Class="JegyzetWPF.Window1"
        x:Name="window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xml:lang="hu-HU"
    Title="Window1" Height="100" Width="200">
   <Grid>
      <TextBlock>
         Ez a szöveg <Bold>félkövér.</Bold><LineBreak />
         Ez pedig <Italic>dőlt betűs.</Italic>
      </TextBlock>
   </Grid>
</Window>
```



A szövegformázást a *Label* is támogatja, de őt nem ezért szeretjük, hanem mert támogatja a gyorsbillentyűket:

```
<Window x:Class="JegyzetWPF.Window1"
        x:Name="window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xml:lang="hu-HU"
    Title="Window1" Height="100" Width="200">
   <Grid>
      <Label Target="{Binding ElementName=tb1}">_TextBox</Label>
      <TextBox x:Name="tb1" Width="100" Height="20" />
   </Grid>
</Window>
```

A *Label Target* tulajdonságának megadhatjuk a hozzá kötött vezérlő nevét, amelyre a megfelelő billentyűkombinációval a fókusz kerül. Az adatkötésekről (amelyet a *Binding* osztály vezérel majd) később beszélünk majd. A gyorsbillentyűt a kiválasztott karakter elé írt alulvonással tudjuk kijelölni, a példában az ALT+t kombináció azt eredményezi, hogy a fókusz a *TextBox* –ra kerül.



A Visual Studio –ban az IntelliSense nem támogatja az adatkötéseket, így a Binding –et sem fogja felajánlani, nekünk kell beírni.

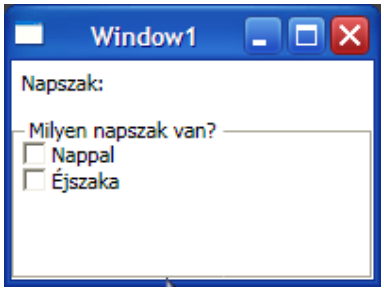
Ezenkívűl a *Label* egy *ContentControl* leszármazott is, erről a következő fejezetben olvashatunk többet.

44.2.2 CheckBox és RadioButton

Vegyük a következő XAML kódot:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="150" Width="200">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="30" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Label x:Name="label1" Content="Napszak: " Grid.Row="0"/>

    <GroupBox Grid.Row="1">
      <GroupBox.Header>
        Milyen napszak van?
      </GroupBox.Header>

      <StackPanel>
        <CheckBox x:Name="cb1"
                  Content="Nappal" />
        <CheckBox x:Name="cb2"
                  Content="Éjszaka" />
      </StackPanel>
    </GroupBox>

  </Grid>
</Window>
```

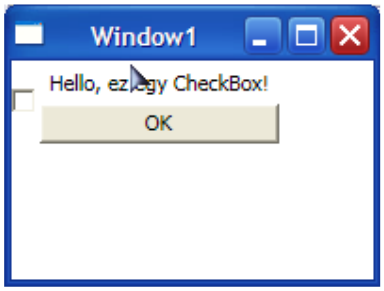
Amiből ez lesz:



Amikor WinForms –szal dolgoztunk, akkor megszokhattuk, hogy egy *GroupBox* –ban elhelyezett *CheckBox* –oknak megadhattunk csoportot, így mindig csak egyet tudtunk megjelelölni. A WPF máshogy működik, a *CheckBox* –ot nem lehet

csoporthoz rendelni, és van még egy nagy különbség. A *CheckBox* a *ContentControl* osztály leszármazottja, így könnyedén átformálhatjuk:

```
<CheckBox>
  <StackPanel>
    <Label>Hello, ez egy CheckBox!</Label>
    <Button>OK</Button>
  </StackPanel>
</CheckBox>
```



A *ContentControl* leszármazottak csakis egy gyermeket tudnak kezelni, így ha többet szeretnénk egy olyan eszközt kell bevetnünk, amely erre képes, ez pedig (pl.) a *StackPanel* lesz (őt már ismerjük).Ezt használtuk a *GroupBox* esetében is, ami feltételezni engedi – és nem téved -, hogy az is egy *ContentControl* leszármazott. Sőt, ez egy még specializáltabb eszköz, egy *HeaderedContentControl*, vagyis lehet valamilyen fejléce.
Kanyarodjunk vissza az eredeti problémához, vagyis, hogy hogyan lehetne megoldani, hogy egyszerre csak egy *CheckBox* –ot jelölhessünk meg? Ez viszont az olvasó feladata lesz, többféle megoldás is létezik (egy kulcsszó: attached event).
A *CheckBox* –nak három állapota lehet: *checked* (amikor bejelöljük), *unchecked* (amikor levesszük róla a jelölést) és *indeterminated* (amikor nincs bejelölve, de nem nyúltunk még hozzá, vagyis ilyen az állapota pl. amikor elindítjuk az alkalmazást). Az állapotoknak megfelelően mindegyikhez létezik eseményvezérlő.
Írjuk át a programot és használjuk ki inkább a *RadioButton* csoportosíthatóságát. Ezt kétféleképpen tehetjük meg, vagy a *GroupName* tulajdonságon keresztül, vagy pedig a csoportosítandó *RadioButton* –okat egy szűlő alá rendeljük (pl. *GroupBox*):

```
<GroupBox Grid.Row="1">
  <GroupBox.Header>
    Milyen napszak van?
  </GroupBox.Header>

  <StackPanel>
    <RadioButton x:Name="rb1" Content="Nappal" />
    <RadioButton x:Name="rb2" Content="Éjszaka" />
  </StackPanel>
</GroupBox>
```

Hasonlóan a *CheckBox* –hoz, a *RadioButton* is háromállású.
Már csak egy dolog van hátra, mégpedig az, hogy a napszaknak megfelelően megváltoztassuk a *Label* feliratát. Ezt egy attached event segítségével fogjuk megtenni, amit a *StackPanel* –hez ragasztunk, lévén ő van a legközelebb:

```
<StackPanel ToggleButton.Checked="StackPanel_Checked">
  <RadioButton x:Name="rb1" Content="Nappal" />
  <RadioButton x:Name="rb2" Content="Éjszaka" />
</StackPanel>
```

Mind a *CheckBox*, mind a *RadioButton* a *ToggleButton* osztályból származik, és a checked/unchecked/indeterminate eseményeket is innen öröklik. Az eseménykezelő ez lesz:

```
private void StackPanel_Checked(object sender, RoutedEventArgs e)
{
   label1.Content = "Napszak: " + ((RadioButton)e.OriginalSource).Content;
}
```

Mivel ez egy routed event, ezért nem konvertálhatjuk közvetlenül az első paramétert, (hiszen az ebben az esetben az az objektum lesz, ami elkapta az eseményt, vagyis most a *StackPanel*), ezért a második paramétertől kérjük el az esemény eredeti forrását.

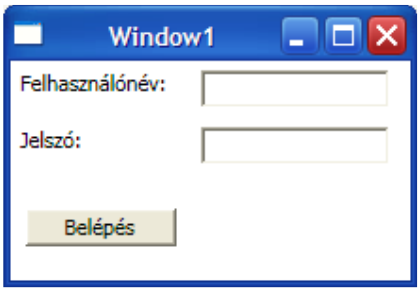
44.2.3 TextBox, PasswordBox és RichTextBox

Ebben a fejezetben a szöveges inputot kezelő vezérlőket vesszük górcső alá. Elsőként készítsünk egy egyszerű beléptető alkalmazást:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="150" Width="220">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="30" />
      <RowDefinition Height="30" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="90" />
      <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Label Content="Felhasználónév:"
           Grid.Row="0"
           Grid.Column="0" />
    <TextBox x:Name="usernamebox"
             Width="100"
             Height="20"
             Grid.Row="0"
             Grid.Column="1" />

    <Label Content="Jelszó:"
           Grid.Row="1"
           Grid.Column="0" />
```

```
    <PasswordBox x:Name="passwordbox"
                 Width="100"
                 Height="20"
                 Grid.Row="1"
                 Grid.Column="1" />

    <Button x:Name="loginbutton"
            Content="Belépés"
            Margin="5 0 0 0"
            Width="80"
            Height="20"
            Grid.Row="2"
            Grid.Column="0"
            Click="loginbutton_Click" />

  </Grid>
</Window>
```

Az eredmény:



Ha elindítjuk az alkalmazást látható lesz, hogy a *PasswordBox* ugyanazt csinálja, mint a WF -beli *TextBox* beállított *PasswordChar* tulajdonsággal. A gomb eseménykezelőjét már könnyen meg tudja írni az olvasó.

Következő vizsgálati alanyunk a *RichTextBox*. Először a hagyományos használatával ismerkedünk meg:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="150" Width="220">
  <Grid>
    <RichTextBox x:Name="richtb1"
                 Width="200"
                 Height="100" />
  </Grid>
</Window>
```

Amikor elindítjuk az alkalmazást, egy teljesen átlagos *RichTextBox* jelenik meg, írhatunk bele, akár formázott tartalmat is, stb…
Van azonban néhány változás is. Az egyik legjelentősebb, hogy mostantól nem tudunk közvetlenül menteni a vezérlőből, ezt nekünk kell intéznünk. Adjunk a *Grid* – hez egy gombot is, amivel ezt majd megtesszük:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="220" Width="220">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="150" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <RichTextBox x:Name="richtb1"
                 Width="200"
                 Height="100"
                 Grid.Row="0"/>

    <Button x:Name="savebutton"
            Width="100"
            Height="20"
            Content="Mentés"
            Grid.Row="1"
            Click="savebutton_Click" />
  </Grid>
</Window>
```

Rendben, most nézzük meg az eseménykezelőt:

```
private void savebutton_Click(object sender, RoutedEventArgs e)
{
   using (FileStream fstream = new FileStream("rtb.rtf", FileMode.Create))
   {
      TextRange range = new TextRange(
         richtb1.Document.ContentStart,
         richtb1.Document.ContentEnd);
      range.Save(fstream, DataFormats.Rtf);
   }
}
```

A *TextRange* osztállyal valamilyen szöveges objektumból tudunk szöveget kiemelni (ez akár egy szimpla strig is lehet). Ennek az osztálynak a segítségével tudunk menteni, méghozzá igen sokféle formátumban. Külső file t- ugyanígy tudunk beolvasni, két különbség van, a stream –nek nem létrehozni, hanem megnyitni kell a forrást, és a *TextRange* objektumnak a *Load* metódusát használjuk.
A *RichTextBox Document* tulajdonsága egy *FlowDocument* objektumot ad vissza, ami tulajdonképpen a tényleges adattárolásért és megjelenítésért felel a *RichTextBox* –on belül. A *ContentStart* és *ContentEnd* tulajdonságok TextPointer objektumokat adnak vissza, amelyet a TextRange tud használni. Egy *RichTextBox* minden esetben tartalmaz pontosan egy *FlowDocument* objektumot, még akkor is, ha ez a forráskódban nem jelenik meg. Ennek segítségével akár előre megadhatunk egy megformált szöveget az XAML kódban:

```
<RichTextBox x:Name="richtb1"
         Width="200"
```

```
        Height="100"
        Grid.Row="0">
  <FlowDocument>
    <Paragraph>
      <Bold>Hello RichTextBox!</Bold>
    </Paragraph>
  </FlowDocument>
</RichTextBox>
```

A *FlowDocument* –en belül *Paragraph* elemeket használtunk, hogy megjelenítsünk egy szöveget. Mind a Pragraph mind a *Bold* elemek tartalmazhatnak egyéb formázásokra utasító parancsot (pl.: karakterkészlet, betűméret).
Maga a *FlowDocument* nem kötődik a *RichTextBox* –hoz, sőt ilyen esetben jelentősen korlátozottak a képességei. A következő oldalon bővebb leírást talál az olvasó erről az osztályról:

http://msdn.microsoft.com/en-us/library/aa970909.aspx

A *RichTextBox* és a *TextBox* egyaránt képes helyesírásellenőrzésre, ezt a *SpellCheck.IsEnabled* attached propeprty segítségével állíthatjuk be. Ugyanígy mindkét vezérlő rendelkezik helyi menüvel, amelyet a jobb egérgombbal való kattintással hozhatunk elő és a kivágás/másolás/beillesztés műveleteket tudjuk vele elvégezni.
A *RichTextBox* alapértelmezetten nem támogatja a görgetősávot, ezt nekünk kell bekapcsolni a *Horizontal/VerticalScrollBarVisibility* tulajdonságokkal, amelyek vízszintes vagy függőleges görgetősávot eredményeznek.

44.2.4 ProgressBar, ScrollBar és Slider

Ezzel a két vezérlővel különösebb gondunk nem lesz, hiszen egyikük sem kínál semmilyen extra lehetőséget, csak azt amit már ismerünk. Bár ez így nem teljesen igaz, mivel a WPF más területeinek lehetőségeivel felvértezve ezek a vezérlők is rendkívül érdekesek lesznek, de ez egy másik fejezet(ek) témája lesz, így most meg kell elégednünk a „hagyományos” módszerekkel.

A *ProgressBar* –ral kezdünk és egyúttal megismerkedünk a WPF szálkezelésével is egy kicsit közelebbről. A feladat a szokásos lesz, gombnyomásra elkezdjük szép lassan növelni a *ProgressBar* értékét. Ez viszont, egy kis trükköt igényel. Azt már tudjuk, hogy a WPF csakis a saját szálában hajlandó módosítani a felhasználói felületet és ez a tudásunk most hasznos lesz. Az nyílvánvaló, hogy a gombnyomásra elindított ciklust le kell lassítanunk, ezt megtehetjük mondjuk a *Thread.Sleep* metódussal. Amikor ezt használjuk, akkor feltűnhet az, hogy a *ProgressBar* meg se mukkan, és a felhasználói felület sem használható, egy idő után pedig a vezérlő megtelik és ismét aktív lesz az alkalmazás. Ez azért van, mert az alkalmazás saját szálát blokkoltuk, annyi ideje azonban nem volt, hogy két ciklislépés közt frissítse a felületet. Az eredmény az lesz, hogy az alkalmazás nem reagál külső beavatkozásra, és csak a futás végén „telik” meg a *ProgressBar*. A megoldás az lesz, hogy egy a főszállal szinkronban működő szálban csináljuk meg a feladatot. Azért kell ennek a szálnak az alkalmazás szálával szinkronban működnie, mert csak így tudjuk azt blokkolni.

A WPF osztályhierarchiájáról szóló fejezetben megtanultuk azt is, hogy a legtöbb WPF beli vezérlő a *DispatcherObject* osztály leszármazottja. Ez az osztály minden egyes alkalmazáshoz biztosít egy *Dispatcher* objektumot, amellyel a főszálat tudjuk vezérelni. Ennek az osztálynak az *Invoke* és *BeginInvoke* metódusaival, tudunk szinkron illetve aszinkron hívásokat végezni, akár egy másik szálból is. Mi most az *Invoke* –ot fogjuk használni. Szükségünk lesz egy delegate –re is, mert mindkét metódus azt vár paramétereként:

```
public delegate void SleepDelegate();

public static void SleepMethod()
{
   System.Threading.Thread.Sleep(100);
}

private SleepDelegate sleepdlgt;
```

Ne felejtsük el hozzáadni a metódust, mondjuk a konstruktorban:

```
sleepdlgt += new SleepDelegate(SleepMethod);
```

Most már nagyon egyszerű dolgunk lesz, a gomb *Click* eseménye így fog kinézni:

```
private void startbutton_Click(object sender, RoutedEventArgs e)
{
   for (int i = 0; i < 100; ++i)
   {
      Dispatcher.Invoke(
         System.Windows.Threading.DispatcherPriority.Render,
         sleepdlgt);
      ++progressbar1.Value;
   }
}
```

Az *Invoke* első paramétere a hívás prioritását jelenti, többféle prioritás létezik és ebben az esetben többet is használhatunk. A Render értékkel hívott delegate ugyanazzal a prioritással rendelkezik, mint az alkalmazás felületét kirajzoló folyamat, ami nekünk most tökéletesen megfelel. Készítünk még egy *Label* –t is, hogy jelezzük, hogy éppen hány százalékánál járunk a folyamatnak, ezt a *ProgressBar ValueChanged* eseményében fogjuk frissíteni:

```
private void progressbar1_ValueChanged(object sender,
RoutedPropertyChangedEventArgs<double> e)
{
   progresslabel.Content = progressbar1.Value.ToString() + "%";
}
```

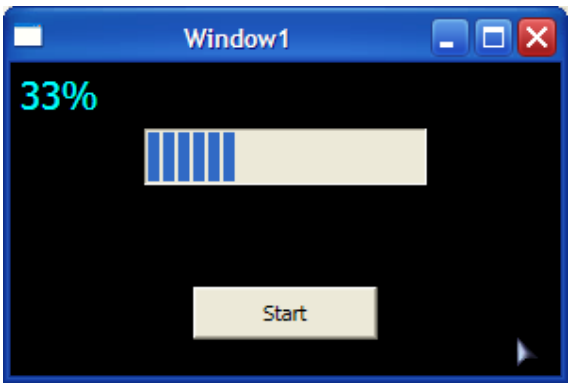
Az eseménykezelő második paramétere is érdekes számunkra. A neve árulkodó lehet, mivel ez a paraméter egy adott tulajdonság változásának körülményeit mutatja. Nézzük meg közelebbről. A *ProgressBar Value* tulajdonsága egy dependency property, a *ValueChanged* pedig egy routed event. Egy későbbi fejezetben fogunk tanulni a WPF adatkötéseiről is, ami pontosabb képet ad a folyamatról. Ez a

paramétertípus egy generikus osztály, a fenti esetben például *double* értékekre specializálódott. A paraméter tulajdonságai közt megtaláljuk a *NewValue* és az *OldValue* tagokat, amelyek értelemszerűen a régi és az új értéket hordozzák magukban.

Az XAML így néz majd ki:

```
<Window x:Class="JegyzetWPF.Window1"
        x:Name="window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="200" Width="300" Background="Black">
   <Grid>
     <Grid.RowDefinitions>
       <RowDefinition Height="100" />
       <RowDefinition Height="*" />
     </Grid.RowDefinitions>

     <ProgressBar x:Name="progressbar1"
                  Minimum="0"
                  Maximum="100"
                  Width="150"
                  Height="30"
                  Grid.Row="0"
                  ValueChanged="progressbar1_ValueChanged"/>

     <Label x:Name="progresslabel"
            FontSize="20"
            Foreground="Cyan"
            Content="0%"
            Grid.Row="0" />

     <Button x:Name="button1"
             Content="Start"
             Width="100"
             Height="30"
             Grid.Row="1"
             Click="button1_Click" />
   </Grid>
</Window>
```

Az eredmény pedig:



Felmerülhet a kérdés, hogy nem –e lehetne ezt egyszerűbben megoldani, például használhatnánk a korábban már bemutatott *Timer* osztályt is. Ha megnézzük a WPF ToolBox –át, akkor ilyet bizony nem fogunk találni (és más komponenst sem). Sőt, igazából a WinForms -os *Timer* -t itt nem is igazán ajánlott használni. Semmi vész, van helyette más: *DispatcherTimer*.
Készítsünk egy új adattagot:

```
private System.Windows.Threading.DispatcherTimer timer;
```

Ez a timer hasonlóan működik, mint a már ismert komponens, a *Tick* eseménye fogja jelezni, hogy ideje cselekedni:

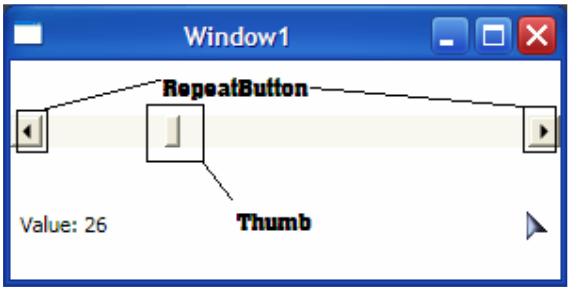
```
timer = new System.Windows.Threading.DispatcherTimer();
timer.Interval = new TimeSpan(10000);
timer.Tick += new EventHandler(timer_Tick);
```

Az eseménykezelő pedig ez lesz:

```
void timer_Tick(object sender, EventArgs e)
{
   if (progressbar1.Value < 100)
   {
     ++progressbar1.Value;
   }
   else timer.Stop();
}
```

Az *Interval* tulajdonság egy *TimeSpan* típusú objektumot vár, ennek (pl.) nanoszekundumban adhatunk értéket.
A *DispatcherTimer* igazából azért jó nekünk, mert a főszálban fut, így módosíthatja a felhasználói felületet minden további nélkül.

Következzen a *ScollBar*. Ez a vezérlő több részből áll, a két végpontján egy-egy *RepeatButton* tartózkodik, illetve az aktuális pozíciót egy *Thumb* objektum jelképezi. Ezek együttesen egy *Track* objektumot alkotnak.
A mostani példa hasonló lesz az előzőhöz, a *ScrollBar* –t mozgatva kiírjuk az aktuális értékét egy *Label* –re. Az értékváltozást kétféleképpen is kezelhetjük, a *Scroll* esemény a változás körülményeit írja le, míg a *ValueChanged* csakis a mutatott érték változását tudja kezelni.



Az XAML:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="150" Width="300">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="75" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <ScrollBar x:Name="scrollbar1"
               Minimum="0"
               Maximum="100"
               Orientation="Horizontal"
               Grid.Row="0"
               ValueChanged="scrollbar1_ValueChanged"/>

    <Label x:Name="valuelabel" Content="Value: 0" Grid.Row="1" />
  </Grid>
</Window>
```

Nézzük az eseménykezelőt:

```
private void scrollbar1_ValueChanged(object sender,
RoutedPropertyChangedEventArgs<double> e)
{
   valuelabel.Content = "Value: " + scrollbar1.Value.ToString();
}
```

Végül foglalkozzunk a *Slider* –rel is. Ezt a vezérlőt gyakorlatilag ugyanúgy kezeljük, mint a *ScrollBar* –t, az XAML definíciója:

```
<Slider x:Name="slider1"
      Width="200"
      Height="30"
      Minimum="0"
      Maximum="100"
      Grid.Row="0"
      ValueChanged="slider1_ValueChanged" />
```

Az állapotváltozásokat szintén a ValueChanged esemény segítségével tudjuk kezelni.

Hamarosan foglalkozunk a WPF adatkötéseivel, amelyek segítségével jóval egyszerűbben oldhatjuk meg az ebben a fejezetben szereplő feladatokat.

44.2.5 ComboBox és ListBox

Ez a két vezérlő is a hagyományos szolgáltatásokat nyújtja számunkra. Kezdjük a ComboBox –xal:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="100" Width="120" Loaded="window1_Loaded">
  <Grid>
    <ComboBox x:Name="combobox1"
              Width="100"
              Height="30" />
  </Grid>
</Window>
```

Az elemeket (először) az ablak betöltésekor, a *Loaded* eseményben adjuk hozzá:

```
private void window1_Loaded(object sender, RoutedEventArgs e)
{
   string[] names = new string[]
   {
      "István", "Béla", "Judit", "Márta"
   };

   combobox1.ItemsSource = names;
}
```

Az *ItemSource* tulajdonságot az *ItemsControl* ősosztálytól örökli a vezérlő és olyan adatszerkezetet vár, amely megvalósítja az *IEnumerable* interfészt, ilyen pl. egy hagyományos tömb.
Elemeket az *Items* tulajdonságon keresztül is megadhatunk, a fentivel tökéletesen megegyező eredményt kapunk, ha így csináljuk:

```
private void window1_Loaded(object sender, RoutedEventArgs e)
{
   combobox1.Items.Add("István");
   combobox1.Items.Add("Béla");
   combobox1.Items.Add("Judit");
   combobox1.Items.Add("Márta");
}
```

Ezeken kívűl XAML –ből is létrehozhatjuk az elemeit:

```
<ComboBox x:Name="combobox1"
        Width="100"
        Height="30" >
  <ComboBoxItem Content="István" />
  <ComboBoxItem Content="Béla" />
  <ComboBoxItem Content="Judit" />
  <ComboBoxItem Content="Márta" />
</ComboBox>
```

A *ComboBoxItem* a *ListBoxItem* –ből származik és leszármazottja a *ContentControl* oszálynak, így összetettebb elemei is lehetnek a vezérlőnek (pl. egy gomb):

```
<ComboBox x:Name="combobox1"
        Width="100"
```

```
        Height="30" >
  <ComboBoxItem>
    <Button>István</Button>
  </ComboBoxItem>
  <ComboBoxItem Content="Béla" />
  <ComboBoxItem Content="Judit" />
  <ComboBoxItem Content="Márta" />
</ComboBox>
```

Ugyanakkor nem vagyunk rákényszerítve a *ComboBoxItem* használatára:

```
<ComboBox x:Name="combobox1"
        IsEditable="True"
        TextSearch.TextPath="Name"
        Width="100"
        Height="30" >
  <Button>István</Button>
  <Button>Béla</Button>
  <Button>Judit</Button>
  <Button>Márta</Button>
</ComboBox>
```

A *ComboBox* alapértelmezetten nem jeleníti meg az általunk beírt szöveget, de keresni tud az alapján. Ha látni is szeretnénk, amit írunk, akkor az *IsEditable* tulajdonságát kell *true* értékre állítanunk.

A *ComboBoxItem*, akár olyan objektumot is tartalmazhat, amely nem szöveges tíusú. Ekkor is rákereshetünk, ha beállítottuk a *TextSearch.Text* attached property –t:

```
<ComboBox x:Name="combobox1"
        IsEditable="True"
        Width="100"
        Height="30" >
  <ComboBoxItem>
    <Button TextSearch.Text="István">István</Button>
  </ComboBoxItem>
  <ComboBoxItem Content="Béla" />
  <ComboBoxItem Content="Judit" />
  <ComboBoxItem Content="Márta" />
</ComboBox>
```

Hasonló célt szolgál a *TextSearch.TextPath* tulajdonság, amivel az elemek egy tulajdonságának értékét tehetjük kereshetővé:

```
<ComboBox x:Name="combobox1"
        IsEditable="True"
        TextSearch.TextPath="Name"
        Width="100"
        Height="30" >
  <Button Name="István" >István</Button>
  <Button Name="Béla">Béla</Button>
  <Button Name="Judit">Judit</Button>
  <Button Name="Márta">Márta</Button>
</ComboBox>
```

A *ComboBox* számunkra fontos eseménye a *SelectionChanged*, amely akkor aktiválódik, amikor megváltozik a kiválasztott elem.

Nagyon kevés dologban tér el a *ListBox* a *ComboBox* –tól, a legszembetűnőbb, hogy ez a vezérlő nem tesz lehetővé keresést, illetve az elemei típusa *ListBoxItem*:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="120" Width="120">
  <Grid>
    <ListBox x:Name="listbox1"
             Width="100"
             Height="80" >
      <ListBoxItem Content="István" />
      <ListBoxItem Content="Béla" />
      <ListBoxItem Content="Judit" />
      <ListBoxItem Content="Márta" />
    </ListBox>
  </Grid>
</Window>
```

A *SelectionMode* tulajdonságával meghatározhatjuk, hogy hány elemet választhat ki a felhasználó. Három mód létezik *Single* (ez az alapértelmezett, egyszerre egy elem), *Multiple* (több elem is kiválasztható, nem igényel külön billenytűleütést) illetve *Extended* (a SHIFT billentyű nyomvatartásával egymást követő elemeket vagy a CTRL billentyű nyomvatartásával önálló elemeket választhatunk ki).

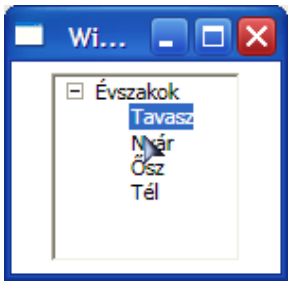
A kiválasztást vagy annak visszavonását a *ListBox SelectionChanged* eseményével tudjuk kezelni. Ennek második paramétere SelectionChangedEventArgs típusú. *AddedItems* illetve *RemovedItems* tulajdonsága az éppen kijelölt, vagy kijelölésből eltávolított elemeket tartalmazza.

44.2.6 TreeView

A *TreeView* vezérlővel hierarchikus sorrendbe állíthatjuk az elemeit. Ő is egy *ItemsControl* leszármazott, elemeinek típusa *TreeViewItem*. Ez utóbbi a *HeaderedItemsControl* osztályból származik, vagyis egy adott elemnek lehetnek gyermek elemei (mivel *ItemsControl* leszármazott) illetve minden elem rendelkezik a *Header* tulajdonsággal, amellyel az egyes elemek feliratát állíthatjuk be:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="130" Width="150" Loaded="window1_Loaded">
  <Grid>
    <TreeView x:Name="treeview1"
              Width="100"
              Height="100">
      <TreeViewItem Header="Évszakok">
```

```
        <TreeViewItem Header="Tavasz" />
        <TreeViewItem Header="Nyár" />
        <TreeViewItem Header="Ősz" />
        <TreeViewItem Header="Tél" />
      </TreeViewItem>
    </TreeView>
  </Grid>
</Window>
```



Mivel a *TreeViewItem* egyben I*temsControl* is, az egyes csomópontok szövegen kívül mást is tartalmazhatnak:

```
<TreeViewItem Header="Évszakok">
  <TreeViewItem Header="Tavasz" >
    <Button>Tavasz - Gomb</Button>
  </TreeViewItem>
  <TreeViewItem Header="Nyár" />
  <TreeViewItem Header="Ősz" />
  <TreeViewItem Header="Tél" />
</TreeViewItem>
```

A vezérlő *ItemSource* tulajdonságán keresztül akár gyűjteményeket is hozzáadhatunk, de csakis akkor, ha nem tartalmaz más elemeket.

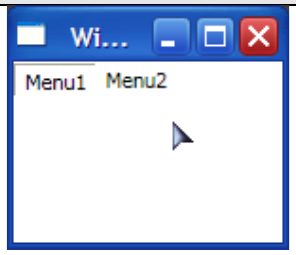
Az *Items* tulajdonság hagyományos gyűjteményként viselkedik, így indexelhetjük, hozzáadhatunk vagy akár törölhetünk is belőle.

44.2.7 Menu

A vezérlővel egy hagyományos menüt tudunk létrehozni. Ez a vezérlő is *ItemsControl* leszámazott, elemeinek típusa *MenuItem*, amelyek a *HeaderedItemsControl* osztályból származnak:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="130" Width="150">
  <Grid>
    <Menu x:Name="menu1">
      <MenuItem Header="Menu1">
        <MenuItem Header="SubMenu1.1" />
      </MenuItem>
      <MenuItem Header="Menu2" />
```

```
    </Menu>
  </Grid>
</Window>
```



Egy menüpont kiválasztását és így egy parancs végrehajtását pl. a Click eseménnyel (amely egyébként a MenuItem osztályhoz tartozik) tudjuk kezelni (van egy sokkal elegánsabb módszer is, de az egy másik fejezet témája lesz):

```
<Menu x:Name="menu1" MenuItem.Click="menu1_Click">
  <MenuItem Header="Menu1">
    <MenuItem Header="SubMenu1.1" />
  </MenuItem>
  <MenuItem Header="Menu2" />
</Menu>
```

Az eseménykezelő:

```
private void menu1_Click(object sender, RoutedEventArgs e)
{
   MessageBox.Show(((MenuItem)e.OriginalSource).Header.ToString());
}
```

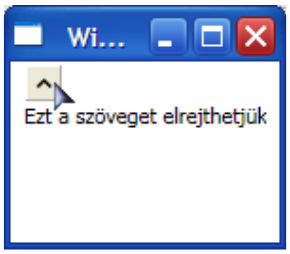
A *MenuItem* attached event -jeinek segítségével több eseményt is tudunk kezelni, mint pl. egy almenü megnyitása.

44.2.8 Expander és TabControl

Az *Expander* a *HeaderedContentContorl* osztályból származik, ezért tartalmazhat más elemeket is. Ezt a tulajdonságát kiegészti azzal, hogy a tartalmazott elemeket képes elrejteni/megjeleníteni:

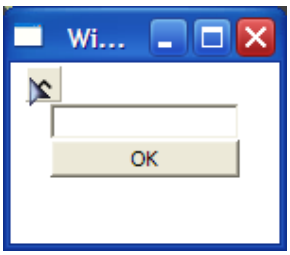
```
<Window x:Class="JegyzetWPF.Window1"
     x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="130" Width="150">
  <Grid>
    <Expander x:Name="expander1"
              Width="130"
              Height="100"
              >
      <Expander.Content>
        <TextBlock>
        Ezt a szöveget elrejthetjük...
        </TextBlock>
      </Expander.Content>
```

```
    </Expander>
  </Grid>
</Window>
```



Az *Expander* (a *ContentControl* osztályból való származása miatt) csak egyetlen gyermekelemet képes kezelni, így ha több vezérlőt akarunk elhelyezni benne szükségünk lehet egy olyan elemre, amely képes őket kezelni (pl.: *Grid*, *StackPanel*, stb…):

```
<Expander x:Name="expander1"
        Width="130"
        Height="100"
        >
  <Expander.Content>
    <StackPanel>
      <TextBox x:Name="textbox1"
               Width="100"/>
      <Button x:Name="button1"
              Width="100"
              Height="20"
              Content="OK" />
    </StackPanel>
  </Expander.Content>
</Expander>
```

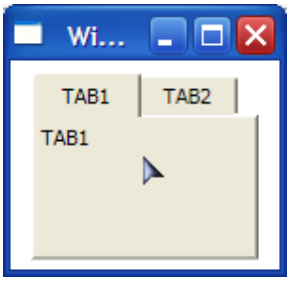


A vezérlő nyitott vagy csukott állapotát az *IsExpanded* tulajdonsággal kérdezhetjük le (illetve állíthatjuk be), ennek alapértelmezett értéke *true*.
A nyitás „irányát” az *ExpandDirection* tulajdonsággal állíthatjuk.

A *TabControl* vezérlő több „oldalt” tartalmaz, amelyek közül mindig csak egy látszik teljes valójában. Az egyes oldalak *TabItem* tulajdonságúak és a *HeaderedContentControl* osztályból származnak.

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="130" Width="150">
  <Grid>
    <TabControl x:Name="tabcontrol1"
                Width="120"
                Height="100"
                >
      <TabItem Header="TAB1">
        <TextBlock Text="TAB1" />
      </TabItem>
      <TabItem Header="TAB2">
        <TextBlock Text="TAB2" />
      </TabItem>
    </TabControl>
  </Grid>
</Window>
```



44.2.9 Image és MediaElement

Az *Image* vezérlő képek megjelenítésére szolgál (gyakorlatilag megfelel a WinForms –os *PictureBox* –nak). A megjelenítendő képet a *Source* tulajdonságán keresztül tudjuk megadni:

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="130" Width="150">
  <Grid>
    <Image x:Name="image1"
           Width="100"
           Height="100"
           Source="image.jpg" />
  </Grid>
</Window>
```

A *MediaElement* lehetővé teszi különböző médiafile –ok lejátszását. Minden olyan típust támogat, amelyet a Windows Media Player 10 is.
Kezelése rendkívűl egyszerű (viszonylag könnyen készíthetünk egy egyszerű lejátszó programot), rendelkezik az alapvető műveletek (lejátszás, megállítás, stb…) elvégzéséhez szükséges metódusokkal.

```
<Window x:Class="JegyzetWPF.Window1"
      x:Name="window1"
```

```
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="130" Width="150">
   <Grid>
     <MediaElement x:Name="mediaelement1" Source="file.wmv" />
   </Grid>
</Window>
```

Nézzük meg a fontosabb metódusokat, tulajdonságokat és eseményeket:

- *Play* – ez a metódus elindítja média lejátszását.
- *Pause* – a lejátszás megállítása, ismételt Play után az aktuális pozíciótól folytatja a lejátszást.
- *Stop* – a lejátszás megállítása és az aktuális pozicíó a file eleje lesz.
- *Volume* – ez a tulajdonság a hangerő beállításáért felelős.
- *MediaOpened* és *MediaEnded* – ezek az események a média megnyitásakor illetve a lejátszás végén lépnek akcióba.

45. Erőforrások

Általában amikor valamilyen erőforrásról beszélünk, akkor egy az alkalmazásba „belefordított” bináris adatra (pl. egy kép, vagy hang) gondolunk, ezt *binary*- vagy *assembly resource* –nak nevezzük. A WPF bevezet egy másikfajta erőforrást, amelynek neve: *object resource*. A név alapján könnyű kitalálni, hogy ebben az esetben .NET objektumokról beszélünk.
Ebben a fejezetben mindkét típust megvizsgáljuk.

45.1 Assembly resource

Ezzel a típussal már találkoztunk, még ha nem is tudtunk róla. Az XAML is ilyen erőforrás lesz a fordítás során, amikor BAML –lé alakul.
Egy projecthez nagyon könnyen tudunk erőforrást hozzáadni, kattintsunk jobb egérgombbal a solution- ön, majd válasszuk az Add menü Existing Item lehetőségét. Ezután a kiválasztott elem megjelenik a project file –jai közt is. Jobb gombbal kattintsunk rajta és válasszuk a Properties menüpontot. Ezután a Build Action –t állítsuk Resource –ra. Készen vagyunk, most lássuk, hogyan használhatjuk fel őket. Ennek két módja van, először az egyszerűbbikkel foglalkozunk: tegyük fel, hogy egy kép file –t raktároztunk el, a neve legyen picture.jpg. Helyezzünk el az alkalmazásban egy *Image* vezérlőt is:

```
<Image x:Name="image1" Source="picture.jpg"  Width="100" Height="100" />
```

Látható, hogy egyszerűen csak a file nevével hivatkoztunk képre (persze elhelyezhetnük erőforrást mappában is, ekkor a teljes elérési utat oda kell írnunk). Ha ugyanezt forráskódból akarjuk elérni, akkor a következőt írjuk:

```
image1.Source = new BitmapImage(new Uri("picture.jpg", UriKind.Relative));
```

Rugalmasabban tudjuk kezelni az erőforrásokat egy *StreamResourceInfo* objektummal (ez a *System.Windows.Resources* névtérben van):

```
StreamResourceInfo sri = Application.GetResourceStream(
    new Uri("picture.jpg", UriKind.Relative));
```

Ezután két dolgot tudunk kinyerni ebből az objektumból, a *ContentType* tulajdonsággal ez erőforrás típusát (ez most image/jpg) a *Stream* tualjdonsággal pedig az erőforrásra mutató stream –et kapunk (ennek típusa *UnmanagedMemoryStream*).

Az elérési út nem mindig ilyen egyértelmű, mi van akkor, ha az erőforrás egy másik assembly –ben van? A következő példához tegyük fel, hogy ennek a másik assembly –nek a neve ImageAssembly:

```
image1.Source = new BitmapImage(
new Uri("ImageAssembly;component/picture.jpg", UriKind.Relative));
```

45.2 Object resource

Az object resource koncepciója új számunkra, de a WPF számos lehetősége épít rá (pl.: stílusok). Emellett ezzel a megoldással könnyen használhatunk egy adott objetumot több helyen (pl. *TextBlock* objektumokhoz definiálunk egy stílust, ekkor ezt bárhol felhasználhatjuk később).
Erőforrást bármely *FrameworkElement* vagy *FrameworkContentElement* leszármazott tartalmazhat. Minden ilyen osztály rendelkezik a *Resources* nevű tulajdonsággal amely egy *ResourceDictionary* típusú listát ad vissza, amely megvalósítja (többek közt) az *IDictionary* interfészt, így hasonlóan működik, mint a már ismert *Dictionary* osztály, ebből következik az egyik legfontosabb jellemzője is: vagyis minden tartalmazott erőforrásnak egyedi azonosítójának kell lennie.
A gyakorlatban a legtöbb esetben a top-level-element fogja az alkalmazás (pontosabban az aktuális form) erőforrásait tartalmazni.

Egy erőforrásra kétféleképpen statikusan és dinamikusan hivatkozhatunk, elöbbi esetben az alkalmazás indításakor értékelődik ki a kifejezés, vagyis a WPF kikeresi az összes rendelkezésre álló erőforráslistából a megfelelő tagot. A dinamikus esetben ez akkorra halasztódik, amikor az erőforrás ténylegesen használatra kerül, és ezt minden egyes alkalommal megteszi, vagyis reagál arra, ha az adott erőforrás tulajdonságai megváltoztak.
Nézzünk egy példát:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="300">
  <Grid>
    <Grid.Resources>
      <LinearGradientBrush x:Key="buttoncolor">
        <GradientStop Color="Black" Offset="0.0" />
        <GradientStop Color="Red" Offset="0.7" />
      </LinearGradientBrush>
    </Grid.Resources>

    <Button Width="100" Height="30" Background="{StaticResource buttoncolor}"
VerticalAlignment="Top" />

    <Button Width="100" Height="30" Background="{DynamicResource buttoncolor}"
VerticalAlignment="Bottom" />
  </Grid>
</Window>
```

A két gomb ugyanazt az erőforrást használja, de az első esetben statikusan (*StaticResource*), míg a másodiknál dinamikusan (*DynamicResource*) használtuk fel.

A Visual Studio az erőforrások kezeléséhez nem nyújt IntelliSense támogatást, így azt nekünk kell beírnunk.

Erőforrást természtesen kódból is hozzáköthetünk egy vezérlőhöz:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
```

```
    resbutton.Background = (Brush)this.FindResource("buttoncolor");
}
```

Persze a megfelelő objektumon kell meghívnunk a *FindResource* metódust, a fenti esetben a *this* a legfelső szintű elemre a Window –ra mutat, tehát az előtte lévő példával nem fog működni, helyette írhatjuk ezt (vagy adhatunk nevet a *Grid* –nek):

```
<Window x:Class="JegyzetWPF.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">

  <Window.Resources>
    <LinearGradientBrush x:Key="buttoncolor">
      <GradientStop Color="Black" Offset="0.0" />
      <GradientStop Color="Red" Offset="0.7" />
    </LinearGradientBrush>
  </Window.Resources>

  <Grid>
    <Button Width="100" Height="30" Background="{StaticResource buttoncolor}"
VerticalAlignment="Top" />

    <Button x:Name="resbutton" Width="100" Height="30"
VerticalAlignment="Bottom" />
  </Grid>
</Window>
```

A *FindResource* helyett használhatjuk a *Resource* tulajdonság indexelőjét is:

```
resbutton.Background = (Brush)this.Resources["buttoncolor"];
```

A *FindResource* biztonságosabb változata a *TryFindResource*, amely null értéket ad vissza, ha az erőforrás nem található.

Az App.xaml file –ban elhelyezhetünk „globális” erőforrásokat, amelyek az alkalmazás bármely pontjáról elérhetőek:

```
<Application x:Class="JegyzetWPF.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml">
  <Application.Resources>
    <LinearGradientBrush x:Key="buttoncolor">
      <GradientStop Color="Black" Offset="0.0" />
      <GradientStop Color="Red" Offset="0.7" />
    </LinearGradientBrush>
  </Application.Resources>
</Application>
```

Még jobban kiterjeszthetjük ezt, ha ResourceDictionary –t használunk, ezt egy hagyományos XAML file –ba helyezhetjük (projecten jobb klikk, Add/New Item és a WPF fül alatt válasszuk a Resource Dictionary –t):

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
   <LinearGradientBrush x:Key="buttoncolor">
      <GradientStop Color="Black" Offset="0.0" />
      <GradientStop Color="Red" Offset="0.7" />
   </LinearGradientBrush>
</ResourceDictionary>
```

Ezt a file -t nevezzük el mondjuk AppRes.xaml –nek. Ahhoz, hogy a tartalmát használni tudjuk be kell olvasztanunk az alkalmazásunkba, ezt az App.xaml –ben kell megtennünk:

```
<Application x:Class="JegyzetWPF.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml">
   <Application.Resources>
      <ResourceDictionary>
         <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="AppRes.xaml" />
         </ResourceDictionary.MergedDictionaries>
      </ResourceDictionary>
   </Application.Resources>
</Application>
```

Természetesen bármennyi forrást felvehetünk, illetve ha magunk akarunk itt erőforrásokat felvenni, azt is megtehetjük, csak írjuk a *MergedDictionaries* alá.

46. Stílusok

Az előző fejezetben bemutatott erőforrások egyik leggyakrabban használt területe a stílusok. A koncepció nagyon hasonló a CSS –éhez, vagyis megadott tulajdonságokhoz rendelünk értékeket. Van azonban egy dolog, amely még erőteljesebbé teszi a WPF stílusait, mégpedig az, hogy a segítségükkel bármely dependency property –t beállíthatjuk – de csakis azokat.
A stílusok témaköréhez tartoznak a triggerek és a sablonok is, velük is megismerkedünk ebben a fejezetben.

Kezdjük rögtön egy példával:

```
<Window x:Class="JegyzetTestWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="100" Width="300">

  <Window.Resources>
    <Style x:Key="labelfontstyle">
      <Setter Property="Control.FontFamily" Value="Arial Black" />
      <Setter Property="Control.FontSize" Value="42" />
    </Style>
  </Window.Resources>
  <Grid>
    <Label Style="{StaticResource labelfontstyle}" Content="Hello Style!" />
  </Grid>
</Window>
```

Az eredmény:

A kód önmagáért beszél.
Stílust nem csak XAML –ből rendelhetünk egy elemhez:

```
label1.Style = (Style)this.FindResource("labelfontstyle");
```

A *Style* tulajdonságot minden vezérlő a *FrameworkElement* osztálytól örökli, típusa pedig - mint az a fenti példában is létszik – *Style* lesz. Látható, hogy a stílusokat az erőforrásokhoz hasonlóan kezelhetjük (lévén maga is egy erőforrás, csak van saját neve). Ebből a példából az is kiderül, hogy egy elemhez csak egyetlen stílust rendelhetünk hozzá, tehát több stílus összeollózása nem fog működni – legalábbis így nem.
Azt sem árt tudni, hogy egy stílus bármely értékét felülbírálhatjuk, azzal, hogy explicit módon megadjuk a vezérlő adott tulajdonságának az értékét. Vegyük például a fent létrehozott stílust és rendeljük hozzá egy *Label* vezérlőhöz:

```
<Label Style="{StaticResource labelfontstyle}" FontSize="10" Content="Hello Style!" />
```

Ebben az esetben csakis a betűtípust fogja a stílus meghatározni, mivel beállítottuk a betűméretet, amely felülbírálja a stílust.

Gyakran előfordul, hogy egy tulajdonság értékét nem tudjuk egy egyszerű értékkel megadni, ekkor a következőt tehetjük:

```
<Style x:Key="backgroundstyle">
  <Setter Property="Control.Background">
    <Setter.Value>
      <RadialGradientBrush>
        <GradientStop Color="Red" Offset="0.0" />
        <GradientStop Color="Black" Offset="0.75" />
      </RadialGradientBrush>
    </Setter.Value>
  </Setter>
</Style>
```

Nem csak tulajdonságokat, de eseményeket is hozzárendelhetünk egy vezérlőhöz az EventSetter objektumokkal:

```
<Style x:Key="eventstyle">
  <EventSetter Event="ButtonBase.Click" Handler="Button_Click" />
</Style>
```

A *BasedOn* tulajdonság beállításával megadhatunk egy másik stílust, amelynek értékeit az új stílus is használja, így lehetővé téve több stílus összeolvasztását:

```
<Style x:Key="style1">
  <Setter Property="Control.FontFamily" Value="Arial Black" />
  <Setter Property="Control.FontSize" Value="42" />
</Style>

<Style x:Key="style2" BasedOn="{StaticResource style1}">
  <Setter Property="Control.Foreground" Value="Red" />
</Style>
```

Ezt alkalmazva az eredmény:

```
<Label Style="{StaticResource style2}" Content="Hello Style!" />
```

Eddig a pontig az általános *Control* tulajdonságokra hivatkoztunk, a *TargetType* tulajdonság segítségével viszont egy adott vezérlőtípusra érévnyesíthetjük a stílust:

```
<Style x:Key="typestyle" TargetType="Label" >
  <Setter Property="FontFamily" Value="Arial Black" />
  <Setter Property="FontSize" Value="42" />
</Style>
```

Egy stílust hozzárendelhetünk egy adott típushoz, anélkül, hogy azt az elem definíciójánál külön jeleznénk. Ebben az esetben a stílust definiáló elem összes megfelelő típusú gyermekelemére vonatkozni fognak a beállítások:

```
<Style x:Key="{x:Type Label}" TargetType="Label" >
  <Setter Property="FontFamily" Value="Arial Black" />
  <Setter Property="FontSize" Value="42" />
</Style>

<Label Content="Hello Style!" />
```

A fenti példában a *Label* vezérlő – bár ezt explicite nem jeleztük rendelkezni fog a stílus beállításaival (ugyanígy az összes többi *Label* objektum is).
Amennyiben megadunk a vezérlőnek egy másik stílust, az felülbírálja a „globálisat”. Azt is megtehetjük, hogy eltávoltunk minden stílust egy vezérlőről:

```
<Label Style="{x:Null}" Content="Hello Style!" />
```

46.1 Triggerek

Egy stílus rendelkezhet triggerekkel, amelyek segítségével beállíthatjuk, hogy hogyan reagáljon egy vezérlő egy esemény bekövetkeztével, vagy egy tulajdonság megváltozásakor. Ahogyan azt a WPF –től megszokhattuk, a hozzárendelt tulajdonságok csakis dependency property –k lehetnek. Trigger –t köthetünk közvetlenül egy vezérlőhöz is a *FrameworkElement.Triggers* tulajdonságon keresztül, de ekkor csak és kizárólag esemény trigger –t használhatunk.
Ötféle trigger létezik, ezek közül most hárommal fogunk megismerkedni, a többiek az adatkötések témaköréhez tartoznak:

- *Trigger*: a legegyszerűbb, egy dependency property –t figyel és aszerint változtatja meg a stílust.
- *MultiTrigger*: hasonló az előzőhöz, de több feltételt is megadhatunk, amelyek mindegyikének teljesülnie kell.
- *EventTrigger*: a fent már említett esemény trigger, ő a megadott esemény hatására lép akcióba.

Ebből a felsorolásból kimaradt a *DataTrigger* és a *MultiDataTrigger*, velük egy későbbi fejezetben lehet majd találkozni.

46.1.1 Trigger

Készítsünk egy egyszerű trigger –t, ami az IsMouseOver tulajdonságot figyeli és akkor aktiválódik, amikor az értéke igaz lesz:

```
<Style x:Key="triggerstyle">
  <Style.Triggers>
```

```
    <Trigger Property="Control.IsMouseOver" Value="true">
      <Setter Property="Control.FontSize" Value="42" />
    </Trigger>
  </Style.Triggers>
</Style>
```

Rendeljük hozzá a stílust egy Label –hez:

```
<Label Style="{StaticResource triggerstyle}" Content="Hello Style!" />
```

Ezután ha a *Label* fölé visszük az egeret, akkor a betűméret megnő. Nem marad viszont úgy, a *Trigger* által definiált viselkedés csakis addig tart, amíg az aktiválását kiváltó tulajdonság a megfelelő értéken áll.
Természetesen egy stílus több trigger –t és egy trigger több *Setter* –t is tartalmazhat.

46.1.2 MultiTrigger

A *MultiTrigger* segítségével összetettebb feltételeket is felállíthatunk. Ezek között a feltételek közt ÉS kapcsolat van, azaz minden feltételnek teljesülnie kell ahhoz, hogy a trigger aktiválódjon.

```
<Style x:Key="multitriggerstyle">
  <Style.Triggers>
    <MultiTrigger>
      <MultiTrigger.Conditions>
        <Condition Property="Control.IsEnabled" Value="true" />
        <Condition Property="Control.IsFocused" Value="true" />
      </MultiTrigger.Conditions>
      <MultiTrigger.Setters>
        <Setter Property="Control.Foreground" Value="Red" />
      </MultiTrigger.Setters>
    </MultiTrigger>
  </Style.Triggers>
</Style>
```

Ebben a példában az aktív és fókuszban lévő vezérlőket célozzuk meg, mondjuk legyen ez egy *TextBox*:

```
<TextBox Style="{StaticResource multitriggerstyle}" Width="100" Height="20" />
```

Amikor gépelni kezdünk a *TextBox* –ba, akkor a szöveg pirosra vált, de ha lekerül róla a fókusz visszatér a rendszerértékhez.

46.1.3 EventTrigger

Az *EventTrigger* eseményekre reagál, de eltérően társaitól ő egy animációt fog elindítani (ez a témakör egy következő fejezet tárgya lesz, most csak egy egyszerű példa következik):

```
<Style TargetType="{x:Type Button}">
  <Style.Triggers>
    <EventTrigger RoutedEvent="MouseEnter">
```

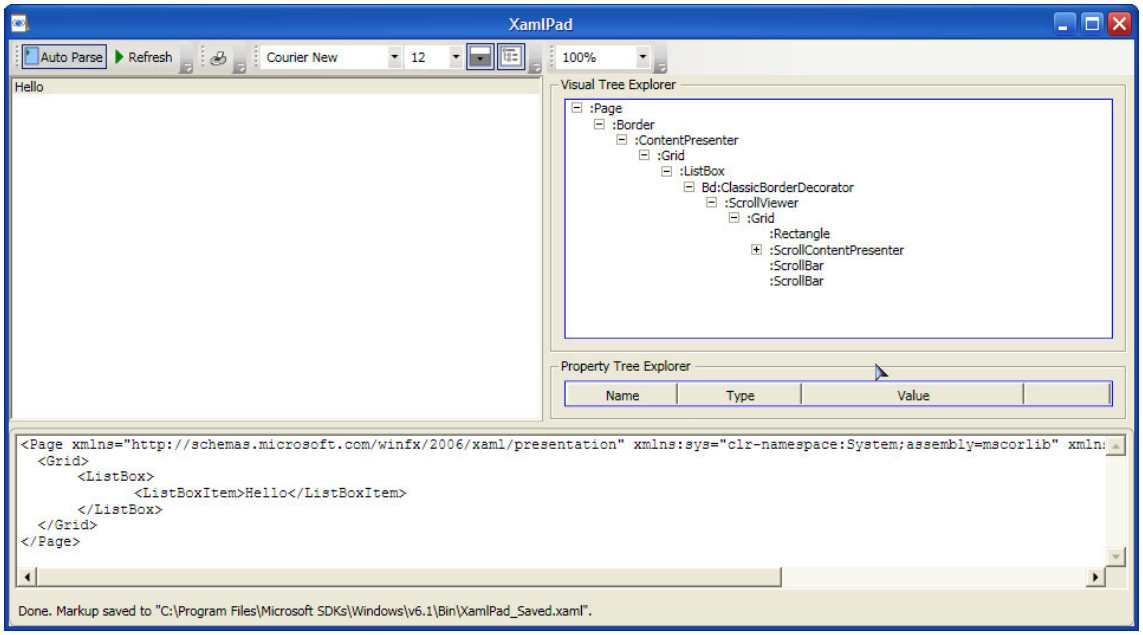
```
        <EventTrigger.Actions>
          <BeginStoryboard>
            <Storyboard>
              <ColorAnimation

Storyboard.TargetProperty="(Button.Background).(SolidColorBrush.Color)"
                        From="White" To="Black" Duration="0:0:5" />
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
    </Style.Triggers>
</Style>
```

A példában egy egyszerű animációt definiáltunk, amely akkor kezdődik, amikor az egeret egy gomb fölé visszük és egy fehérből feketébe tartó színátmenetet eredményez.

47. Sablonok

Ez a fejezet a vezérlők sablonjaival foglalkozik, a WPF még két másik sablontípussal (DataTemplate és HierarchicalDataTemplate) rendelkezik, róluk egy másik fejezetben esik szó majd.

Sablonok segítségével gyakorlatilag bármilyen változást eszközölhetünk egy vezérlő kinézetén, úgy, hogy az semmit sem veszítsen funkcionalitásából (sőt, akár egy teljesen eltérő elemet is készíthetünk). Amikor Windows Forms –szal dolgoztunk, egy ilyen akció kivitelezéséhez User Control –t kellet gyártanunk és rengeteg munkával átírnunk a renderelésért felelős metódusokat.
A WPF – felépítéséből adódóan – változtat ezen.
Egy korábbi fejezetben már megismerkedtünk a logikai- és vizuális fák intézményével, most ezt a tudásunkat fogjuk egy kicsit bővíteni. Segítségünkre lesz az XamlPad nevű program, amely a Windows SDK –val érkezik (általában beépül a Start menübe). Indítsuk el és készítsünk egy egyszerű *ListBox* vezérlőt egy *ListBoxItem* –mel. Fölül találunk egy ikont, amely megmutatja a vizuális fát. Kapcsoljuk be és nézzük meg az eredményt:



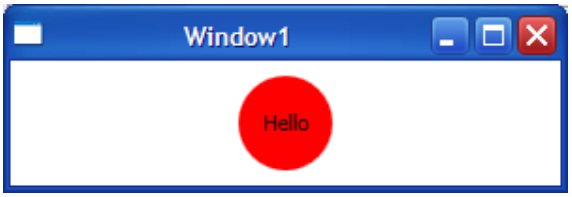
Amit látunk az nem más, mint a *ListBox* vezérlő alapértelmezett sablonja. Tartalmaz pl. egy *Grid* –et egy *Rectangle* –t, stb…
Most már könnyű kitalálni, hogy miként tudunk új sablont készíteni: ezt a fát kell átdefiniálni. Egy példán keresztül világosabb lesz a dolog egy kissé, ezért készítsünk egy a megszokottól eltérő formájú gombot!
Sablont az erőforrásoknál megszokott módon a *Resources* „tulajdonságon” belül a *ControlTemplate* osztállyal tudunk definiálni:

```
<ControlTemplate x:Key="circlebuttontemplate" TargetType="{x:Type Button}">
  <Grid Width="50" Height="50">
    <Ellipse Fill="Red">
    </Ellipse>

    <ContentPresenter VerticalAlignment="Center"
```

```
                HorizontalAlignment="Center"
                Content="{TemplateBinding Content}" />
  </Grid>
</ControlTemplate>
```

```
<Button Content="Hello" Template="{StaticResource circlebuttontemplate}" />
```

Az eredmény:



A sablon definiciójának elején rögtön megmondtuk, hogy ez a sablon gombok számára készült. Ez egyrészt konvenció másrészt elengedhetetlen a helyes működéshez, mivel a *ContentControl* leszármazottak – amilyen a *Button* is – enélkül nem jelenítenék meg a tartalmukat. Ez utóbbihoz szükséges a *ContentPresenter* osztály is amely a sablonunk alján helyezkedik el, ő fogja tárolni a „tartalmazott” adatokat. A *TemplateBinding* segítségével ki tudjuk nyerni a „sablonozott” vezérlő tulajdonságainak értékét jelen esetben ez a *Content* lesz. A *TemplateBinding* működéséről az adatkötésekről szóló fejezet többet is mesél majd.
A gombunk egyelőre nem csinál túl sokat, legalábbis vizuálisan nem. Egyébként pont ugyanúgy működik mint egy mezei gomb, rendelhetünk hozzá pl. *Click* eseménykezelőt, amely rendesen le is fut majd, de nem fog látszani, hogy rákattintottunk. Mi sem könnyebb, mint megoldani ezt a problémát, Trigger –eket fogunk használni:

```
<ControlTemplate x:Key="circlebuttontemplate" TargetType="{x:Type Button}">
  <Grid Name="main" Width="50" Height="50">
    <Ellipse Name="back" Fill="Red">
    </Ellipse>

    <ContentPresenter VerticalAlignment="Center"
                HorizontalAlignment="Center"
                Content="{TemplateBinding Content}" />
  </Grid>

  <ControlTemplate.Triggers>
    <Trigger Property="IsMouseOver" Value="true">
      <Setter TargetName="back" Property="Fill" Value="Black" />
      <Setter TargetName="main" Property="TextElement.Foreground"
       Value="White" />
    </Trigger>
  </ControlTemplate.Triggers>
</ControlTemplate>
```

Mostantól ha az egér a gombunk fölké kerül, akkor azt látni is fogjuk. Ugyanígy létrehozhatjuk a többi esemény stílusát is.

48. Commanding

Képzeljük el a következő helyzetet: egy szövegszerkesztő programot fejlesztünk és éppen a kivágás/másolás/beillesztés funkciókat próbáljuk megvalósítani. A hasonló programok esetében megszokhattuk, hogy ezeket a funciókat többféleképpen is elérhetjük: menüből, helyi menüből, billentyűkombinációval, stb… Az nyílvánvaló, hogy bármelyiket válasszuk is ki az ugyanazt a metódust fogja meghívni. Ezen a ponton már belefutunk abba a problémába, hogy írnunk kell egy rakás eseménykezelőt, amelyek mind ugyanazt csinálják. És ez még nem minden, hiszen a felhasználói felületet is módosítanunk kell, hiszen nem mindegyik funkció érhető el mindig (pl. másolni csak akkor lehet ha van kijelölt szöveg, beilleszteni csak akkor, ha van valami a vágólapon, stb…), ezért ennek megfelelően változtatni kell a felület elemeinek állapotát is, ami újabb eseménykezelők készítését jelenti.
Ennyi probléma épp elég ahhoz, hogy egy amúgy egyszerű alkalmazás elkészítése rémálommá váljon.
Szerencsére a WPF erre is ad választ a *command model* bevezetésével. Mit is takar ez? Definiálhatunk ún. parancs objektumokat, amelyeket vezérlőkhöz köthetünk, amelyek állapota a parancs elérhetőségétől függően változik.
Ennek a megoldásnak is vannak hiányosságai, pl. nincs beépített lehetőség a parancsok „követésére” és ezért vissza sem tudjuk vonni azokat.

A *command model* négy főszereplővel bír, ezek a következőek:

- Maga a parancs (command), ez reprezentálja a valódi kódot a parancs mögött és számontartja annak aktuális állapotát (aktív/passzív).
- A kötés (command binding) vagyis egy parancsot hozzáköthetünk egy vagy több vezérlőhöz.
- A forrás (command source), a vezérlő amely kiváltja az adott parancs végrehajtását.
- A cél (command target), amin a parancs végrehajtja a feladatát (pl. a példánkban ez lehet egy *RichTextBox*, amibe beillesztünk).

Mielőtt rátérünk a gyakorlatiasabb részre fontos megismernünk egy kicsit mélyebben a commanding működését. Minden egyes parancs objektum megvalósítja az *ICommand* interfészt amely így néz ki:

```
public interface ICommand
{
   void Execute(object parameter);
   bool CanExecute(object parameter);

   event EventHandler CanExecuteChanged;
}
```

Az első metódus a parancs végrehajtásáért felel, a második a parancs végrehajthatóságát adaj vissza, míg a harmadik tag egy esemény, amely a parancs állapotának megváltozásakor aktivizálódik.
Ez persze csak egy leegyszerűsített magyarázat, nézzük meg, hogy mi történik valójában: az *Execute* metódus meghívásakor egy esemény váltódik ki, amely jelzi az alkalmazás egy másik részének, hogy ideje cselekedni (ez lesz a valódi kód). A

CanExecuteChanged pedig jelzi az adott parancshoz kötött összes vezérlőnek, hogy meg kell hívniuk a *CanExecute* metódust és ennek megfelelően módosítani magukat (pl. a másolás menüpont ne legyen aktív).

Ez a működés a Commanding egy nagy hátránya is egyben, hiszen ha a felületen történik valami akkor az összes parancsot végig kell ellenőrizni. Ez persze finomítható, de ez már túlmutat a jegyzeten.

Az előbb azt mondtuk, hogy minden parancs megvalósítja az *ICommand* interfészt, de ez nem teljesen igaz. Valójában a WPF egyetlen olyan osztállyal rendelkezik amely direkt módon megvalósítja ezt az interfészt (és ide értjük a saját magunk által készített parancsokat is) ez pedig a *RoutedCommand*.
Ez az osztály csal egy kicsit ugyanis az *ICommand* metódusait privátként implementálja és saját metódusokat készit, amelyek eltakarják azokat. Miért van ez így? Az osztály nevének előtagja megadja a választ: azért, hogy támogatni tudja a routed event –eket. Az *ICommand* egy „platformfüggetlen" megoldás, vagyis nem tud arról, hogy őt most WPF alatt szeretnénk használni, a parancsoknak és az őket kiváltó vezérlőkek viszont tudniuk kell kommunikálni egymással. Ezenkívűl a *RoutedCommand* valósítja meg a tényleges parancskezelést, vele részletesebben meg fogunk ismerkedni, amikor saját parancsot készítünk.
A legtöbb parancs mégsem ebből az osztályból származik, hanem az ő leszármazottjából a *RoutedUICommand* –ból. Ez az osztály egy extra *Text* tulajdonsággal rendelkezik, amely a parancs vizuális reprezentációja lesz, tehát ez fog megjelenni pl. egy menüben. Ez azért jó, mert ha többnyelvű megvalósítást szeretnénk, akkor csak egy helyen kell a lokalizációról gondoskodnunk.

A WPF jónéhány előre definiált paranccsal rendelkezik szám szerint 144 –gyel. Ezek a parancsok öt csoportba oszthatóak:

- *ApplicationCommands*: a legáltalánosabb parancsok tartoznak ide, pl. a vágólaphoz kapcsolódóak (másolás, kivágás, beillesztés, stb.) is.
- *EditingCommands*: a dokumentumok kezelésével kapcsolatos parancsok, pl.: kijelölés, mozgás a szövegben, szövegformázás, stb…
- *MediaCommands*: multimédiás tartalmakat irányítanak ezek a parancsok (lejátszás, szünet, stb.).
- *NavigationCommands*: a navigációért felelős parancsok gyűjteménye, velük egy külön fejezetben fogunk foglalkozni.
- *ComponentCommands*: ezek a parancsok nagyon hasonlóak az *EditingComands* csoporthoz, azzal a különbséggel, hogy ők a felhasználói felületen fejtik ki a hatásukat.

Eljött az ideje, hogy a gyakorlatban is felhasználjuk tudásunkat. Az *EditingCommands* csoportot fogjuk ehhez felhasználni. Tehát, parancsunk már van, most kell egy forrás, ami kiváltja ezt a parancsot. Ez bármelyik vezérlő lehet, amely megvalósítja az *ICommandSource* interfészt. Ilyen pl. az összes *ButtonBase* leszármazott, egy *ListBox* elemei vagy egy menüben a menüpontok.
Ez az interfész három tulajdonságot definiál:

- *Command*: ez az alkalmazandó parancsra mutat.
- *CommandParameter*: az esetleges paraméter, amit a paranccsal küldünk.

- *CommandTarget*: az az elem, amelyen a parancs kifejti hatását.

Nézzük a következő XAML –t:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="400" Loaded="Window_Loaded" >
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="200" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <RichTextBox Name="rtbox1"
                 Width="300"
                 Height="200"
                 Grid.Row="0" />

    <Button Name="boldbutton"
            Width="100"
            Height="30"
            Content="Félkövér"
            Grid.Row="1"
            Command="EditingCommands.ToggleBold"
            CommandTarget="{Binding ElementName=rtbox1}"/>
  </Grid>
</Window>
```

Amikor rákattintunk a gombra akkor a *RichTextBox* –ban a betűtípus félkövérre módosul. A gomb Command tulajdonságánál névterestől adtuk meg a parancs nevét, ez egyrészt az átláthatóságot szolgálja, másrészt elkerüljük a névütközéseket (ugyanakkor nem kötelező így használni).

A Visual Studio 2008 XAML szerkesztője sajnos nem ad IntelliSense kiegészítést ehhez a feladathoz.

A parancsok végrehajtásához rendelhetünk eseménykezelőket, ami hasznos lehet pl. a szövegszerkesztő programunkban, amikor meghatározzuk, hogy melyik parancs legyen elérhető. Ehhez ún. *CommandBinding* –et fogunk használni, amelyet a következőképpen definiálhatunk:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="150" Width="150" Loaded="Window_Loaded" >

  <Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Close"
                    Executed="CommandBinding_Executed" />
  </Window.CommandBindings>

  <Grid>
    <Button Command="ApplicationCommands.Close"
```

```
            Content="Close"
            Width="100"
            Height="30" />
  </Grid>
</Window>
```

Négy eseményt kezelhetünk:

- *Executed*: a parancs végrehajtásakor aktivizálódik, a stratégiája bubbling, vagyis a forrástól indul.
- *PreviewExecuted*: ugyanaz mint az előző, de a stratégiája tunneling.
- *CanExecute*: akkor aktivizálódik, amikor a hozzárendelt vezérlő meghívja a *CanExecute* metódust, a stratégiája bubbling.
- *PreviewCanExecute*: ugyanaz mint az előző de a stratégiája tunneling.

Az eseménykezelők hozzárendelését csakis egyzer kell megtennünk, ha az adott parancsot hozzárendeltük legalább egy vezérlőhöz azután automatikusan meghívódnak ezek is.

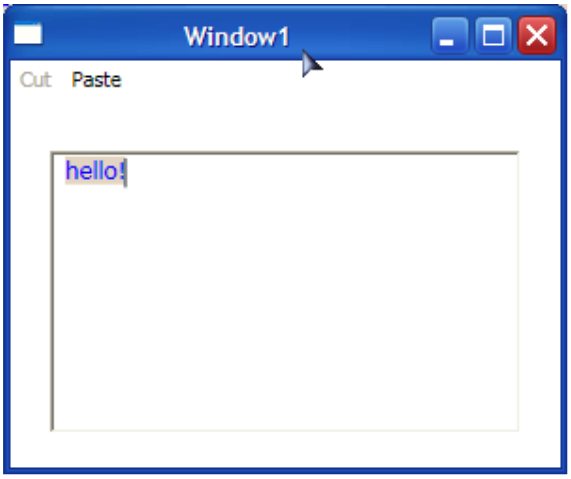
Már beszéltünk a *RoutedUICommand Text* tulajdonságáról. Ezt felhasználhatjuk a vezérlőinken, így azzal sem kell törödnünk, hogy „ráírjuk” pl. egy gombra, hogy mit csinál:

```
<Button Command="ApplicationCommands.Close"
     Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"
     Width="100"
     Height="30" />
```

Elsőre egy kicsit ijesztő lehet ez a megoldás és most nem is foglalkozunk vele majd csak az adatkötésekről szóló fejezetben.
Nézzünk meg egy másik példát:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="250" Width="300" Loaded="Window_Loaded" >

  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="30" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Menu Grid.Row="0">
      <MenuItem Command="ApplicationCommands.Cut" />
      <MenuItem Command="ApplicationCommands.Paste" />
    </Menu>

    <RichTextBox Width="250"
                 Height="150"
                 Grid.Row="1"/>
  </Grid>
</Window>
```

Két dolog is feltűnhet, az egyik, hogy a menüpontoknál nem adtunk meg szöveget, a másik, hogy látszólag nem kötöttük rá a parancsot a *RichTextBox* –ra. Az első esetben azt használtuk ki, hogy bizonyos vezérlők automatikusan tudják használni a *RoutedUICommand Text* tulajdonságát.
A második eset egy kicsit bonyolultabb: bizonyos parancscsoportok egy adott vezérlőcsoporthoz köthetőek, pl. a mi esetünkben ezek a szöveges vezérlők (*RichTextBox* és *TextBox*). Ha mást nem adunk meg, akkor a parancs mindig az éppen fókuszban lévő, az ő csoportja által használható vezérlőhöz lesz kötve. Van még egy kikötés, mégpedig az, hogy ez csakis a *ToolBar* és *Menu* vezérlőkre vonatkozik, minden más esetben explicit meg kell adnunk a kötést. Ha elindítjuk a programot minden működni is fog:



Parancsok készítésével egy későbbi fejezetben foglalkozunk.

49. Animációk és transzformációk

Már megtanultuk, hogy a WPF merően új grafikus rendszerrel rendelkezik, ennek egyik hatalmas előnye, hogy olyan tulajdonságokkal vértezhetjük fel az alkalmazásainkat, amelyek megvalósítása eddig embertpróbáló feladat vagy éppen lehetetlen volt.
Ilyen lehetőség a vezérlőink „életrekeltése" is. A WPF rögtön két módszert is felkínál, animációk és transzformációk képében. Előbbi a vezérlők tulajdonságainak (pl.: helyzet, szélesség, stb…) manipulálásával dolgozik, míg transzformációk alkalmazása esetén alacsonyabb szintre visszük a dolgot és a koordinátarendszerben elfoglalt pozíciót módosítjuk.

Ebben a részben csakis 2D –s transzformációkat és animációkat nézünk meg.

49.1 Transzformációk

Minden transzformációs osztály a *System.Windows.Media.Transform* osztályból származik. Ennek az osztálynak hat leszármazottja van, ebből négy előre definiált viselkedést hordoz, egy lehetővé teszi tetszőleges transzformáció létrehozását míg az utolsó több transzformációból állhat. A következő fejezetekben mindegyikükkel megismerkedünk.
Előtte viszont megnézünk egy kis elméletet is (bár ez nem feltétele a megértésnek):
Minden transzformáció leírható egy ún. transzformációs mátrixxal:

```
  M11,      M12,     0
[ M21,      M22,     0 ]
  OffsetX,  OffsetY, 1
```

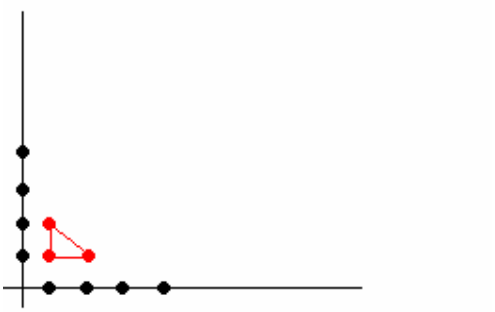
A harmadik oszlop – mivel a WPF csak lineáris transzformációkat használ – állandó lesz, az *OffsetX* és *OffsetY* tagok pedig az X illetve Y tengelyen való eltolás mértékét jelzik. A maradék négy változó a szándékunktól függően kap szerepet. A transzformációs mátrix kiindulási helyzete a következő:

```
  1 0 0
[ 0 1 0 ]
  0 0 1
```

Tehát az egységmátrix. Ennek gyakorlatilag csak a *MatrixTransform* esetén lesz szerepe.
A transzformációs mátrix segítségével kiszámolhatjuk az alakzat új koordinátáit a következő képletekkel (X és Y a régi koordinátákat jelölik X' és Y' pedig az újakat):

```
X' = X * M11 + Y * M21 + OffsetX

Y' = X * M12 + Y * M22 + OffsetY
```

Nézzünk egy gyakorlati példát! Legyen a koordinátarendszer a következő:



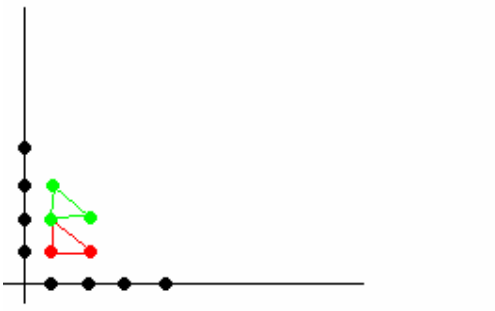
A pirossal jelölt háromszög csúcsainak a koordinátái: (1;1), (1;2) és (2;1). A transzformációs mátrix igen egyszerű lesz, mindössze az M12 változót írtuk át 1 –re:

```
 1 1 0
[ 0 1 0 ]
 0 0 1
```

Most alkalmazzuk a képletet és számítsuk ki a három új koordinátát:

```
X'1 = 1 * 1 + 1 * 0 + 0 = 1
Y'1 = 1 * 1 + 1 * 1 + 0 = 2

X'2 = 1 * 1 + 2 * 0 + 0 = 1
Y'2 = 1 * 1 + 2 * 1 + 0 = 3

X'3 = 2 * 1 + 1 * 0 + 0 = 2
Y'3 = 2 * 1 + 1 * 1 + 0 = 3
```

Az új koordináták így: (1;2), (1;3) és (2;3). Vagyis egy egységgel eltoltuk a háromszöget az Y tengelyen pozitív irányba. Természetesen ugyanezt megtehettük volna sokkal egyszerűbben is az *OffsetY* változó állításával. Az új koordináták ábrázolva (zölddel):

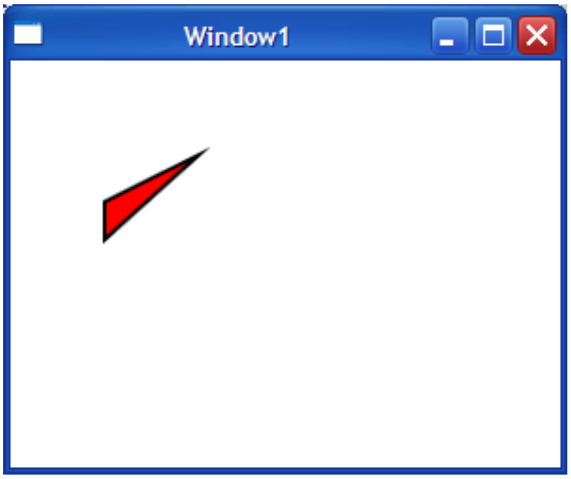


49.1.1 MatrixTransform

Az új ismereteinknek megfelelően az első vizsgált alany a *MatrixTransform* lesz, mivel ez az egyetlen olyan WPF transzformáció, amikor szükségünk van a transzformációs mátrixra, a többiek ezt a részletet elrejtik előlünk. Gyakorlatilag az esetek kilencvenkilenc százalékában nem lesz szükségünk erre a transzformációra. Vegyük a következő példát:

```
<Polygon Points="100,50 50,75 50,95 100,50"
        Stroke="Black"
        StrokeThickness="2">
  <Polygon.Fill>
    <SolidColorBrush Color="Red"/>
  </Polygon.Fill>

  <Polygon.RenderTransform>
    <MatrixTransform Matrix="1 0 0 1 0 0" />
  </Polygon.RenderTransform>
</Polygon>
```

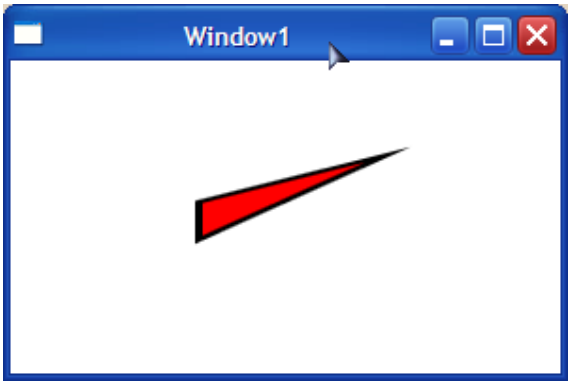
Létrehoztunk egy *Polygon* objektumot, a *Points* tulajdonságnál megadott számpárok a poligon pontjait jelzik, ebben a sorrendben köti össze őket a WPF. Viszonylag könnyű kitalálni, hogy a fönti példa eredménye egy háromszög lesz, mégpedig egy ilyen háromszög:



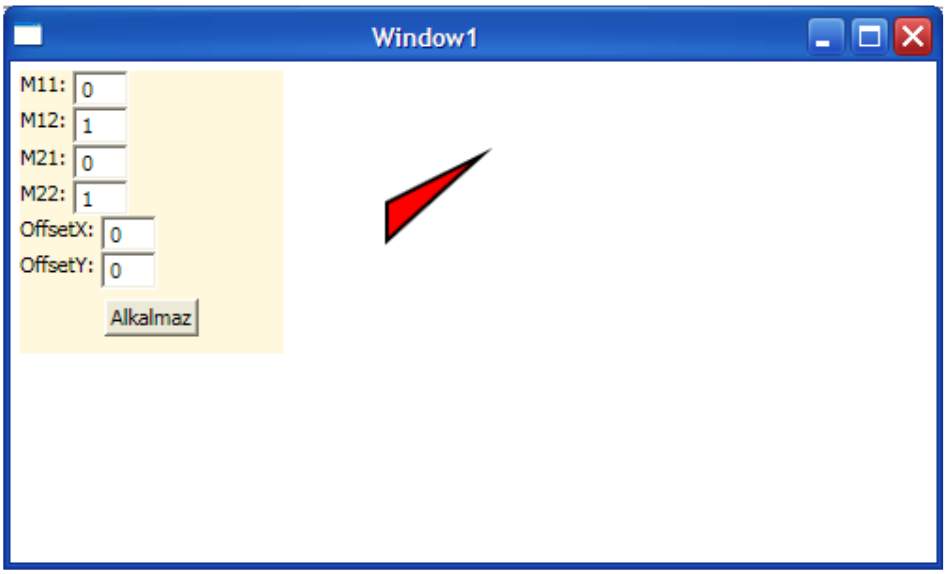
Már meg is adtuk a transzformációs mátrixot, amely épp alaphelyzetben – az egységmátrixon – áll, így hatása nincsen. A transzformációk megadásának később megtanuljuk praktikusabb módjait is, egyelőre így is jó lesz, lehet vele kísérletezni. Például legyen a mátrix a következő:

```
<MatrixTransform Matrix="2 0 0 1 0 0" />
```

Az eredmény egészen drasztikus, nemcsak eltoltuk az X tengelyen pozitív irányba, de a kétszeresére is nőtt a síkidom. A képlet segítségével elég könnyű rájönni, hogy miért van ez így:



Most készítsünk egy egyszerű programot, amely segítségével egyszerűen állíthatjuk a mátrix értékeit. A kész alkalmazás így néz ki:



Az XAML kód:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="500">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="150" />
      <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <StackPanel Background="Cornsilk"
          Height="150"
          VerticalAlignment="Top"
          Grid.Column="0"
          Margin="5">
    <WrapPanel>
      <TextBlock Text="M11: " />
      <TextBox Name="m11" Width="30">0</TextBox>
```

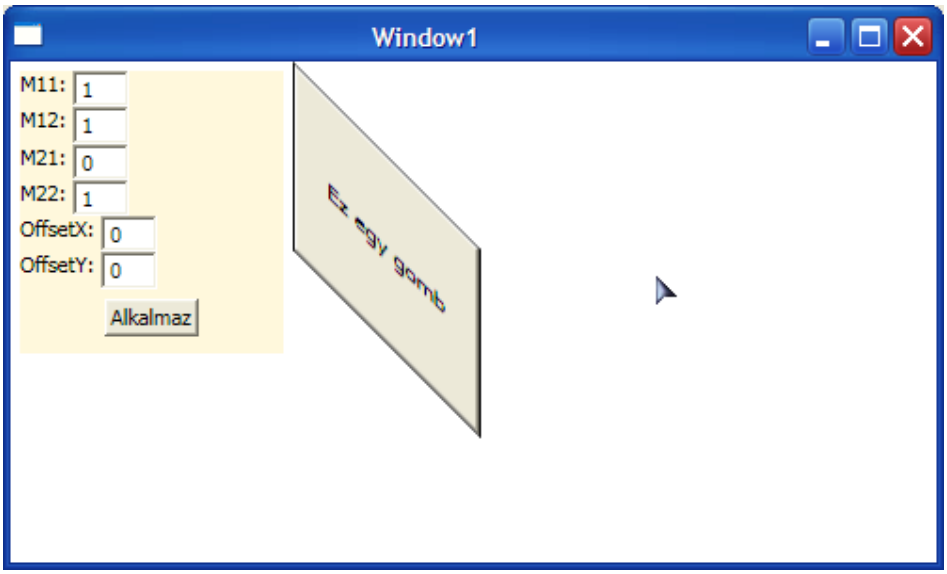
```
    </WrapPanel>
    <WrapPanel>
      <TextBlock Text="M12: " />
      <TextBox Name="m12" Width="30">1</TextBox>
    </WrapPanel>
    <WrapPanel>
      <TextBlock Text="M21: " />
      <TextBox Name="m21" Width="30">0</TextBox>
    </WrapPanel>
    <WrapPanel>
      <TextBlock Text="M22: " />
      <TextBox Name="m22" Width="30">1</TextBox>
    </WrapPanel>
    <WrapPanel>
      <TextBlock Text="OffsetX: " />
      <TextBox Name="offsetx" Width="30">0</TextBox>
    </WrapPanel>
    <WrapPanel>
      <TextBlock Text="OffsetY: " />
      <TextBox Name="offsety" Width="30">0</TextBox>
    </WrapPanel>
    <Button Name="applybutton"
            Content="Alkalmaz"
            Margin="5"
            Width="50"
            Click="applybutton_Click" />
  </StackPanel>

  <Canvas Grid.Column="1">
    <Polygon Points="100,50 50,75 50,95 100,50"
         Stroke="Black"
         StrokeThickness="2">
      <Polygon.Fill>
        <SolidColorBrush Color="Red"/>
      </Polygon.Fill>

      <Polygon.RenderTransform>
        <MatrixTransform x:Name="polytransform" />
      </Polygon.RenderTransform>
    </Polygon>
  </Canvas>
  </Grid>
</Window>
```

Végül a gombhoz tartozó eseménykezelő:

```
private void applybutton_Click(object sender, RoutedEventArgs e)
{
   Matrix m = new Matrix();
   m.M11 = double.Parse(m11.Text);
   m.M12 = double.Parse(m12.Text);
   m.M21 = double.Parse(m21.Text);
   m.M22 = double.Parse(m22.Text);
   m.OffsetX = double.Parse(offsetx.Text);
   m.OffsetY = double.Parse(offsety.Text);
```

```
    polytransform.Matrix = m;
}
```

Látható, hogy a WPF adja nekünk a megfelelő mátrix adatszerkezetet.
Az értékeket megváltoztatva elérhetjük a WPF összes lehetséges transzformációját. Természetesen nem csak az előre megadott poligont használhatjuk, hanem bármely *Shape* (pl.: *Rectangle*) illetve *FrameworkElement* leszármazottat. Ez utóbbira egy példa:

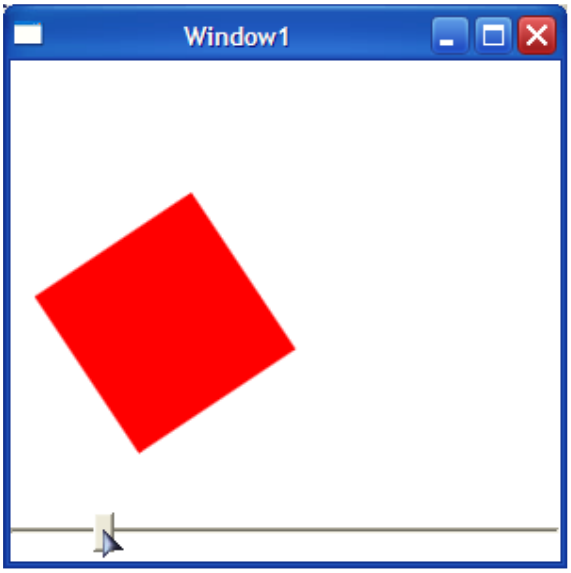


Az XAML nagyon egyszerű, csak a poligon definíciót cseréltük le:

```
<Button Content="Ez egy gomb" Width="100" Height="100">
<Button.RenderTransform>
  <MatrixTransform x:Name="polytransform" />
</Button.RenderTransform>
</Button>
```

49.1.2 RotateTransform

Ezzel a transzformációval egy adott pont körül adott szögben el tudjuk forgatni a megcélzott objektumot, ami a példánkban egy *Rectangle* lesz:



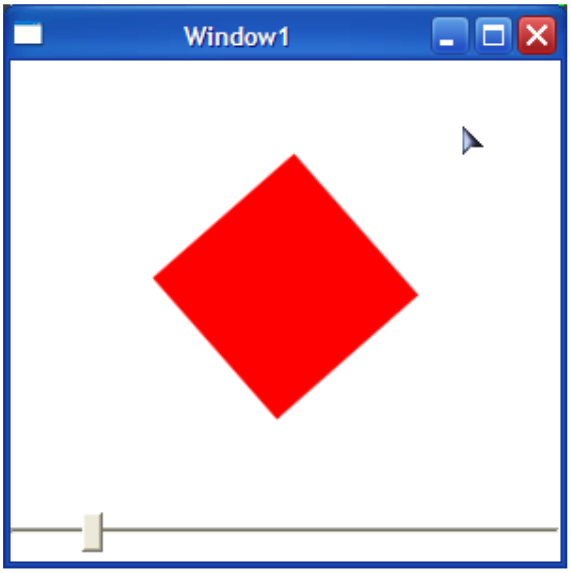
Nézzük meg a hozzá tartozó XAML –t is:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="300">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="240" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Rectangle Width="100"
               Height="100"
               Fill="Red"
               Grid.Row="0">
      <Rectangle.RenderTransform>
        <RotateTransform Angle="{Binding ElementName=angleslider, Path=Value}" />
      </Rectangle.RenderTransform>
    </Rectangle>

    <Slider Name="angleslider"
            Minimum="0"
            Maximum="360"
            Grid.Row="1" />
  </Grid>
</Window>
```

Most csakis a szöget adtuk meg, a forgatás középpontját – tehát azt a pontot ami körül forog – nem. Ilyenkor automatikusan az objektum bal felső sarkát tekinti kiindulópontnak. Adjuk meg ezeket az értékeket is! A középpont koordinátái értelemszerűen arra az objektumra értendőek, amelyen alkalmazzuk a transzformációt, vagyis az objektumon belüli koordinátára gondolunk. A bal felső sarok koordinátája ekkor (0;0) lesz. Mivel most tudjuk a négyzet méreteit nem nehéz kitalálni a valódi középpontot:

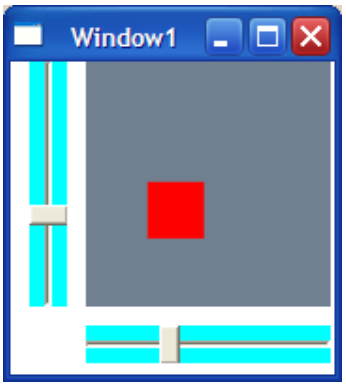
```
<Rectangle Width="100"
        Height="100"
        Fill="Red"
        Grid.Row="0">
  <Rectangle.RenderTransform>
    <RotateTransform Angle="{Binding ElementName=angleslider, Path=Value}"
                     CenterX="50"
                     CenterY="50" />
  </Rectangle.RenderTransform>
</Rectangle>
```

Az eredmény:



49.1.3 TranslateTransform

Ezzel a transzformációval a transzformációs mátrix *OffsetX* és *OffsetY* tagjait tarthatjuk kézben. Ennek megfelelően két olyan tulajdonsággal rendelkezik, amelyek hatnak az objektumra: *X* és *Y*. Nézzük a példát:

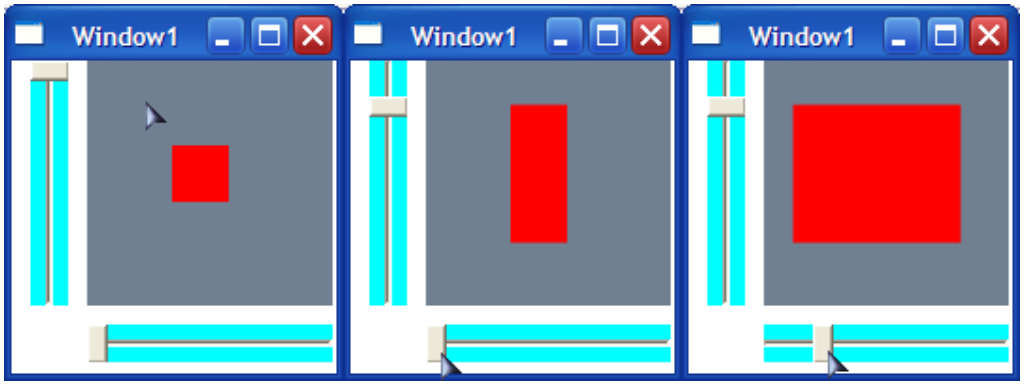


Az XAML:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="200" Width="180">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="130" />
      <RowDefinition Height="40" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="40" />
      <ColumnDefinition Width="130" />
    </Grid.ColumnDefinitions>

    <Canvas Background="SlateGray"
            Grid.Row="0"
            Grid.Column="1">
      <Rectangle Width="30"
                 Height="30"
                 Fill="Red"
                 Canvas.Top="0"
                 Canvas.Left="0">
        <Rectangle.RenderTransform>
          <TranslateTransform
                X="{Binding ElementName=xslider, Path=Value}"
                Y="{Binding ElementName=yslider, Path=Value}" />
        </Rectangle.RenderTransform>
      </Rectangle>
    </Canvas>

    <Slider Name="xslider"
            Background="Aqua"
            Minimum="0"
            Maximum="100"
            Height="20"
            Width="130"
            Grid.Row="1"
            Grid.Column="1" />

    <Slider Name="yslider"
            Background="Aqua"
            IsDirectionReversed="True"
            Orientation="Vertical"
            Width="20"
            Height="130"
            Minimum="0"
            Maximum="100"
            Grid.Row="0"
            Grid.Column="0" />
  </Grid>
</Window>
```

49.1.4 ScaleTransform

A *ScaleTransform* az X illetve Y tengelyeken való nyújtásért felel, ezeket a *ScaleX* és *ScaleY* tulajdonságain keresztül állíthatjuk. A tulajdonságok értékei szorzóként funkcionálnak, így alapértelmezett értékük 1 lesz. Ezenkívül megadhatjuk a transzformáció középponttját is a *CenterX* és *CenterY* tulajdonságokkal. A példa:

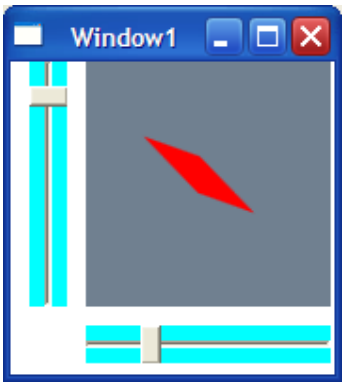



```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="200" Width="180">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="130" />
      <RowDefinition Height="40" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="40" />
      <ColumnDefinition Width="130" />
    </Grid.ColumnDefinitions>

    <Canvas Background="SlateGray"
            Grid.Row="0"
            Grid.Column="1">
      <Rectangle Width="30"
                 Height="30"
                 Fill="Red"
                 Canvas.Top="45"
                 Canvas.Left="45">
        <Rectangle.RenderTransform>
          <ScaleTransform
                 ScaleX="{Binding ElementName=xslider, Path=Value}"
                 ScaleY="{Binding ElementName=yslider, Path=Value}"
                 CenterX="15" CenterY="15" />
        </Rectangle.RenderTransform>
      </Rectangle>
    </Canvas>

    <Slider Name="xslider"
            Background="Aqua"
            Minimum="1"
            Maximum="10"
            Height="20"
            Width="130"
```

```
            Grid.Row="1"
            Grid.Column="1" />

    <Slider Name="yslider"
            Background="Aqua"
            IsDirectionReversed="True"
            Orientation="Vertical"
            Width="20"
            Height="130"
            Minimum="1"
            Maximum="10"
            Grid.Row="0"
            Grid.Column="0" />
  </Grid>
</Window>
```

49.1.5 SkewTransform

Ez a transzformáció objektumok „elhajlítására” szolgál:



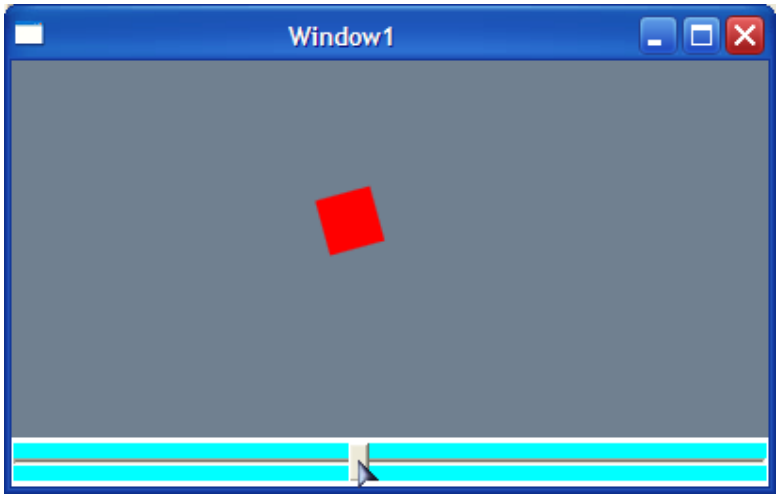
Látható, hogy ez egyfajta 3D imitáció is lehet. Az XAML:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="200" Width="180">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="130" />
      <RowDefinition Height="40" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="40" />
      <ColumnDefinition Width="130" />
    </Grid.ColumnDefinitions>

    <Canvas Background="SlateGray"
            Grid.Row="0"
            Grid.Column="1">
        <Rectangle Width="30"
                   Height="30"
```

```
                Fill="Red"
                Canvas.Top="45"
                Canvas.Left="45">
            <Rectangle.RenderTransform>
              <SkewTransform
                    AngleX="{Binding ElementName=xslider, Path=Value}"
                    AngleY="{Binding ElementName=yslider, Path=Value}"
                    CenterX="15" CenterY="15" />
            </Rectangle.RenderTransform>
          </Rectangle>
        </Canvas>

        <Slider Name="xslider"
                Background="Aqua"
                Minimum="0"
                Maximum="180"
                Height="20"
                Width="130"
                Grid.Row="1"
                Grid.Column="1" />

        <Slider Name="yslider"
                Background="Aqua"
                IsDirectionReversed="True"
                Orientation="Vertical"
                Width="20"
                Height="130"
                Minimum="0"
                Maximum="180"
                Grid.Row="0"
                Grid.Column="0" />
      </Grid>
    </Window>
```

49.1.6 TransformGroup

Ennek az osztálynak a segítségével több transzformációt tudunk összefogni. A következő példában pl. egyszerre alkalmazzuk a Rotate- és TranslateTransformation –t:



Az XAML:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="260" Width="410">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="200" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Canvas Background="SlateGray"
            Grid.Row="0">
        <Rectangle Width="30"
                   Height="30"
                   Fill="Red"
                   Canvas.Top="70"
                   Canvas.Left="0">
        <Rectangle.RenderTransform>
          <TransformGroup>
            <RotateTransform CenterX="15" CenterY="15"
                             Angle="{Binding ElementName=xslider, Path=Value}" />
            <TranslateTransform X="{Binding ElementName=xslider, Path=Value}" />
          </TransformGroup>
        </Rectangle.RenderTransform>
      </Rectangle>
    </Canvas>

    <Slider Name="xslider"
            Background="Aqua"
            Minimum="0"
            Maximum="360"
            Height="20"
            Width="400"
            Grid.Row="1" />
  </Grid>
</Window>
```

49.1.7 Transzformáció mint erőforrás

Természetesen egy transzformációt nem kell helyben kifejtenünk, megadhatjuk erőforrásként is így növelve a kódújrafelhasználást. Készítsük el az előző program erőforrást használó változatát! Sok dolgunk igazából nincsen, mindössze két helyen kell változtatnunk:

```
<Window.Resources>
  <TransformGroup x:Key="transformresource">
    <RotateTransform CenterX="15" CenterY="15"
                     Angle="{Binding ElementName=xslider, Path=Value}" />
    <TranslateTransform X="{Binding ElementName=xslider, Path=Value}" />
  </TransformGroup>
</Window.Resources>
```

```
/* ... */

<Rectangle Width="30"
      Height="30"
      Fill="Red"
      Canvas.Top="70"
      Canvas.Left="0"
      RenderTransform="{StaticResource transformresource}" />
```

Ez a lehetőség egyúttal azt is jelenti, hogy akár stílusokhoz is rendelhetünk transzformációt.

49.2 Animációk

A WPF esetében az animációk tulajdonképpen tulajdonságok manipulációjával jönnek létre. Hogy még pontosabbak legyünk csakis olyan objektumokat animálhatunk, amelyek megfelelnek három feltételnek:

- Az animálandó tulajdonság egy dependency property
- Az osztály aminek a példányán az animációt alkalmazzuk megvalósítja az *IAnimatable* interfészt és a *DependencyObject* osztályból származik (ld. első pont)
- Kell legyen alkalmas animációtípus a tulajdonsághoz (erről mindjárt)

Ezek a feltételek első ránézésre szigorúnak tűnnek, de nem azok. Szinte az összes vezérlő eleget tesz a feltételeknek, szóval a legnagyobb akadályt a képzeletünk lehet.
Említettük, hogy „alkalmas animációkra” van szükségünk. Mit is jelenthet ez? Mondjuk inkább úgy: kompatibilis. Egész egyszerűen arról van szó, hogy az adott tulajdonság típusát kezelni tudú animációs objektumra van szükségünk. Fogjuk majd látni, hogy a legtöbb beépített típushoz létezik a megfelelő animáció, de ha ez nem elég mi magunk is készíthetünk ilyet (és fogunk is a fejezet végén).

A WPF animációit három csoportra oszthatjuk és ennek megfelelően az elnevezési konvenciók is különbözőek lesznek:

- *Linear Interpolation*: a legegyszerűbb mind közül, a dependency property értéke egy kezdő- és végpont között fokozatosan változik. Ilyen pl. – ahogyan azt majd látni is fogjuk – a *DoubleAnimation*. Ezt a típust *from-to-by* animációnak is nevezzük. Az ide tartozó animációk neve *Animation* utótagot kap.
- *Key Frame Animation*: míg az előző típus megadott kezdő- és végértékkel dolgozott, addig itt ez nincs megkötve, azaz a dependency property értéke tetszőlegesen megváltoztatható egy adott pillanatban. A Key Frame animációk utótagja: *AnimationUsingKeyFrames*.
- *Path-Based Animation*: ebben az esetben az animálandó objektumot mozgásra bírjuk úgy, hogy egy általunk meghatározott útvonalat kövessen. Az utótag: *AnimationUsingPath*.

Mielőtt továbblépünk a gyakorlati dolgok felé, még két osztályról kell megemlékeznünk. Az első minden animációk ősatyja a *TimeLine*. Minden egyes

animációtípus ebből az osztályból származik. A szerepe az, hogy a neki beállított időszelet viselkedését felügyelje: meghatározza a hosszát, hányszor ismételjen, stb…
A másik fontos osztály a *StoryBoard*. Ő egy *TimeLine* leszármazott (a *ParallelTimeLines*, ez képes több *TimeLine* kezelésére) leszármazottja. Ez az osztály közvetlenül kontrollálja a „gyermekanimációit". XAML kódban nem is definiálhatunk animációt *StoryBoard* nélkül. Ezenkívűl a *StoryBoard* fogja szolgáltatni azokat a tulajdonságokat, amelyek az animáció céljának kijelölésére szolgálnak.

49.2.1 From-to-by animációk

Eljött az ideje, hogy tudásunkat a gyakorlatban is kamatoztassuk. Egy egyszerű példával fogunk kezdeni, készítünk egy *Rectangle* objektumot és hozzákötünk egy animációt, amely az egérmutató *Rectangle* fölé érésekor szép lassan megváltoztatja az objektum színét. Bonyolultabb elmondani, mint megcsinálni. Az esemény bekövetkezését természetesen egy *EventTrigger* fogja jelezni. Az XAML:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="200" Width="200">

  <Grid>
    <Rectangle Width="150" Height="150">
      <Rectangle.Fill>
        <SolidColorBrush x:Name="brush" Color="Red" />
      </Rectangle.Fill>

      <Rectangle.Triggers>
        <EventTrigger RoutedEvent="Rectangle.MouseEnter">
          <BeginStoryboard>
            <Storyboard>
              <ColorAnimation Storyboard.TargetName="brush"
                              Storyboard.TargetProperty="Color"
                              To="Black" Duration="0:0:3" />
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
      </Rectangle.Triggers>
    </Rectangle>
  </Grid>
</Window>
```

Akárcsak a transzformációk az animációk is tökéletesen használhatóak stílusokkal vagy sablonokkal (erre már láttunk is példát). A következő példák az egyszerűség kedvéért csakis „helyi" animációkkal dolgoznak.

A *ColorAnimation To* tulajdonsága a célszínt fogja jelölni, míg a *Duration* az időtartamot amíg eljutunk oda. Ez utóbbi a klasszikus óra-perc-másodperc formátumot követi, vagyis a fenti példában az animáció három másodperc alatt lesz kész. A *Storyboard.TargetType/Property* tulajdonságok attached property –k, vagyis akár a *StoryBoard* deklarációban megadhattuk volna őket.

Azt bizonyára mindenki kitalálta, hogy egy olyan animációtípussal ismerkedtünk meg, amely színekhez kötődő tulajdonságokra specializálódott. Ez eléggé leszűkíti a lehetséges célpontokat, nézzünk meg egy általánosabb típust:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="250" Width="250">

  <Grid>
    <Rectangle Width="150" Height="150">
      <Rectangle.Fill>
        <SolidColorBrush Color="Red" />
      </Rectangle.Fill>

      <Rectangle.Triggers>
        <EventTrigger RoutedEvent="Rectangle.MouseEnter">
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimation Storyboard.TargetProperty="Width"
                               To="170" Duration="0:0:1" />
              <DoubleAnimation Storyboard.TargetProperty="Height"
                               To="170" Duration="0:0:1" />
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>

        <EventTrigger RoutedEvent="Rectangle.MouseLeave">
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimation Storyboard.TargetProperty="Width"
                               To="150" Duration="0:0:1" />
              <DoubleAnimation Storyboard.TargetProperty="Height"
                               To="150" Duration="0:0:1" />
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
      </Rectangle.Triggers>
    </Rectangle>
  </Grid>
</Window>
```

Sokmindennel kiegészítettük a kódot, haladjunk sorjában: elsőként átírtuk az animációt, hogy *double* értékeket kezeljen, majd a céljának megadtuk a *Rectangle* szélességét. Ezenkívűl hozzáadtunk egy ugyanolyan animációt, a magasságra vonatkozóan. Magyarul, ha az egérmutató az objektum fölé ér, az szép lassan megnő.
Ugyanezt eljátszottuk arra az esetre is, ha az egeret elvesszük. Ez utóbbi valójában felesleges volt és csak a példa kedvéért szerepelt. Létezik ugyanis a *FillBehavior* nevű tulajdonság, amely azt határozza meg, hogy mji történjen ha az animáció véget ért. Az alapértelmezett értéke *HoldEnd*, vagyis semmi nem történik, az animáció által módosított értékek úgymaradnak. A másik lehetésges érték a *Stop*, vagyis az animáció végén visszaállnak az eredeti értékek. Van azonban egy probléma is ezzel a megoldással, mégpedig az, hogy azonnal visszaállnak az értékek, vagyis mégsem

volt olyan felelsleges a fenti megoldás, mivel így tudtunk az egér elmozdulására reagálni. Tulajdonképpen ez az egyetlen módszer, hogy eseményekhez kössük az animációk élettartamát.
A *FillBehavior* –hoz hasonló szerepet lát el az *AutoReverse*, de attól kevésbé direktebb, a „visszaállás” a tényleges animáció megfordítása annak minden tulajdonságával és időtartamával.
Az *AccelerationRatio* és *DeccelerationRatio* nemlineárissá teszik az animációt, azaz az nem egyenletes sebességgel megy végbe hanem vagy az elején vagy a végén gyorsabb. Mindkét tulajdonság egy 0 és 1 közötti (*double* típusú) értéket vár, amely a gyorsulás időtartamára vonatkozik százalékos arányban. Tehát ha az *AccelerationRation* tulajdonságnak 0.2 értéket adunk az azt jelenti, hogy az animáció az időtartamának első húsz százalékában fog gyorsítani.

Említést érdemel még a *RepeatBehavior* tulajdonság is, amellyel meghatározhatjuk, hogy hányszor ismétlődjön az animáció. XAML kódban kissé nehézkes ennek a tulajdonságnak a kezelése, mivel többféle viselkedést definiálhatunk. A következő deklarációk mindegyike helyes:

```
//az animáció kétszer ismétlődik
<DoubleAnimation RepeatBehavior="2x" />

//az animáció 5 másodpercig ismétlődik
<DoubleAnimation RepeatBehavior="0:0:5" />

//folyamatosan ismétlődik
<DoubleAnimation RepeatBehavior="Forever" />
```

A két példa alatt már biztosan körvonalazódik a tény, hogy az egyes tulajdonságokat a megfelelő típust kezelni tudó animációtípusokkal animálhatjuk. A WPF a legtöbb beépített típushoz tartalmaz támogatást, ugyanakkor vannak olyan típusok amelyeket szándékosan kihagytak, ilyenrk pl. a logikai és a felsorolt típusok (elöbbi csak két értéket vehet föl, míg utóbbi értékkészlete nincs előre definiálva). Természetesen adott a lehetőség saját animációs osztályok létrehozására, hamarosan készíteni is fogunk egyet.

Mielőtt továbbmennénk nézzünk meg egy másik példát is:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="70" Width="250">

  <Canvas>
    <Rectangle Width="20" Height="20" Canvas.Left="0">
      <Rectangle.Fill>
        <SolidColorBrush Color="Red" />
      </Rectangle.Fill>

      <Rectangle.Triggers>
        <EventTrigger RoutedEvent="Rectangle.MouseDown">
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimation Storyboard.TargetProperty="Canvas.Left"
```

```
                                To="220" Duration="0:0:5" />
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
      </Rectangle.Triggers>
    </Rectangle>
  </Canvas>
</Window>
```

Első ránézésre ez egy teljesen jól működő program. Lefordul, el is indul, de amikor el akarjuk indítani az animációt egy *InvalidOperationException* –t kapunk. Nézzük meg mégegyszer a kódot! Mi az ami más az eddigiektől? Hát persze! Egy attached property –t használtunk, ezt pedig a CLR nem tudta feloldani, hiszen nem tudja róla, hogy nem a *Rectangle* tulajdonságai közt kell keresgélnie. Amikor ilyen tulajdonságot használunk, azt jeleznünk kell a fordítónak is, hogy tudja mit hol kell keresnie. A fenti példát nagyon egyszerűen kijavíthatjuk:

```
<DoubleAnimation Storyboard.TargetProperty="(Canvas.Left)"
                 To="220" Duration="0:0:5" />
```

Tehát a szabály: ha attached property –t akarunk animálni, akkor a tulajdonság nevét zárójelek közé kell írni.

Egy objektum egyszerre több animációt is tud kezelni (itt nem a fenti példában látott „több animáció egy *StoryBoard* objektumban" esetre gondolunk, hanem arra amikor több forrásunk van (pl. két stílus)). Van azonban egy kis probléma, mivel a második animáció megérkezésekor az első azonnal leáll, ez pedig nem túl szép. A megoldást a *BeginStoryBoard HandoffBehavior* tulajdonsága jelenti. Ha ennek értékét „*Compose*" –ra állítjuk akkor a két animáció egyesül. Ennek egyetlen hátulütője a memóriaigénye, ugyanis ekkor egy teljesen új animáció jön létre, ugyanakkor az eredeti animáció(k) is a memóiában marad(nak), egészen addig amíg az animált elemet a GC el nem takarítja, vagy amíg egy új animációt alkalmazunk az objektumon.

Teljesítménynövelés céljából szabályozhatjuk az animáció framerate tulajdonságát vagyis az egy másodperc alatt megjelenített – kiszámolt – képek számát. A WPF alapértelmezés szerint 60 frame/sec értékkel dolgozik, de ha kímélni akarjuk a processzort vagy szeretnénk ha gyengébb számítógépen is jól teljesítsen az alkalmazásunk akkor érdemes ezt az értéket lejjebb venni. Az emberi szem nagyjából 25-30 frame/sec –et lát folytonos mozgásnak.

```
<Storyboard Timeline.DesiredFrameRate="30">
```

Egyetlen dolog van amiről még nem beszéltünk ez pedig a *By* tulajdonság. Ezt viszonylag ritkán használjuk, de érdemes tudni róla. Az első és legfontosabb információ róla az az, hogy nem használhatjuk egyszerre mindhárom tulajdonságot. Ha a *To* mellett a *By* is jelen van, akkor utóbbi semmilyen hatással nem lesz.
By csakis *From* –mal vagy önmagában állhat, előbbi esetben az animált tulajdonság a *From* értékétől a *From* és a *By* összegéig tart, míg utóbbi esetben az alapértéktől a *By* –ig.

49.2.2 Key Frame animációk

A from/to/by animációk meghatározott úton lineárisan működtek, egy kezdőponttal és egy végponttal. A *Key Frame* típusú animációknál nincs ilyen megkötés. A Key Frame animációk nevének utótagja *UsingKeyFrames*. Ezeknek a típusoknak van egy *KeyFrameCollection* tulajdonsága, amelyek az egyes mozgásokat rögzítik. A következő példában egy primitív „lüktetés” effektet implementálunk:

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="200" Width="200">

  <Grid>
    <Rectangle Width="40" Height="40">
      <Rectangle.Fill>
        <SolidColorBrush Color="Red" />
      </Rectangle.Fill>

      <Rectangle.Triggers>
        <EventTrigger RoutedEvent="Rectangle.MouseDown">
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="Width"
                                             Duration="0:0:4">
                <LinearDoubleKeyFrame Value="50" KeyTime="0:0:1" />
                <LinearDoubleKeyFrame Value="40" KeyTime="0:0:2" />
                <LinearDoubleKeyFrame Value="50" KeyTime="0:0:3" />
                <LinearDoubleKeyFrame Value="40" KeyTime="0:0:4" />
              </DoubleAnimationUsingKeyFrames>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
      </Rectangle.Triggers>
    </Rectangle>
  </Grid>
</Window>
```

A *KeyTime* tulajdonság a kakukktojás, ami nem egyértelmű, ez kijelöli azt az időszeletet ahol az adott *KeyFrame* működik (a példában az első az első másodpercben, a második a másodikban és így tovább…).

Ez az animációtípus maga is többfelé oszlik, aszerint, hogy miként érik el a számukra kijelölt végértéket. Az első és leggyakrabban használt ilyen típus a lineáris interpolációt (itt interpoláció alatt értjük azt, ahogyan a tulajdonságok értéke változik) használ, vagyis az animált tulajdonság értéke konstans mód, folyamatosan növekszik/csökken a neki kiszabott idő alatt. Ezt már láttuk az előző példában is. Ez a típus „*Linear*” előtagot kap.
A második az ún. diszkrét interpoláció mechanizmust használja, vagyis a megszabott végértéket a az animációnak járó idő legvégén egyetlen „mozdulattal” éri el. Egy példa:

```
<Window x:Class="JegyzetWPF.Window1"
```

```
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="200" Width="200">

  <Grid>
    <Rectangle Width="40" Height="40">
      <Rectangle.Fill>
        <SolidColorBrush Color="Red" />
      </Rectangle.Fill>

      <Rectangle.Triggers>
        <EventTrigger RoutedEvent="Rectangle.MouseDown">
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="Width"
                                             Duration="0:0:8">
                <DiscreteDoubleKeyFrame Value="50" KeyTime="0:0:2" />
                <DiscreteDoubleKeyFrame Value="40" KeyTime="0:0:4" />
                <DiscreteDoubleKeyFrame Value="50" KeyTime="0:0:6" />
                <DiscreteDoubleKeyFrame Value="40" KeyTime="0:0:8" />
              </DoubleAnimationUsingKeyFrames>

            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
      </Rectangle.Triggers>
    </Rectangle>
  </Grid>
</Window>
```

Most a *DoubleKeyFrame* diszkrét interpolációt használó változatát vettük elő. Az egyszerű típusok legtöbbjéhez rendelkezésünkre áll mindhárom altípus (a harmadikkal is mindjárt megismerkedünk). A diszkrét interpolációt használó animációk előtagja értelemszerűen „*Discrete*” lesz.

49.2.3 Spline animációk

Végül a harmadik típus a *Spline* interpolációk családja. Ez már bonyolultabb, és a jegyzet nem is fogja részletesen tárgyalni, hiszen a teljes megértéséhez erős matematikai ismeretek szükségesek. Az animáció a *Bezier* görbén alapszik, két kontrollponttal tudjuk szabályozni, hogy az animáció az időszelet egyes pillanataban mennyire gyorsoljun/lassuljon. Egy egyszerű példa:

```
<Window x:Class="JegyzetWPF.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="200" Width="200">

  <Grid>
    <Rectangle Width="40" Height="40">
      <Rectangle.Fill>
        <SolidColorBrush Color="Red" />
      </Rectangle.Fill>
```

```
      <Rectangle.Triggers>
        <EventTrigger RoutedEvent="Rectangle.MouseDown">
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="Width"
                                             Duration="0:0:8">
                <SplineDoubleKeyFrame Value="50" KeyTime="0:0:2"
KeySpline="0.0,1.0 1.0,0.0"  />
                <SplineDoubleKeyFrame Value="40" KeyTime="0:0:4"
KeySpline="0.0,1.0 1.0,0.0" />
                <SplineDoubleKeyFrame Value="50" KeyTime="0:0:6"
KeySpline="0.0,1.0 1.0,0.0" />
                <SplineDoubleKeyFrame Value="40" KeyTime="0:0:8"
KeySpline="0.0,1.0 1.0,0.0" />
              </DoubleAnimationUsingKeyFrames>

            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
      </Rectangle.Triggers>
    </Rectangle>
  </Grid>
</Window>
```

A *KeySpline* tulajdonsággal jelöljük ki a kontrollpontokat, oly módon, hogy ezt a képzeetbeli koordinátarendszert egy 1x1 egységnyi nagyságú négyzetnek vesszük.

49.2.4 Animációk és transzformációk

Egy transzformáció aktuális állapota egy vagy több tulajdonságának értékétől függ, márpedig ha tulajdonság van, akkor azt animálni is lehet. Hozzuk össze a két fogalmat és készítsünk egy „forgó” gombot. Elsőként szükségünk van egy már létező transzformációra, ez pedig mi más is lehetne most, mint a *RotateTransform*. Másodszor egy animációra lesz szükségünk, méghozzá egy *DoubleAnimation* –ra. Már csak egyvalami kell, össze kell kötnünk a kettőt. Ezt úgy tehetjük meg, hogy a transzformációt elnevezzük, az animációban pedig erre a névre tudunk hivatkozni:

```
<Button Width="100" Height="30" Content="Anitrans button"
RenderTransformOrigin="0.5, 0.5">
  <Button.RenderTransform>
    <RotateTransform x:Name="transform" />
  </Button.RenderTransform>

  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.MouseEnter">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation Storyboard.TargetName="transform"
                           Storyboard.TargetProperty="Angle"
                           FillBehavior="Stop"
                           To="360" Duration="0:0:5" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Button.Triggers>
```

```
</Button>
```

A *RenderTransformOrigin* tulajdonságot azért állítottuk be, hogy a gomb a középpontja körül forogjon, míg az animációban a *FillBehavior* tulajdonságot *Stop* –ra állítottuk, hogy az animáció befejeztével az eredeti értékek visszaálljanak és ismét lehetőségünk legyen elindítani a mozgást.

50. Vezérlők készítése

Az eddigi fejlesztőplatformok viszonylag nagy hangsúlyt fektettek saját vezérlők készítésére. Gondoljunk csak a WPF elől Windows Forms –ra! Ott, ha pl. egy nem téglalap alakú gombra volt szükségünk, akkor kénytelenek voltunk származtatással és rengeteg kód írásával készítenünk egy „új” gombtípust.
A WPF nagyban megváltoztatta ezt az ideológiát, hiszen a stílusok, sablonok, stb. megjelenésével borzasztó egyszerűen testre szabható a vezérlők viselkedése és kinézete. Ugyanakkor továbbra is rendelkezésünkre áll a lehetőség, hogy készítsünk egy teljesen új vezérlőt. Igaz, erre az esetek nagy többségében nem lesz szükségünk, hiszen a WPF ebből a szempontból eddig elképzelhetetlen lehetőségeket nyújt.
Az egész WPF filozófia alapja az, hogy az összes vezérlő ún. *lookless* „állapotú”, vagyis a vezérlők megjelenését és funkcionalitását teljesen szétválasztották.
Épp ezért általános jótanács, hogy ha csak lehet hagyatkozzunk az új elemekre és csak a legvégső esetben nyúljunk a „kőkori” eszközökhöz.

A későbbiekben megismerkedünk majd az adatkezeléshez szükséges sablontípusokkal is, ezzel is bővítve a fegyvertárunkat.

Felmerülhet a kérdés, hogy „akkor mikor van szükségünk sajátkészítésű vezérlőkre?”. Nos, míg a megjelenése egy vezérlőnek gyorsan megváltoztatható, addig a viselkedése, funkcionalitása már más tészta. Gondoljunk pl. arra, hogyegy olyan TextBox –ra van szükségünk, amely bizonyos formátumú adatot vár, pl. egy telefonszámot. Itt bizony már szükségünk van arra, hogy új alapokra helyezzük a TextBox viselkedését.
Hasonlóan elődjéhez a WPF is alapvetően két választási lehetőséget ad: származtatunk vagy pedig UserControl –t készítünk. Innentől viszont a játékszabályok teljesen megváltoznak: a UserControl –ból származtatás lesz a „nem is igazi vezérlő” típus, hiszen ezt a módszert többnyire (de nem kizárólagosan) több már létező vezérlő összefogására használjuk. Windows Forms –szal való munka során ez egy viszonylag gyakran alkalmazott módszer volt, a WPF felépítése miatt azonban kicsit háttérbe szorult, nagy részben a limitált lehetőségei miatt (erről a megfelelő fejezetben).
A következő fejezetekben mindkét úttal megismerkedünk majd, megtanuljuk, hogy hogyan hozzunk létre olyan vezérlőket, amelyek tökéletesen illeszkednek a WPF filozófiájához, illetve, hogyan vértezzük fel őket a szükséges tulajdonságokkal (dependency property, routed event, stb…).

Windows Forms alatt általában három célpontunk volt amikor új vezérlőt készítettünk: vagy valamelyik már létező vezérlőt terjesztettük ki, vagy közvetlenül a Control osztályból származtattunk, vagy pedig a UserControl -ból. A WPF szerteágazó „családfával” rendelkezik, így több lehetőségünk is van, attól függően, hogy mire van szükségünk. A következőkben megnézzük a főbb ősosztályokat:

- *FrameworkElement*: ez a legalacsonyabb szint, akkor fordulunk ehhez a megoldáshoz, ha a felületet teljes mértékben magunk akarjuk kialakítani. Ugyanakkor ez egy egy viszonylag ritkán alkalmazott módszer, mivel a legtöbb esetben jobb egy speciálisabb alapról indulni.

- *Control*: ez az osztály már rendelkezik azzal, amivel a FrameworkElement nem, mégpedig a sablonok támogatásával. Ezt az osztályt használjuk a leggyakrabban.
- *ContentControl*: olyan helyzetekben használjuk, amikor arra van szükség, hogy a vezérlő maga is tartalmazhasson más elemeket.
- *UserControl*: ő tulajdonképpen egy ContentControl, amelyet egy Border objektum foglal magába. Kifejezetten arra készítették, hogy grafikus felületen tudjuk megtervezni az új vezérlőt. Vannak azonban korlátai, erről majd a megfelelő fejezetben.
- *ItemsControl, Selector*: mindkét osztály olyan vezérlők őse, amelyek képesek gyermekobjektumok kezelésére. A kettő között a fő különbség, hogy a *Selector* (amely maga is az *ItemsControl* leszármazottja) lehetővé teszi elemek kiválasztását. A *Selector* osztályból származik többek között a *ComboBox* és a *ListBox*, míg az *ItemsControl* neves leszármazottja a *TreeView*. Érdekesség, hogy bár az *ItemsControl* nem definiál vizuális megjelenést, attól még használhatjuk közvetlenül, ha megadunk neki egy sablont.
- *Panel*: olyan vezérlőt hozhatunk létre a használatával, amelyek képesek gyermekvezérlők kezelésére.

A fentieken kívűl meg kell említsük a lehető legegyszerűbb lehetőséget is, mégpedig azt, hogy egy már létező vezérlőt egészítünk ki.

50.2 UserControl

Elsőként a „gyengébb” *UserControl* osztályból való származtatással ismerkedünk meg. Korábban ez a módszer nagyjából egyenértékű volt a „hagyományos” származtatással, de a WPF felépítése miatt mostanra háttérbe szorult. Mi is tulajdonképpen az ok? Eddig jónéhány WPF alkalmazást készítettünk és a felhasználói felület felépítését megszabtuk az XAML kódban. Az alkalmazás „dinamikus” működését a sablonok, stílusok, animjációk jelenlétének köszönhette, így nem volt rá szükség, hogy a meglévő vezérlőket különösebben átvariáljuk. Az XAML kód pedig adta a könnyű kezelhetőséget. Amikor egy WPF usercontrol –t készítünk, akkor ez a dinamizmus kicsit felborul, mivel egy usercontrol eleve megszabott kinézettel rendelkezik, amelyet viszonylag nehezen lehet módosítani (de nem lehetetlen, ezt is megnézzük).

Ebben a fejezetben elsőként elkészítünk egy viszonylag egyszerű usercontrol –t, felruházzuk a dependency property –kkel, routed event –ekkel, stb., végül pedig megnézzük, hogyan tehető a megjelenése dinamikussá.

A szóban forgó vezérlő egy űrlap lesz, amellyel a felhasználó megadhatja az adatait (név, cím, telefonszám), illetve majd meg is jeleníthessük vele azokat. Már most el kell döntenünk, hogy szeretnénk –e támogatni adatkötéseket, vagyis, hogy az űrlapot feltölthessük mondjuk egy adatbázisból. A válasz ebben az esetben igen lesz (persze, ha csak adatbekérésre lenne szükség, akkor nemet is mondhatunk), mivel így megtanuljuk, hogyan kell dependency property –t készíteni, mivel a WPF az ilyen tulajdonságokkal szeret adatkötni (erről hamarosan).

Tegyük fel, hogy mindössze három adatra vagyunk kíváncsiak: név, cím és telefonszám, ez három darab dependency property –t kíván.

Tegyük meg az előkészületeket, a projecten jobb egérgommbal kattintva válasszuk ki az Add menüpontot, azon belül pedig a User Control –t. Nevezzük el mondjuk Form –nak. Ezután létrejön a vezérlő, látható, hogy gyakorlatilag ugyanolyan felületen dolgozhatunk, mint eddig, azzal a különbséggel, hogy a legfelsőbb szintű elem most a UserControl lesz.
Tulajdonképpen ez a tervező nézet a legnagyobb előnye UserControl –ok használatának.

Adjunk hozzá egy osztályt is a projecthez FormPart.cs néven. Ezt az átláthatóság miatt tesszük, mivel a Form osztály egy parciális osztály megvan a lehetőségünk, hogy tovább bontsuk és most élünk is vele. Az „új osztályunk" fogja tartalmazni a dependency property –ket. Kiindulásként nézzen ki valahogy így:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace JegyzetTestWPF
{
   public partial class Form
   {

   }
}
```

Ne felejtsük el a névtereket is hozzáadni, nélkülük nem használhatjuk a WPF lehetőségeit.
Most készítsük el a három DP –t. Minden dependency property statikus tag kell legyen, valamint a nevük konvenció szerint Property utótaggal rendelkezik.:

```
static public DependencyProperty UserNameProperty;
static public DependencyProperty AddressProperty;
static public DependencyProperty PhoneProperty;
```

A név tulajdonságánál azért nem csak simán *NameProperty* lett, mert azt a *FrameworkElement* már lefoglalta magának.

Még korántsem vagyunk készen, szükségünk van egy statikus konstruktorra ahol regisztráljuk a tulajdonságokat, megadva nevüket, típusukat valamint az osztályt amelyhez tartoznak:

```
static Form()
{
   UserNameProperty = DependencyProperty.Register
      (
         "UserName", typeof(string), typeof(Form)
      );

   AddressProperty = DependencyProperty.Register
      (
         "Address", typeof(string), typeof(Form)
      );

   PhoneProperty = DependencyProperty.Register
      (
         "Phone", typeof(string), typeof(Form)
      );
}
```

Itt álljunk meg egy kicsit! A fenti esetben a legegyszerűbb módot használtuk a regisztrációhoz, nézzünk meg néhány bonyolultabb esetet. Tegyük fel, hogy a DP egy egész számot vár, amely nem lehet kisebb nullánál:

```
static public DependencyProperty PositiveIntProperty;

PositiveIntProperty = DependencyProperty.Register
   (
      "PositiveInt", typeof(int), typeof(Form),
      new FrameworkPropertyMetadata(int.MaxValue, null, null),
      new ValidateValueCallback(IsPositive)
   );

static public bool IsPositive(object value)
{
   return (int)value >= 0;
}
```

A *FrameworkPropertyMetadata* osztály segítségével a dependency property működéséhez adhatunk meg adatokat. Az első paraméter az alapértelmezett értéke a tulajdonságnak, a második paraméter amely egy *FrameworkPropertyMetadataOptions* enum egy értékét veszi fel olyan dolgok beállítsára szolgál, mint pl. az adatkötések engedélyezése, hogy a renderelés melyik fázisában vegye figyelembe a tulajdonság értékét a WPF, stb... A harmadik paraméter tulajdonképpen egy eseménykezelő, amely a tulajdonság megváltozásakor fut le, hamarosan megnézzük ezt is. Nekünk most a *Register* metódus utolsó paramétere fontos, amely szintén egy eseménykezelőre mutat, ez szolgál a kapott érték ellenőrzésére. A Register metódus rendelkezik még további paraméterekkel is, hamarosan őket is megnézzük.

Egyetlen dologgal tartozom még, ez pedig az, hogy hogyan lesz mindebből „igazi” tulajdonság. A választ a *DependecyObject* –től örökölt *GetValue* és *SetValue* metódusok jelentik, amelyek a dependency property –k értékének lekérdezésére illetve beállítására szolgál.

```
public int PositiveInt
{
   get
   {
      return (int)GetValue(PositiveIntProperty);
   }
   set
   {
      SetValue(PositiveIntProperty, value);
   }
}
```

Érdemes figyelni arra, hogy ebben a „burkoló” tulajdonságban ne nagyon használjunk semmilyen ellenőrzést az adatok felé, mivel a *SetValue* önmagában is gond nélkül hívható.

Teszteljük is le az új tulajdonságot, ehhez adjuk hozzá mondjuk az új vezérlőnkhöz, majd valahol a kódban példányosísunk egy *Form* objektumot, most nem jelenítjük meg:

```
Form f = new Form();
f.PositiveInt = 10;
f.PositiveInt = -1;
```

Fordulni lefordul, de kapunk egy *ArgumentException* –t, mivel a -1 nem megfelelő érték.

Térjünk vissza a *Register* metódushoz, azon belül is a *FrameworkPropertyMetadata* konstruktorához. Ennek utolsó paramétere szintén egy eseménykezelőre mutat, amely a kapott érték korrigálására szolgál:

```
PositiveIntProperty = DependencyProperty.Register
   (
      "PositiveInt", typeof(int), typeof(Form),
      new FrameworkPropertyMetadata(
         int.MaxValue,
         FrameworkPropertyMetadataOptions.None,
         new PropertyChangedCallback(PositivePropertyChanged),
         new CoerceValueCallback(CoercePositiveValue)),
      new ValidateValueCallback(IsPositive)
   );
```

Egyúttal elkészítettük az érték változásakor lefutó eseménykezelőt is. Van tehát három metósusunk, amelyek a tulajdonság értékének változásakor futnak le. Jó lenne tudni, hogy milyen sorrendben érkeznek:

- Elsőként a *ValidateValueCallback* fut le. Ami nagyon fontos, az az, hogy itt csakis a kapott értéket ellenőrizhetjük, magához a fogadó objektumhoz nem férünk hozzá.
- Következő a sorban a *CoerceValueCallback*. Ebben a fázisban finomíthatunk a kapott értéken, illetve figyelembe vehetünk más dependency property –ket

is, mivel hozzáférünk a küldő objektumhoz. A *CoerceValueCallBack* használatára jó példa a *Slider* osztály: rendelkezik Minimum és Maximum értékekkel, amelyeket az aktuális érték nem léphet át.
- Végül a *PopertyChangedCallback* jön. Itt elindíthatunk egy eseményt, amellyel értesítjük a rendszert, hogy valami megváltozott.

Felmerülhet a kérdés (méghozzá jogosan), hogy nem –e lehetne ezekkel a metódusokkal kikerülni a felhasználó által megadott helytelen adatokat. Lehetne, de nem ajánlott. Egész egyszerűen nem lehet arra építeni, hogy kijavítjuk a felhasználó hibáit, az ajánlott módszer továbbra is egy érthető hibaüzenet legyen.

Most nézzük a metódusokat:

```
static private object CoercePositiveValue(DependencyObject d, object value)
{
   return value;
}
```

Ugyan most nem csináltunk semmi különöset, azért sok lehetőségünk van: ha az érték valamiért nem megfelelő a *DependencyObject* metódusait használhatjuk:

- *ClearValue*: törli a paraméterként megadott dependency property helyi értékét (ld. korábban).
- *CoerceValue*: meghívja a paraméterként megadott DP *CoerceValueCallback* –jához rendelt eseménykezelőt.
- *InvalidateProperty*: újraszámítja az adott DP –t.
- *ReadLocalValue*: visszadja a megadott DP helyi értékét.

```
static public void PositivePropertyChanged(DependencyObject d,
        DependencyPropertyChangedEventArgs e)
{

}
```

Itt a *DependencyPropertyEventArgs* fontos nekünk. Három tulajdonsága van:

- *OldValue*: a régi érték
- *NewValue*: az új érték
- *Property*: a megváltoztatott dependency property

Most már mindent tudunk, készítsük el a vezérlőnk „igazi” tulajdonságait:

```
public string UserName
{
   get
   {
      return (string)GetValue(UserNameProperty);
   }
   set
   {
      SetValue(UserNameProperty, value);
   }
```

```
}

public string Address
{
   get
   {
      return (string)GetValue(AddressProperty);
   }
   set
   {
      SetValue(AddressProperty, value);
   }
}

public string Phone
{
   get
   {
      return (string)GetValue(PhoneProperty);
   }
   set
   {
      SetValue(PhoneProperty, value);
   }
}
```

Az egyszerűség kedvéért nem alkalmaztunk semmilyen ellenőrzést, ezt az olvasó majd megteszi az ismertetett eszközökkel.

Az adatokat felvettük, most el is kellene küldeni őket. A feldolgozás nem a vezérlő feladata lesz, az csak tárolja az információkat. Tehát, amikor a felhasználó megadta az adatait, szólni kell a rendszernek, hogy dolgozza fel azokat. Ezt legegyszerűbben úgy tehetjük meg, hogy készítünk erre a célra egy eseményt, mégpedig egy routed event –et. Az eseményt „természetesen” gombnyomásra küldjük el, de ezt hagyjuk akkorra, amikor elkészítettük a felhasználói felületet.
Routed event készítése nagyon hasonlít a dependency property –éhoz. Legyen az eseményünk neve *SubmitEvent*:

```
static public readonly RoutedEvent SubmitEvent;
```

Ezt is regisztrálnunk kell, szintén a statikus konstruktorban:

```
SubmitEvent = EventManager.RegisterRoutedEvent
   (
      "Submit", RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(Form)
   );
```

A paraméterek sorrendben: az esemény neve, stratégiája, az eseménykezelő típusa végül pedig a tartalmazó osztály.

Lehetőséget kell adnunk arra, hogy más osztályok is feliratkozzanak az eseményre, innentől a „hagyományos” eseménydefiníciót használjuk:

```
public event RoutedEventHandler Submit
{
   add
   {
      base.AddHandler(Form.SubmitEvent, value);
   }
   remove
   {
      base.RemoveHandler(Form.SubmitEvent, value);
   }
}
```

Az eseményt el is kell indítanunk, ezt a WPF világban az *UIElement* osztálytól örökölt *RaiseEvent* metódussal tesszük meg:

```
protected void RaiseSubmitEvent()
{
   RoutedEventArgs args = new RoutedEventArgs(Form.SubmitEvent);
   RaiseEvent(args);
}
```

Eljött az idő, hogy megvalósítsuk a felhasználói felületet. Borzasztóan puritánok leszünk, ez most igazából nem lényeges:

```
<UserControl x:Class="JegyzetWPF.Form"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Height="110" Width="180" BorderThickness="2" BorderBrush="Blue">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="80" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <UniformGrid Width="150" Height="70" Rows="3" Columns="2" Grid.Row="0">
      <TextBlock Text="Név:" />
      <TextBox Name="nametextbox" />

      <TextBlock Text="Cím:" />
      <TextBox Name="addresstextbox" />

      <TextBlock Text="Telefonszám:" />
      <TextBox Name="phonetextbox" />
    </UniformGrid>

    <Button Name="submitbutton"
            Content="OK"
            Width="70" Height="20"
            Grid.Row="1"
            Click="submitbutton_Click" />
  </Grid>
</UserControl>
```

A *submitbutton Click* eseménye fogja elindítani az általunk definiált eseményt:

```
private void submitbutton_Click(object sender, RoutedEventArgs e)
{
   if (nametextbox.Text == "" || addresstextbox.Text == "" || phonetextbox.Text ==
"")
   {
      MessageBox.Show("Töltse ki az adatokat!");
   }
   else
   {
      RaiseSubmitEvent();
      ClearFields();
   }
}
```

Tettünk bele egy kis felhasználóbarátságot is, a *ClearFields* metódus pedig kitakarít:

```
public void ClearFields()
{
   nametextbox.Clear();
   addresstextbox.Clear();
   phonetextbox.Clear();
}
```

Rendben vagyunk, a legfontosabb dolgot hagytuk a végére, az új vezérlő felhasználását. Térjünk vissza a „főablakunkhoz" és nyissuk meg a hozzá tartozó XAML t. Ahhoz, hogy használni tudjuk a vezérlőt regisztrálnunk kell a tartalmazó névteret. Ezt a kód „tetején" tehetjük meg (vastagbetűvel kiemelve):

```
<Window x:Class="JegyzetWPF.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:uc1="clr-namespace:JegyzetWPF"
   Title="Window1" Height="400" Width="400" >
  <Grid>
    <uc1:Form x:Name="form" Submit="Form_Submit" />
  </Grid>
</Window>
```

Meg kell adnunk egy prefixet is, amivel majd a névtérre hivatkozunk. Szerencsére itt van IntelliSense támogatás, ezért gyorsan kiválaszthatjuk a nekünk kellő névteret.
Látható, hogy megjelenik a *Submit* esemény is, ehhez tetszőleges esménkezelőt rendelhetünk.

A következő napirendi pont a commanding támogatás bevezetése. Itt két választásunk van, vagy egy már létező parancsot használunk fel, vagy pedig újat csinálunk. Először az egyszerűbbet nézzük meg, vagyis azt amikor létező parancsot használunk. Ez a parancs az *ApplicationCommands.Save* lesz, elmentjük az űrlap tartalmát. Korábban azt mondtuk, hogy nem a form dolga lesz az adatok mentése, most ezen változtatunk egy kicsit. A mentésért felelős kód ezért mostantól a vezérlő része lesz, de ezt most nem fogjuk megírni, mivel nem kapcsolódik szorosan a témához (legyen házi feladat az olvasó számára).
Parancsot kétféleképpen adhatunk hozzá egy vezérlőhöz: vagy közvetlenül a *CommandBindings* listához adjuk hozzá, ami viszont azzal jár, hogy a

végfelhasználó tetszés szerint emódosíthatja azt, vagy pedig az „erősebb” megoldást választjuk és a *RegisterClassCommandBinding* metódussal kapcsoljuk hozzá az osztályhoz a parancsot.
Kezdjük az egyszerűbb esettel! A form konstruktorába helyezzük a következő kódrészletet:

```
CommandBinding binding = new CommandBinding
   (
      ApplicationCommands.Save, SaveCommand_Execute, SaveCommand_CanExecute
   );
this.CommandBindings.Add(binding);
```

A *SaveCommand_Execute* és a *SaveCommand_CanExecute* a parancs két eseményének kezelői lesznek, elöbbibe fogjuk helyezni az adatok mentéséért felelős kódot, míg utóbbi azt vizsgálja, hogy lehet –e mentenünk. Most azt a feltételt adtuk meg, hogy minden mező legyen kitöltve. A két metódus az osztály tagja lesznek. Így néznek ki:

```
private void SaveCommand_Execute(object sender, ExecutedRoutedEventArgs e)
{
   //mentünk...
}

private void SaveCommand_CanExecute(object sender, CanExecuteRoutedEventArgs
e)
{
   e.CanExecute = nametextbox.Text != "" && addresstextbox.Text != "" &&
phonetextbox.Text != "";
}
```

A bizotnságosabb változat nem sokban különbözik az előzőtől. Most a statikus konstruktort fogjuk használni:

```
CommandManager.RegisterClassCommandBinding(
   typeof(Form),
   new CommandBinding(
      ApplicationCommands.Save,
      SaveCommand_Execute, SaveCommand_CanExecute));
```

Ebben az esetben a kezelőmetódusok statikusak lesznek és mivel már nem részei az osztálynak ezért a sender paramétert konvertálnunk kell, hogy hozzáférjünk az eredeti példány adataihoz:

```
static private void SaveCommand_Execute(object sender, ExecutedRoutedEventArgs
e)
{
   //mentünk...
}

static private void SaveCommand_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
{
   var form = (Form)sender;
```

```
    e.CanExecute = form.nametextbox.Text != "" && form.addresstextbox.Text != "" &&
form.phonetextbox.Text != "";
}
```

Már csak egyvalami hiányzik a fegyvertárunkból, ez pedig az új parancsok létrehozásának képessége. Korábban már volt róla szó, hogy a WPF a RoutedComand vagy a RoutedUICommand osztályt használja parancsok létrehozásához, mi az elöbbit fogjuk választani. Adjunk hozzá az osztályhoz egy statikus tagot:

```
static public RoutedCommand SaveCommand;
```

A statikus konstruktorban fogjuk beállítani:

```
InputGestureCollection igc = new InputGestureCollection();
igc.Add(new KeyGesture(Key.S, ModifierKeys.Shift));

FormSaveCommand = new RoutedCommand("Save", typeof(Form), igc);
```

Az IputGestureCollection segítségével a gyorsbillentyűket állítottuk be.

51. ADO.NET

Az ADO (ActiveX Data Objects) a COM (Component Object Model – a .NET előtti fő fejlesztői platform) adatelérési rétege volt. Hídat képezett a programozási nyelv és az adatforrás között, vagyis anélkül írhattunk programot, hogy ismertük volna az adatbázist.
A .NET Framework sok más mellett ezt a területet is jelentősen felújította, olyannyira, hogy az ADO.NET és az ADO közt a név jelenti a legnagyobb hasonlóságot. Az évek során az ADO.NET a Framework szinte minden verzióváltásakor bővült egy kicsit, így napjainkban már igen széleskörű választék áll rendelkezésünkre.
A következő néhány fejezetben nagyjából időrendi sorrendben fogunk megismerkedni ezekkel az eszközökkel. Elsőként a „hagyományos" móddal kerülünk közelebbi kapcsolatba, ezek az osztályok lényegében már a .NET első verzióiban is megvoltak. Az adatbázist ún. *Data Provider* –eken keresztül érhetjük el, a lekérdezett adatokat pedig *DataSet* objektumokban tároljuk, amelyek tulajdonképpen megfeleltek egy „*in – memory*" relációs adatbázisnak, vagyis az adatok a memóriában vannak XML formátumban. Emiatt ezt a metódust „*Disconnected – model"* –nek nevezzük, hiszen nincs szükség folyamatos kapcsolatra az adatbázissal.
A következő lépést a LINQ család jelentette, amely több részre oszlik és ezek nem mindegyike tartozik az ADO.NET alá (de mindannyiukkal megismerkedünk majd). A LINQ a ***L****anguage* ***In****tegrated* ***Q****uery* kifejezés rövidítése, amely annyit tesz, hogy közvetlenül a forráskódból a nyelvben ténylegesen szereplő eszközökkel SQL szerű lekérdezéseket írhatunk. A LINQ jelentősége az, hogy nem csak relációs adatbázisból kérdezhetünk le, hanem bármilyen forrásból, amelyre implementálva van – így hagyományos objektum vagy listák esetében is használhatjuk. Hivatalosan öten alkotják a LINQ családot:

- *LINQ To Objects*: Ő nem az ADO.NET része. A memóriában lévő gyűjteményeket kérdezhetünk le vele.
- *LINQ To XML*: XML forrásokon hajthatunk végre lekérdezéseket.
- *LINQ To SQL*: A Microsoft első ORM (Object Relational Mapping) eszköze. Az adatbázis tábláit hagyományos objektumként kezelhetjük.
- *LINQ To DataSets*: Mivel a LINQ To SQL csakis Microsoft SQL adatbázissal működik ezért szükség volt valamire, ami a többivel is elbír. Miután az adatok bekerülnek egy *DataSet* –be írhatunk rá lekérdezést.

Ebből a felsorolásból egyvalaki kimaradt, méghozzá a *LINQ To Entities*. Ez a .NET Framework 3.5 SP1 –ben debütált *Entity Framework* –höz készült. Az EF a LINQ To SQL továbbgondolása, jóval rugalmasabb és sokrétűbb annál. Az EF egy másik lehetőséget is biztosít a lekérdezésekhez, az ún. *Entity SQL* –t. Hogy a kettő közt mi a különbség majd a megfelelő fejezetekben kiderül.

51.1 MS SQL Server 2005/2008 Express

A következő fejezetek megértéséhez, illetve a tanultak alkalmazásához szükségünk lesz valamilyen relációs adatbázisra. A .NET –hez legjobban nyílván valamilyen Microsoft termék illik. Szerencsére Redmondban gondoltak azokra is, akik még tanulják a mesterséget, így a „nagy" MS SQL Server –ek mellett ott vannak a kistestvérek is, az Express család. Ezek a szoftverek teljes mértékben ingyenesek, akár céges keretek közt is használhatjuk őket. Természetesen nem nyújtják az

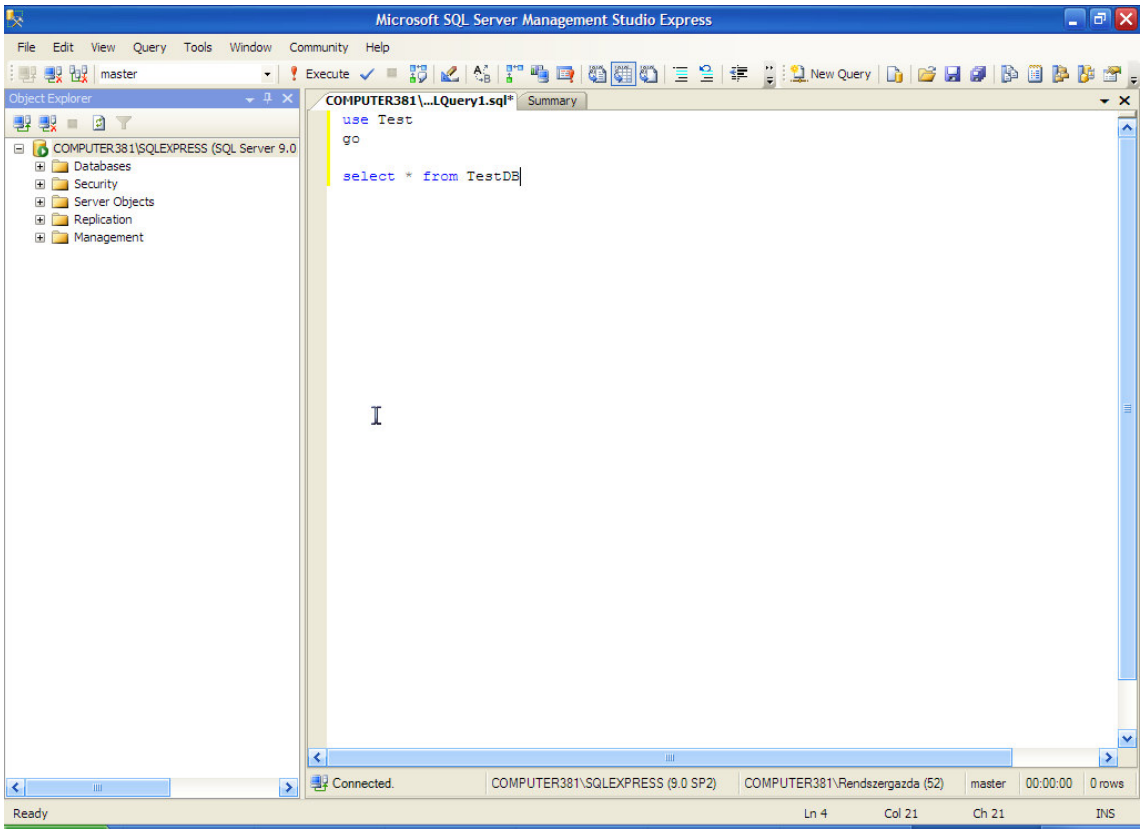
összes szolgáltatást, mint bátyjaik, de tanuláshoz (és kisebb kereskedelmi programokhoz) tökéletesek. A letöltőoldalak:

- http://www.microsoft.com/downloads/details.aspx?familyid=220549b5-0b07-4448-8848-dcc397514b41&displaylang=en

- http://www.microsoft.com/downloads/details.aspx?FamilyID=58ce885d-508b-45c8-9fd3-118edd8e6fff&DisplayLang=en

A Visual Studio 2005/2008 nem–Express változatai feltelepítik az SQL Server Express -t, ha a telepítésnél ezt kérjük.

Az adatbázisok kezeléséhez érdemes telepíteni a megfelelő verzióhoz tartozó SQL Server Management Studio Express –t is, amely grafikus felületet nyújt lekérdezések írásához, adatbázisok kezeléséhez.



A képen a Management Studio 2005 Express látható.

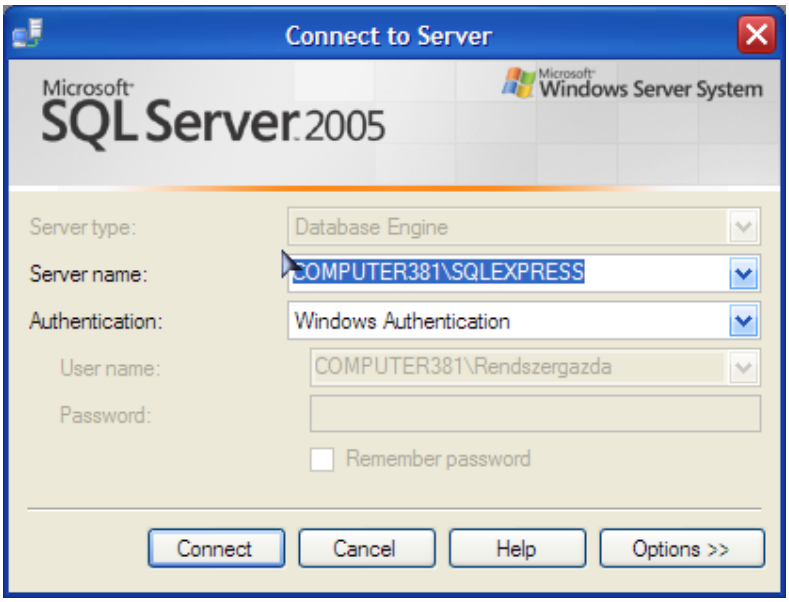
52. SQL alapok

Az SQL (Structured Query Language) egy adatbázisokhoz kifejlesztett nyelv. Tulajdonképpen ez csak egy gyűjtőnév, mivel az összes RDBMS (Relation Database Management System) saját dialektust „beszél”. Ugyanakkor az eltérések általában nem nagyok, így viszonylag könnyű váltani. A következő fejezetekben az MS SQL nyelvével ismerkedünk meg, ennek neve Transact – SQL vagy TSQL.

A következő webhelyen egy kiváló TSQL leírást talál a kedves olvasó, angol nyelven:

- http://www.functionx.com/sqlserver/index.htm

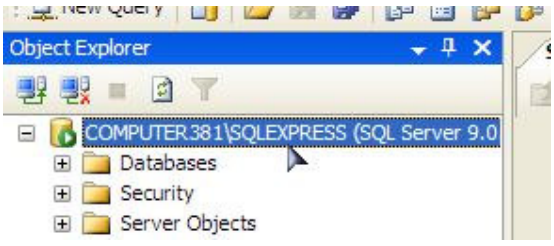
SQL lekérdezések írásához indítsuk el a Management Studio –t, a belépésnél válasszuk a Windows Authentication módot, majd kattintsunk a Connect felíratú gombra:



Természetesen a *Server name* sor változhat.

52.1 Adatbázis létrehozása

Mivel még nincs adatbázisunk, ezért kattintsunk jobb egérgombbal a bal oldalon látható Object Explorer –ben a legfelsőbb szintre:



Válasszuk a New Query menüpontot, ekkor megjelenik a szerkesztőablak.

A CREATE DATABASE paranccsal fogunk dolgozni:

```
create database TestDataBase
```

Az SQL nem case-sensitive, azaz teljesen mindegy, hogy kis- vagy nagybetűvel írjuk az utasításokat.
A végrehajtáshoz nyomjuk meg az F5 billentyűt, ekkor – ha nem rontottunk el semmit - megjelenik az adatbázisunk a Databases fül alatt (ha nem, akkor jobb egérgomb rajta és Refresh):

52.2 Táblák létrehozása

Eljött az ideje, hogy feltöltsük adatokkal az adatbázisunkat. Egy adatbázisban egy vagy több tábla tárolja az adatokat. Egy tábla sorokból (Row) és oszlopokból (Column) áll. Minden egyes oszlopnak van típusa, illetve azonosítója, amellyel hivatkozhatunk rá. Az egyes sorok ún. rekordok, amelyek egy-egy bejegyzései a táblának. Vegyük a következő egyszerű példát: egy táblában eltároljuk emberek nevét és címét. Ekkor az oszlopok neve legyen pl. *Name* és *Address*, típusuk pedig valamilyen szöveges formátum (erről bővebbet hamarosan). Egy sor a táblában pl. így néz majd ki:

Dr. A. Kula	Erdély, Baljós Kastély Fasor. 13/b.

Ezt a táblát a következőképpen tudjuk létrehozni: a Databases fül alatt keressük ki az adatbázisunkat, majd jobb egérgombbal kattintsunk rajta és New Query:

```
create table Persons (Name varchar(50), Address varchar(100));
```

A tábla neve után az oszlopok, és típusaik következnek. A *varchar* nagyjából megfelel egy string típusnak, zárójelben a maximális hossz szerepel.
Ezután, ha kinyitjuk a Databases –t, akkor a Tables alatt megtaláljuk az új adattáblánkat (ha nem látszik, akkor Refresh).
Egy táblának lehetnek speciális oszlopai. Gyakran kerülünk olyan helyzetbe, amikor egy tábla egy sorát egyértelműen meg akarjuk különböztetni a többitől, pl. legyen egy tábla a dolgozóknak, és egy a munkahelyeknek. A táblák felépítése legyen olyan, hogy a dolgozók táblája tartalmaz egy „Munkahely” oszlopot, amely a munkahely nevét tartalmazza. Ezzel a megoldással egyetlen nagy probléma van, mégpedig az, hogy feltételezi, hogy nincs két azonos nevű cég. Ha viszont van, akkor egy munkahelyre vonatkozó lekérdezésnél bajban leszünk. Erre a problémára megoldás az ún. elsődleges kulcs (*primary key v. PK*). Egy tábla több elsődleges kulcsot jelképező oszlopot is tartalmazhat, és ezek értékének egyedinek kell lennie (értsd.: nem lehet egy táblában két azonos értékű, azonos oszlopban lévő PK). Elsődleges kulcsot a következőképpen hozunk létre:

```
create table Persons (ID int primary key, Name varchar(50), Address varchar(100))
```

Elsődleges kulcsnak legegyszerűbb szimpla numerikus értéket adni (más megoldáshoz kulcsszó: GUID), ezt tettük a fenti példában is, az *int* egy egész számot

jelöl. Egy kis kényelmetlenség azonban még van, ugyanis minden esetben kézzel kell megadnunk a kulcsot, ez pedig nem túl biztonságos. Szerencsére van megoldás, automatizálhatjuk a folyamatot, megadva a kezdőértéket és a kulcs növelésének mértékét:

```
create table Persons (ID int identity(1,10)primary key, Name varchar(50), Address varchar(100))
```

Most az első PK 1 lesz és tízesével fog növekedni, tehát a másodikként beillesztett elem kulcsa 11, a másodiknak 21, stb… lesz.

Egy oszlopnak azt is megengedhetjük, hogy nullértéket tartalmazzon. Módosítsuk a fenti kifejezést, hogy a Name sor lehessen „semmi”, az Address pedig nem:

```
create table Persons (ID int identity(1,10)primary key, Name varchar(50) null, Address varchar(100) not null)
```

Elsődleges kulcs nem tartalmazhat nullértéket, ilyen táblát nem is hozhatunk létre.

52.3 Adatok beszúrása táblába

Az INSERT parancsot fogjuk használni:

```
insert into Persons (Name, Address) values ('Dr. A. Kula', 'Erdély, Baljós Kastély Fasor 13/b')
```

Megadtuk a tábla nevét, majd azokat az oszlopokat, amelyeknek értéket adunk. Most az utoljára létrehozott táblát használtuk, amelyik automatikusan állítja be az elsődleges kulcsokat, ezért azt nem kell megadnunk. Ha ezt nem kértük, akkor a PK –t is meg kell adnunk. Azokat az oszlopokat sem kell megadnunk, amelyeknek megengedtük, hogy elfogadjanak nullértéket.

52.4 Oszlop törlése táblából

Ha el akarunk távolítani egy oszlopot, akkor azt a következőképpen tehetjük meg:

```
alter table Persons drop column Name
```

Elsőként meg kell adnunk a módosítani kívánt táblát, majd a törlendő oszlop nevét.

52.5 Kiválasztás

Egy lekérdezés legelemibb tagja, amelyet minden egyes lekérdezés tartalmaz, az a *select* utasítás:

```
select * from Persons
```

Ezzel az utasítással a Persons táblából kiválasztottuk az összes oszlopot. Természetesen megadhatunk oszlopnevet is:

```
select Name from Persons
```

A wildcard karaktert (*) éles alkalmazásoknál ritkán, de leginkább soha nem használjuk, mivel szinte soha nincs szükség minden oszlopra.

52.6 Szűrés

A *where* a lekérdezett adatok szűrésére szolgál, megadhatunk vele egy feltételt, és csak azokat a sorokat kapjuk vissza, amelyek ennek megfelelnek:

```
select Name from Persons where Name='Judit'
```

Ez a lekérdezés az összes olyan sort fogja megjeleníteni a Persons táblából, amelyeknek a Name oszlopa megegyezik a megadottal.
A where számos operátort használhat, néhány példa:

```
/*Azokat a sorokat választuk ki, ahol a kor nem kissebb mint 18*/
select * from Persons where Age !< 18

/*Azokat a sorokat választjuk ki, ahol a kor 18 és 65 között van*/
select * from Persons where Age between 18 and 65

/*Azokat választjuk ki akiket nem István -nak hívnak*/
select * from Persons where Name != 'István'

/*Ugyanaz mint elöbb*/
select * from Persons where Name <> 'István'
```

Összetett feltételt is megadhatunk:

```
/*A kor nagyobb mint 18 és kissebb mint 65*/
select * from Persons where Age > 18 and Age < 65

/*A Jenő vagy József nevűeket választjuk ki*/
select * from Persons where Name = 'Jenő' or Name = 'József'
```

Azt is ellenőrizhetjük vele, hogy egy adott mező nullértéket tartalmaz:

```
select * from Persons where Age is null
```

A like operátorral tovább finoimíthatjuk a szűrést:

```
select * from Persons where Name like 're%'
```

Itt a '%' azt jelenti, hogy bármelyik és bármennyi karakter, tehát azokat a neveket keressük, amelyek a 're' „szócskával" kezdődnek.
Egy másik ilyen szűrű a '_' karakter, ez a '%' –kal ellentétben csak egyetlen karaktert jelent:

```
select * from Persons where Name like '_stvan'
```

Ebben a némileg kicsavart példában az összes olyan személyt keressük, akiknek egy valamilyen betűvel kezdődik a nevük és utána a „stvan” karaktersor áll.

A szögletes zárójelekkel több karaktert is megadhatunk, amelyek valamelyike illeszkedik:

```
select * from Persons where Name like '[AB]%'
```

Most azokat keresük, akiknek neve A vagy B betűvel kezdődik.

A where-t nagyon gyakran fogjuk használni törlésnél, módosításnál, kiválasztásnál.

Egy másik kulcsszó amely a kapott eredmény finomítására szolgál a distinct, amely a többszöri előfordulást javítja. Képzeljük el, hogy egy webáruházat működtetünk és szeretnénk megtudni, hogy melyik városokból érkeznek megrendelések. A vásárlók táblája (*Customers*) tartalmaz is egy városra vonatkozó oszlopot (*City*). Ekkor a lekérdezés így alakul:

```
select distinct City from Customers
```

Az eredeti tábla egy részlete:

Kis Miska	Baja
Kovács János	Budapest
Nagy Ede	Budapest

És az eredmény:

```
Baja
Budapest
```

52.7 Rendezés

A lekérdezett adatokat abban a sorrendben kapjuk vissza, ahogy a táblában szerepelnek. Ez persze nem mindig jó, ezért tudnunk kell rendezni a sorokat. Erre a feladatra szolgál az *order by*.

```
select Names, Address from Persons order by Address
```

A lekérdezés elején megadtuk a megnézni kívánt oszlopokat, eddig semmi szokatlan nem történt. A végén viszont megjelenik az order by, mögötte pedig az az oszlop, amely alapján majd rendezi a végeredményt. A rendezés abc szerint történik, pl. legyenek a címek a következők(a nevek most lényegtelenek):

Budapest,…
Budaörs,…
Baja,…

A rendezés után az eredmény ez lesz:

Baja,…
Budaörs,…
Budapest,…

Itt az ötödik pozicióban lesz eltérő karakter (ö – p) és ez alapján már tud rendezni.

52.8 Adatok módosítása

Az UPDATE parancsot fogjuk használni:

```
update Persons
    set Name='Béla'
    where Name='Attila'
```

Először megadjuk a táblát, majd azokat a mezőket, amelyeket módosítani akarunk az új értékekkel együtt. Ezután jön a feltétel, amivel kiválasztjuk a kívánt sort.

52.9 Relációk

A relációk megértéséhez képzeljünk el egy egyszerű példát. Vegyük a már létező Persons adatbázist és készítsünk egy új adattáblát Houses névvel. Tehát vannak személyek és hozzájuk tartozó házak. Felmerülhet a kérdés, hogy miért nem használjuk a már meglévő táblát és egészítjük ki a házakra vonatkozó információkkal? Az ok nagyon egyszerű, ugyanis egy személy több házzal is rendelkezhet. Ezt a relációt *egy–több* (onee-to-many) relációnak nevezzük, léteznek ezenkívűl *egy-egy* illetve *több-több* relációk is.
Rendben, van még egy probléma, mégpedig az, hogy hogyan rendeljük össze a két táblát. Erre fogjuk használni az elsődleges kulcsokat, vagyis az egyes házak tartalmaznak egy azonosítót, amely az őt birtokló személyre mutat (a Persons tábla megfelelő sorának elsődleges kulcsára). Ezt az azonosítót, amely egy másik táblára mutat idegen kulcsnak (foreign key) nevezzük.
Nézzük meg, hogy néz ki a két tábla (a példa nagyon egyszerű):

Persons:

ID	Name
1	Kovács János
2	Nagy Sándor

Houses:

HID	PID	Address
11	1	Budapest, …
21	2	Kecskemét, …

A táblák létehozásánál ún. *constraint* –et használunk, amely meghatározza egy oszlop tulajdonságait. Több típusa is van, mi most egy *foreig key constraint* –et fogunk bevezetni. A táblákat létrehozó utasítások a következőek lesznek:

```
create table Persons
(
	ID int not null primary key,
	Name varchar(50)
)

create table Houses
(
	HID int not null primary key,
	PID int foreign key references Persons(ID),
	Address ntext
)
```

Az idegen kulcs létehozásakor meg kell adnunk, hogy melyik tábla melyik oszlopára hivakozunk.

52.10 Join

Több táblából kérdezhetünk le a join utasítással. Vegyük az előző fejezet két tábláját és írjunk egy lekérdezést, amely visszaadja a házakhoz tartozó személyeket:

```
select Name, Address from Persons inner join Houses
	on Houses.PID = Persons.ID
```

53. Kapcsolódás az adatbázishoz

Mielőtt nekilátunk adatokkal dolgozni létre kell hoznunk egy olyan objektumot, amely segítségével parancsokat tudunk küldeni és adatokat visszakapni. Ez egy ún. *connection object* lesz, ami tulajdonképpen egy tolmács szerepét fogja betölteni köztünk és az adatforrás közt.
A kapcsolatot kétféleképpen tudjuk elkészíteni, vagy a beépített varázslókat használjuk, vagy kézzel kapcsolódunk. Mivel előbbihez szükség van plusz ismeretekre, ezért a kézi módszert fogjuk először átnézni.
Elsőként ki kell választanunk, hogy milyen adatbázishoz kapcsolódunk. Az egyszerűség kedvéért a jegyzet az MSSQL adatbáziskezelőt preferálja, de lesz példa másra is. A szükséges osztályok a *System.Data* illetve a *System.Data.SqlClient* névterekben rejtőznek.
A kapcsolatot az *SqlConnection* osztállyal fogjuk megteremteni, ehhez szükségünk lesz egy ún. connectionstring –re, amely a kapcsolódáshoz szükséges információkat (szerver neve, adatbázis neve, kapcsolódás módja, stb…) tartalmazza.
Az S*qlConnection* a *DbConnection* absztrakt osztály leszármazottja. A *DbConnection* utódja lesz minden más kapcsolódásért felelős osztály, így gyakorlatilag ugyanúgy dolgozhatunk egy MySQL szerverrel, mint az MSSQL -lel.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace JegyzetTestDB
{
   class Program
   {
      static void Main(string[] args)
      {
         SqlConnection connection = new SqlConnection
         (
         @"Data Source=computer381\SQLEXPRESS;Initial
Catalog=testDataBase;Integrated Security=True"
         );

         connection.Open();

         Console.WriteLine(connection.State);

         connection.Close();

         Console.ReadKey();
      }
   }
}
```

Ebben a programban a már elkészített adatbázishoz csatlakoztunk. Az *SqlConnection* konstruktora a connectionstring –et kapja paraméterül (be lehet ezt állítani utólag is, a *ConnectionString* tulajdonságon keresztül). Egy connectionstring sokkal bonyolultabb is lehet, most a lehető legegyszerűbbet alkalmaztuk. Elsőként a

szerver nevét adtuk meg, ahol az adatbázis megtalálható. Ez – ha a szerver a saját gépünk - lekérdezhető a parancssorba beírt *hostname* paranccsal. Használhatjuk helyette az univerzálisabb *(local)* –t is, erre hamarosan lesz példa. Az SQLEXPRESS az SQL Server Express példányának a neve, ez alapértelmezett telepítésnél mindig ugyanaz lesz. A következő a sorban az adatbázis – és nem a tábla – neve. Ezután megnyitjuk a kapcsolatot. Most már dolgozhatunk az adatbázissal, kezdetnek lekérjük a kapcsolat állapotát, ha mindent jól csináltunk, akkor a képernyőn meg kell jelennie az *Open* szónak. Ha viszont elrontottuk (pl. elírtuk az adatbázis nevét), akkor egy *SqlException* kivételt fogunk kapni. Végül bezárjuk a kapcsolatot. Szerencsére az *SqlConnection*, pontosabban a *DbConnection* megvalósítja az *IDisposable* interfészt, így nem muszáj mindig kézzel lezárni a kapcsolatot, egyszerűen betehetjük az egészet egy using blokkba:

```
using(SqlConnection connection = new SqlConnection(...))
{
    connection.Open();
}
```

Egy connectionstring elég bonyolult is lehet, így megadásakor könnyen hibázhatunk. A .NET tartalmaz egy *DbConnectionStringBuilder* nevű osztályt, ami a connectionstring felépítését könnyíti meg. Ezt az osztályt az összes provider specializálja, így létezik *OracleConnectionStringBuilder*, *OleDbConnectionStringBuilder*, stb… Mi most az *SqlConnectionStringBuilder* –t fogjuk használni:

```
SqlConnectionStringBuilder csBuilder = new SqlConnectionStringBuilder();
csBuilder.DataSource = @"(local)\SQLEXPRESS";
csBuilder.InitialCatalog = "testDataBase";
csBuilder.IntegratedSecurity = true;

SqlConnection connection = new SqlConnection();
connection.ConnectionString = csBuilder.ConnectionString;
connection.Open();

Console.WriteLine(connection.State);

connection.Close();
```

A connectionstring –ek könnyebb kezeléséhez egy másik lehetőség, hogy eltároljuk a konfigurációs file –ban.

53.1 Kapcsolódás Access adatbázishoz

A fejezet címében szereplő cselekvést a Microsoft Jet adatázis „vezérlőn” keresztűl fogjuk megtenni. Valójában a Jet jóval több szolgáltatást rejt magában, mint gondolnánk, de az évek során használata egybeforrt az Access –szel. Mi sem fogjuk másra használni, hiszen a többi adatbáziskezelőhöz van specializált .NET osztálykönyvtár.
Használatához a *System.Data.OleDb* névtérre van szükségünk, ezen belül is az *OleDbConnection* osztályra, amely egy újabb *DbConnection* leszármazott.

```
using (OleDbConnection connection = new
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data Source='test.mdf';Persist
Security Info=False"))
{
   connection.Open();
}
```

Az *OleDb* névtér osztályainak segítségével minden olyan adatforráshoz hozzáférünk, amely támogatja az OLE protokollt (pl. az Access ilyen). Ezt a névteret leginkább akkor használjuk, amikor az adatforráshoz nincs más specifikus .NET osztály.

53.2 Kapcsolódás Oracle adatbázishoz

A többivel ellentétben az ehhez szükséges névtér nem szerepel az alapbeállítások között, így nekünk kell hozzáadnunk az assembly –t, ehhez jobb egérgomb a References fülön, majd Add Reference. Az assembly neve *System.Data.OracleClient*.

Oracle adatbázishoz OLE protokoll segítségével is csatlakozhatunk.

A kapcsolatot – nem meglepő módon – az *OracleConnection* osztállyal hozzuk létre.

A fentieken kívűl (sőt, inkább helyett) Oracle adatbázisokkal való fejlesztéshez érdemes telepíteni az Oracle Data Provider for .NET (ODP.NET) könyvtárakat illetve az Oracle Developer Tools for Visual Studio –t. A letöltés mindkét esetben ingyenes, és a dokumentáció is széleskörű. A letöltőoldalak:

- http://www.oracle.com/technology/tech/windows/odpnet/index.html
- http://www.oracle.com/technology/tech/dotnet/tools/index.html

54. Kapcsolat nélküli réteg

Az ADO.NET kapcsolat néküli rétegét azok az osztályok alkotják, amelyek önállóan, kapcsolat nélkül tárolják magukban az adatokat. Vagyis lekérdezzük az adatbázisból a szükséges dolgokat és azokat a kliens oldalon eltároljuk. Ezután módosításokat végzünk és ha kész vagyunk, akkor ismét kapcsolódunk a kiszolgálóhoz és végrehajtjuk a változtatásokat. Ez az egész folyamat elsőre bonyolultul hangzik, de a munka nagy részét nem nekünk kell elvégeznünk.
A kapcsolat nélküli réteg osztályai a *System.Data* névtérben vannak.

54.1 DataTable

A *DataTable* osztály lesz a kapcsolat nélküli réteg alapköve. Gyakorlatilag megfelel egy adatbázis egy adattáblájának. Egy *DataTable* oszlopokból és sorokból áll, illetve az ezeknek megfelelő *DataColumn* és *DataRow* osztályok példányaiból. Hozzunk létre egy egyszerű *DataTable* –t:

```
DataTable table = new DataTable("PersonTable");
table.Columns.Add("Name", typeof(string));
table.Columns.Add(new DataColumn("Address", typeof(string)));
```

A tábal konstruktorában megadtuk a tábla nevét (ez nem kötelező). Ezután kétféle módja következik oszlop hozzáadásának. Vagy helyben intézzük, megadva a nevét és típusát, vagy pedig egy már létező *DataColumn* példányt adunk hozzá, hasonlóan kivitelezve. A *DataTable* az oszlopait egy *DataColumnCollection* típusú gyűjteményben tárolja, amely az *InternalDataCollection* osztályból származik, amely megvalósítja az *ICollection* és *IEnumerable* interfészeket, így gyakorlatilag tömbként (is) kezelhető, illetve használható foreach konstrukcióban.
A *DataColumn* konstruktorában első helyen az oszlop neve, majd a típusa áll. Utóbbinál egy *Type* típust vár és a *typeof* operátor pont ezt adja vissza.
Amit fontos tudnunk, hogy egy *DataTable* –hoz csakis akkor adhatunk hozzá adatokat (sorokat), ha tartalmaz legalább egy oszlopot.
Egy oszlopnak számos más tulajdonsága is lehet:

```
DataColumn dc = new DataColumn("TestColumn");
dc.DataType = typeof(string);
dc.AllowDBNull = true; //nullérték engedélyezett
dc.MaxLength = 30; //maximális szöveghossz
dc.DefaultValue = "semmi"; //a sorokban az oszlop alapértelmezett értéke
dc.Unique = true; //egyedi érték az összes sora vonatkoztatva
```

Egy *DataTable*- höz elsődleges kulcso(ka)t is beállíthatunk:

```
DataTable table = new DataTable("PersonTable");
table.PrimaryKey = new DataColumn[]
{
   new DataColumn("ID", typeof(int))
};
```

A *PrimaryKey* tulajdonság egy *DataColumn* – tömböt vár, hiszen egy hagyományos SQL adatbázis is rendelkezhet több elsődleges kulccsal.
Egy elsődleges kulcshoz (de tulajdonképpen bármelyik oszlophoz) automatikusan is hozzárendelhetünk értékeket (hasonlóan mint az „igazi” adatbázisnál):

```
DataTable table = new DataTable("PersonTable");
DataColumn dt = new DataColumn("TestColumn");
dt.DataType = typeof(int);
dt.AutoIncrement = true;
dt.AutoIncrementSeed = 0; //kezdőérték
dt.AutoIncrementStep = 1; //amennyivel növeli

table.PrimaryKey = new DataColumn[]
{
   dt
};
```

Miután létrehoztuk a tábla sémáját feltölthetjük adatokkal. Ezt a *DataTable Rows* tulajdonságán keresztül tehetjük meg, amely értelemszerűen *DataRow* példányokat vár illetve ad vissza. Van azonban egy kis probléma. Nyílván azért állítottuk be az elsődleges kulcs automatikus növelését, mert azt nem szeretnénk minden alkalommal kézzel megadni. Nézzük mi okozza a bajt. A hagyományos módszer így nézne ki:

```
table.Rows.Add("Reiter Istvan", "Toalmas");
```

Ekkor a *Rows* tulajdonság *Add* metódusa egy paramétertömböt vár, amelyben a sor adatai vannak. Ebben az esetben azonban ez a sor kivételt (*ArgumentException*) okoz, mivel nem adtunk meg elsődleges kulcsot. Ez elsőre meglepő lehet hiszen beállítottuk az elsődleges kulcs tulajdonságait. A hiba oka az, hogy ilyenkor csakis a megfelelő sémával rendelkező sorokat adhatunk meg, vagyis olyat ami ismeri a tábla felépítését és automatikusan beállítja a megfelelő értékeket, vagy pedig megadjuk az összes oszlopot és így az elsődleges kulcsot is.
A *DataTable* osztálynak van is egy *NewRow* nevű metódusa, ami pont erre való, vagyis egy olyan DataRow példányt ad vissza, amely megfelel a tábla sémájának. A teljes példa most ez lesz:

```
DataTable table = new DataTable("PersonTable");

DataColumn primKey = new DataColumn("ID");
primKey.DataType = typeof(int);
primKey.AutoIncrement = true;
primKey.AutoIncrementSeed = 0;
primKey.AutoIncrementStep = 1;

table.Columns.Add(primKey);
table.PrimaryKey = new DataColumn[]
{
   primKey
};

table.Columns.Add("Name", typeof(string));
table.Columns.Add(new DataColumn("Address", typeof(string)));
```

```
DataRow drOne = table.NewRow();
drOne["Name"] = "Reiter Istvan";
drOne["Address"] = "Toalmas ...";
table.Rows.Add(drOne);

DataRow drTwo = table.NewRow();
drTwo["Name"] = "Dr. A. Kula";
drTwo["Address"] = "Erdely, Baljos Fasor 13/b.";
table.Rows.Add(drTwo);
```

A *DataRow* indexelőivel az oszlopok nevére hivatkozhatunk.
Egy tábla sorait szinte mindig érdemes a *NewRow* metódussal létrehozni, így sokkal kevesebb a hibalehetőség.

Természetesen egy *DataTable* elemeit elérhetjük a programunkban, a következő példában az előző példa tábláját íratjuk ki:

```
foreach (DataRow row in table.Rows)
{
   Console.WriteLine("Name: {0}, Address: {1}", row["Name"], row["Address"]);
}
```

Az biztosan feltűnt már, hogy az oszlopok neveihez nem kapunk semmilyen támogatást a fejlesztőeszköztől, sőt a fordító sem tud mit csinálni. Ez a „hagyományos” típustalan adattáblák legnagyobb hátránya, erre a problémára (is) megoldást jelenthet a típusos *DataSet* –ek használata.
Egy *DataTable* –hez külső adatforrásból (SQL adatbázis, XML file) is rendelhetünk adatokat, ezeket a módokat a megfelelő fejezet részletezi.

A *DataTable LoadDataRow* metódusával felül tudunk írni létező adatokat:

```
DataTable table = new DataTable("PersonTable");

table.Columns.Add(new DataColumn(("Name"), typeof(string)));
table.Columns.Add(new DataColumn(("Age"), typeof(int)));
table.Columns.Add(new DataColumn(("Address"), typeof(string)));

DataRow dr = table.NewRow();
dr["Name"] = "Reiter Istvan";
dr["Age"] = 22;
dr["Address"] = "Toalmas, ...";
table.Rows.Add(dr);

table.LoadDataRow(new object[]
   {
      "Reiter Istvan", 44, "World"
   }, LoadOption.OverwriteChanges);
```

A metódus az első kulcs (vagy az elsődleges kulcs) alapján „szétnéz” az adatbázisban és ha talál egyezést, akkor azt a bejegyzést módosítja, egyébként létehoz egy új sort. Ha most kiíratnánk a tábla adatait, akkor azt látnák, hogy egyetlen bejegyzést tartalmaz, a módosított adatokkal.

A *DataRow* –nak különböző állapotai lehetnek, amelyeket a *RowState* tulajdonsággal kérdezhetünk le (vigyázat, ezt a tulajdonságot csakis olvashatjuk). Egy sort törölhetünk, módosíthatunk, stb… de a változások csakis akkor lesznek véglegesek, ha meghívtuk az *AcceptChanges* metódust. Ezzel a metódussal szintén rendelkezik a *DataTable* osztály is, amikor meghívjuk, akkor az összes sorára is meghívódik.
Egy sor állapota a következők lehetnek:

- Detached: a *DataRow* példány már elkészült, de nincs hozzáadva egy táblához sem.
- Added: a sort hozzáadtuk egy táblához
- Unchanged: a legutolsó *AcceptChanges* hívás óta nem változott a sor értéke
- Modified: a sor változott az utolsó *AcceptChanges* óta
- Deleted: a sort töröltük a *Delete* metódussal

Néhány példa:

```
DataTable table = new DataTable("PersonTable");

table.Columns.Add(new DataColumn(("Name"), typeof(string)));
table.Columns.Add(new DataColumn(("Age"), typeof(int)));
table.Columns.Add(new DataColumn(("Address"), typeof(string)));

DataRow dr = table.NewRow();

Console.WriteLine(dr.RowState); //Detached

dr["Name"] = "Reiter Istvan";
dr["Age"] = 22;
dr["Address"] = "Toalmas, ...";

table.Rows.Add(dr);

Console.WriteLine(dr.RowState); //Added

dr["Age"] = 56;

Console.WriteLine(dr.RowState); //Added

dr.AcceptChanges();

Console.WriteLine(dr.RowState); //Unchanged

dr["Age"] = 22;

Console.WriteLine(dr.RowState); //Modified
```

Látható, hogy az *Unchanged* és a *Modified* állapotokat csakis az *AcceptChanged* hívása utá veheti fel.
A *BeginEdit* és *EndEdit* metódusokkal több módosítás is elvégezhető egy soron. Módosítsuk az előző programot, hogy az utolsó módosítás így nézzen ki:

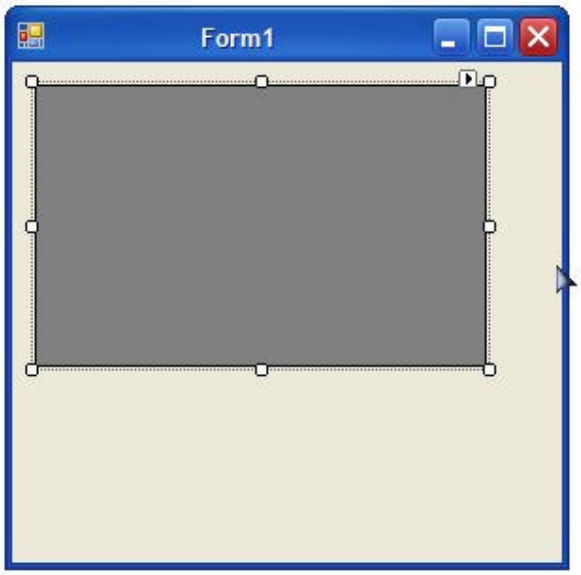
```
dr.BeginEdit();
dr["Age"] = 22;
```

```
Console.WriteLine(dr.RowState); //Unchanged
dr.EndEdit();
```

A fenti példákban a *DataRow* példányokat az eredeti azonosítójukkal kezeltük, de ezt megtehettük volna a *DataTable Rows* tulajdonságával amely *DataRowsCollection*-nel tér vissza és ennek van indexelője.

54.1.1 DataGridView

Korábban már láttunk példát adatkötésekre, amikor a *ListBox* vezérlővel és társaival ismerkedtünk. Ezt ebben az esetben is megtehetjük, hiszen a *DataTable* rendelkezik mindazon tulajdonságokkal (interfészekkel), amelyek alkalmassá teszik erre a feladatra.
Adatok megjelenítésére van egy speciális vezérlőnk, a *DataGridView*. Ez a vezérlő a .NET Framework 2.0 verziójában jelent meg először, a régi (.NET 1.0) *DataGrid* –et leváltva. Készítsünk egy új Windows Forms Application projectet! A ToolBox –ban keressük ki a *DataGridView* –ot (a Data fül alatt lesz) és húzzuk a formra. A vezérlőhöz kapcsolódó beállítások ablakot bezárhatjuk a vezérlő tetején megjelenő ki nyilacskával, most nem lesz rá szükség.

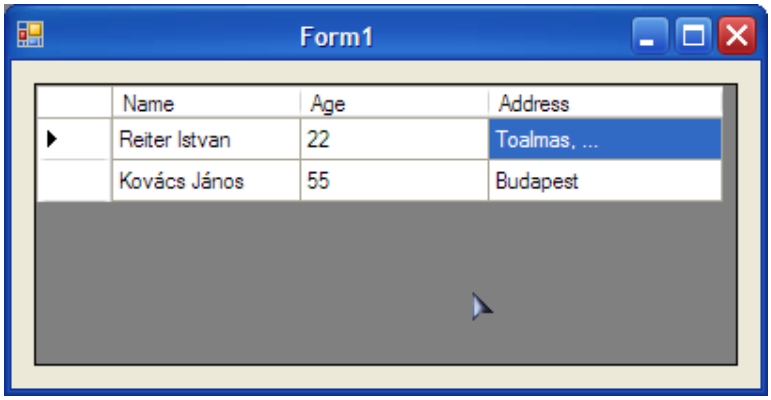


A *DataGridView AllowUserToAddRows* és *AllowUserToDeleteRows* tulajdonságainak értékét állítsuk false –ra.
Ezután hozzuk létre a form *Load* eseménykezelőjét és írjuk bele a következőt:

```
private void Form1_Load(object sender, EventArgs e)
{
    DataTable table = new DataTable("Persons");

    table.Columns.Add(new DataColumn(("Name"), typeof(string)));
    table.Columns.Add(new DataColumn(("Age"), typeof(int)));
    table.Columns.Add(new DataColumn(("Address"), typeof(string)));

    DataRow dr = table.NewRow();
```

```
    dr["Name"] = "Reiter Istvan";
    dr["Age"] = 22;
    dr["Address"] = "Toalmas, ...";
    table.Rows.Add(dr);

    DataRow dr2 = table.NewRow();
    dr2["Name"] = "Kovács János";
    dr2["Age"] = 55;
    dr2["Address"] = "Budapest";
    table.Rows.Add(dr2);

    table.AcceptChanges();

    dataGridView1.DataSource = table;
}
```

Az eredmény:



Bár a *DataGridView* alapértelmezett megjelenése kissé csúnyácska, de részletesen konfigurálható, a jegyzet ezt a témát nem részletezi, de érdemes ellátogatni a következő oldalra:
- http://msdn.microsoft.com/en-us/library/ms171598.aspx

Egy *DataGridView* –hoz bármely olyan osztály köthető, amely megvalósítja az *IList*, *IListSource*, *IBindingList* illetve *IBindingListView* interfészek valamelyikét.

Az tisztán látszik, hogy egy *DataGridView* sorokból és oszlopokból áll, ezeket *DataGridViewRow* és *DataGridViewColumn* típusokkal írhatjuk le. Az egyes cellák pedig *DataGridViewCell* típusúak, illetve ennek az osztálynak a leszármazottai, erről később még lesz szó. Több cella kombinációját a *DataGridViewBand* osztállyal csoportosíthatjuk, így pl. kiválaszthatjuk egy sor vagy oszlop összes celláját is:

```
DataGridViewBand rowBand = dataGridView1.Rows[0];
DataGrdiViewBand columnBand = dataGridView1.Columns[1];
```

A *Rows* és *Columns* tulajdonságok *DataGridViewRowCollection* illetve *DataGridViewColumnCollection* típusú gyűjteményekkel térnek vissza, sok hasonlóval találkoztunk már, ezeket is lehet indexelni, felsorolni, stb…

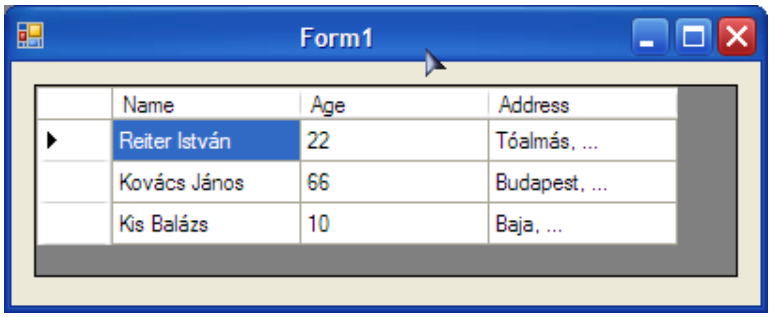
Egy *DataGridView* –hoz nem csak *DataTable* objektumokat köthetünk, hanem hagyományos listákat is. Hozzunk létre egy egyszerű osztályt:

```
public class Person
{
   public Person(string name, int age, string address)
   {
      Name = name;
      Age = age;
      Address = address;
   }

   private string name;
   public string Name
   {
      get { return name; }
      set { name = value; }
   }

   private int age;
   public int Age
   {
      get { return age; }
      set { age = value; }
   }

   private string address;
   public string Address
   {
      get { return address; }
      set { address = value; }
   }
}
```

Ennek az osztálynak a példányaival fogjuk feltölteni a *DataGridView* –ot. Egyetlen dolognak kell eleget tennünk, el kell készíteni a megfelelő tulajdonságokat, mivel ezeket kötjük majd hozzá. A táblázatban az adatok a tulajdonságok definiciójának sorrendjében jelennek meg „felülről lefelé” haladva. Most hozzuk létre a listánkat:

```
private void Form1_Load(object sender, EventArgs e)
{
   List<Person> personList = new List<Person>();

   Person p1 = new Person("Reiter István", 22, "Tóalmás, ...");
   Person p2 = new Person("Kovács János", 66, "Budapest, ...");
   Person p3 = new Person("Kis Balázs", 10, "Baja, ...");

   personList.Add(p1);
   personList.Add(p2);
   personList.Add(p3);

   table.AcceptChanges();

   dataGridView1.DataSource = personList;
}
```

Az eredmény:



Az eddigi példáink nem voltak túlzottan rugalmasak, mivel csak egyszeri adatokat jelenítettünk meg, hiszen a *DataTable* illetve a lista lokális objektumok voltak. Természetesen felmerülhet az igény, hogy módosítsuk az adatainkat a *DataGridView* –ból. A következő programunk erre is alkalmas lesz, ezenkívül új sorok beszúrását is meg fogjuk oldani.
A *DataGridView AllowUserToAddRows* tulajdonságát állítsuk *true* értékre. Ezután hozzuk létre az adattáblát, de ezúttal a form osztály tagjaként. A konstruktorban beállítjuk az oszlopokat:

```
table.Columns.Add(new DataColumn("Name", typeof(string)));
table.Columns.Add(new DataColumn("Age", typeof(int)));
table.Columns.Add(new DataColumn("Address", typeof(string)));
```

A form *Load* eseményében pedig feltöltjük a táblát és hozzákötjük a *DataGridView* –hoz:

```
DataRow dr1 = table.NewRow();
dr1["Name"] = "Kovács János";
dr1["Age"] = 55;
dr1["Address"] = "Budapest, ...";
table.Rows.Add(dr1);

DataRow dr2 = table.NewRow();
dr2["Name"] = "Kis Balázs";
dr2["Age"] = 8;
dr2["Address"] = "Kecskemét, ...";
table.Rows.Add(dr2);

dataGridView1.DataSource = table;
```

Húzzunk egy gombot is a formra, ezzel fogjuk véglegesíteni a változtatásokat, azáltal, hogy meghívjuk a *DataTable AcceptChanges* eseményét.
Három feladatunk lesz: elsőként ellenőrizzük, hogy a felhasználó megfelelő adatot adjon meg (a kornál ne szerepeljen pl. string), másodszor a már létező adatok módosítása, végül pedig új sor hozzáadása az adatforráshoz.
Haladjunk sorban: bármilyen módosítás nélkül indítsuk el az alkalmazást és valamelyik kor oszlopba írjunk néhány betűt. Ha megpróbáljuk elhagyni azt a cellát, akkor egy szép nagy *MessageBox* –ot kapunk a képünkbe. Ez – valljuk be – nem túl szép megoldás, ezért módosítjuk a programot, hogy egy barátságosabb

hibaüzenettel rukkoljon elő. A *DataGridView* bármely hiba esetén a *DataError* eseményt aktiválja, amely *DataGridViewDataErrorEventArgs* típusú paramétere hordozza az információt, amivel már azonosítani tudjuk a hiba forrását. Hozzuk létre az eseménykezelőt, de ne írjunk bele semmit. Ha most elindítjuk a programot és előidézünk valamilyen hibát (mondjuk a kor oszlopba nem numerikus karaktert írunk), akkor az alkalmazás nem szól semmit, de nem léphetünk tovább a cellából, amíg jó értéket nem adtunk meg. Finomítsunk kicsit a dolgon:

```
private void dataGridView1_DataError(object sender,
        DataGridViewDataErrorEventArgs e)
{
   if (e.Exception != null)
   {
      MessageBox.Show("Kérem adjon meg helyes adatokat!");
   }
}
```

A paraméter *Exception* tulajdonsága *null* értéket vesz fel, ha nem volt kivétel, de ha mégis, akkor kirakunk egy *MessageBox* –ot és tájékoztatjuk a felhasználót a hibájáról. Ha csak bizonyos kivételek érdekelnek minket, az is megoldható. A példánknál maradva kezeljük a *FormatException* –t, amely a helytelen formátumú, vagy típusú adatok esetében váltódik ki:

```
private void dataGridView1_DataError(object sender,
        DataGridViewDataErrorEventArgs e)
{
   if (e.Exception is FormatException)
   {
      MessageBox.Show("Kérem adjon meg helyes adatokat!");
   }
}
```

Ezenkívül az is megoldható, hogy továbbadjuk a kivételt, ha a *ThrowException* tulajdonság értékét igazra állítjuk:

```
e.ThrowException = true;
```

A *Context* tulajdonsággal a hiba körülményeiről kapunk információt, egy felsorolt típus (*DataGridViewDataErrorContexts*) formájában. Például, ha helytelen adatot adtunk meg egy cellában és megpróbálunk továbblépni, akkor a tulajdonság értéke egyszerre lesz *Parsing*, *Commit* és *CurrentCellChanged*. Ezt a logikai vagy operátorral tudjuk ellenőrizni, az értékek kombinálásával:

```
private void dataGridView1_DataError(object sender,
        DataGridViewDataErrorEventArgs e)
{
   if (e.Context == (DataGridViewDataErrorContexts.Parsing |
DataGridViewDataErrorContexts.Commit |
DataGridViewDataErrorContexts.CurrentCellChange))
   {
      MessageBox.Show("Kérem adjon meg helyes adatokat!");
   }
}
```

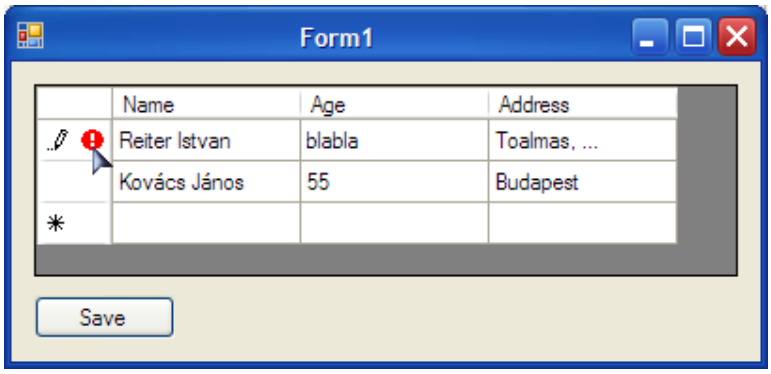
A hibát okozó cellát is egyértelműen azonosíthatjuk, a *ColumnIndex* és *RowIndex* tulajdonságokkal, amelyek a *DataGridView* soraira és oszlopaira hivatkoznak:

```
private void dataGridView1_DataError(object sender,
        DataGridViewDataErrorEventArgs e)
{
   if (e.Context == (DataGridViewDataErrorContexts.Parsing |
DataGridViewDataErrorContexts.Commit |
DataGridViewDataErrorContexts.CurrentCellChange))
   {
      MessageBox.Show("A " + e.ColumnIndex.ToString() + ", " +
e.RowIndex.ToString() + " cella helytelen adatot tartalmaz: " +
dataGridView1[e.ColumnIndex, e.RowIndex].Value.ToString());
   }
}
```

Látható, hogy maga a *DataGridView* is indexelhető, az indexelője egy *DataGridViewCell* típusú objektumot ad vissza.

Máshogy is jelezhetjük a felhasználóknak a hibás adatokat, mégpedig *ErrorProvider* –ekkel. Ezeket a *DataGridView* már „beépítve” tartalmazza, csak be kell állítanunk a sor vagy cella *ErrorText* tulajdonságát:

```
private void dataGridView1_DataError(object sender,
        DataGridViewDataErrorEventArgs e)
{
   if (e.Exception != null)
   {
      dataGridView1.Rows[e.RowIndex].ErrorText = "Hibás adatok";
   }
}
```



A cellák kezeléséhez van még néhány fontos esemény. A *CellValidating* akkor lép életbe, amikor a cella elveszíti a fókuszt. Ezzel az eseménnyel megakadályozhatjuk, hogy nem megfelelő érték kerüljön egy cellába, illetve, hogy a rákötött adatforrásban ne módosuljanak az adatok. Párja a *CellValidated*, amely közvetlenül a *CellValidating* után jön, ez az esemény utólagos formázások kivitelezésére hasznos.

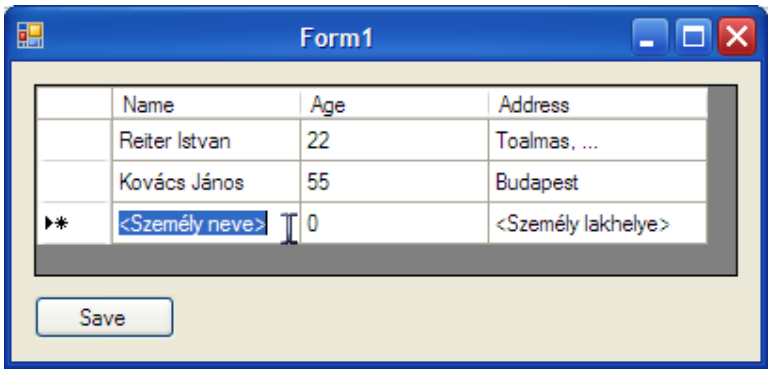
Amennyiben a megadott adatok megfelelőek, akkor a *DataGridView* mögött lévő adatforrás is módosul (ez persze csakis adatkötött környezetben igaz).

Természetesen egy *DataTable* sorainak állapota addig nem végleges, amíg meg nem hívtuk az *AcceptChanges* metódust, ezt a korábban létrehozott gombra drótozzuk rá:

```
private void button1_Click(object sender, EventArgs e)
{
   table.AcceptChanges();
}
```

Sorok hozzáadása rendkívül egyszerűen zajlik. Ha az *AllowUserToAddRows* tulajdonság igaz értéket kapott, akkor a *DataGridView* végén megjelenik egy csillaggal jelzett sor. Ha valamelyik cellájára rákerül a fókusz, akkor az új sor automatikusan létrejön és ez megtörténik a mögötte lévő adatforrásban, jelen esetben a *DataTable* –ben. A sor alapértelmezett értékekkel inicializálódik (ezeket a *DataColumn DefaultValue* tulajdonságával tudjuk beállítani), illetve ha az nincs beállítva akkor nullértékkel (ha engedélyezett: *AllowDBNull* tulajdonság, alapértelmezetten igaz értékű). Az alapértelmezett sorértékeket futás közben is megváltoztathatjuk a *DataGridView DefaultValuesNeeded* eseményében, amely akkor fut le, amikor az új sorra kerül a fókusz.

```
private void dataGridView1_DefaultValuesNeeded(object sender,
DataGridViewRowEventArgs e)
{
   e.Row.Cells[0].Value = "<Személy neve>";
   e.Row.Cells[1].Value = 0;
   e.Row.Cells[2].Value = "<Személy lakhelye>";
}
```



Ekkor az adatforrás még nem módosul, illetve, ha nem írunk be semmit és visszalépünk egy másik sorba, akkor ezek az értékek eltűnnek. Ezt az eseményt a felhasználó irányítására használhatjuk.

Sorok törléséhez engedélyeznünk kell az *AllowUserToDeleteRows* tulajdonságot, az *EditMode* tulajdonság értéke legye *EditOnKeyStrokeOrF2* illetve a *SelectionMode* legyen *FullRowSelect* vagy *RowHeaderSelect*. Ekkor egyszerűen kattintsunk valamelyik sorra, hogy kijelöljük és nyomjuk le a Del billentyűt.
Ezenkívűl használhatjuk még a *DataGridView Rows* tulajdonságának *RemoveAt* metódusát is, amely az eltvolítandó sor indexét várja paraméterként.

Egy *DataGridView* nem csak hagyományos cellákat tartalmazhat, hanem specializáltabb változatokat is, mint pl. egy kép vagy egy *ComboBox*. Akár arra is van lehetőség, hogy egy oszlopot – a megjelenített érték típusától függően – módosítsunk. A specializált oszlopok specializált cellákból állnak, tehát egy *DataGridViewTextBoxColumn* típusú oszlop cellákra vonatkozó tulajdonsága *DataGridViewTextBoxCellCollection* típusú gyűjteményt fog visszaadni, amelynek tagjai értelemszerűen *DataGridViewTextBoxCell* típusúak lesznek.

Egészítsük ki a programunkat, hogy kiválaszthassa a felhasználó a személyek nemét. Elsőként egészítsük ki ezzel az oszloppal a *DataTable* –t:

```
table.Columns.Add(new DataColumn(("Sex"), typeof(string)));
```

Ezzel más dolgunk nincsen, hiszen alapértelmezés szerint engedélyezett nullértéket megadni erre az oszlopra, így a a táblabeli adatainkon nem kell változtatni.
A *DataGridView* ennél problémásabb lesz. Az oszlopokat kézzel fogjuk létrehozni, ehhez állítsuk az *AutoGenerateColumn* tulajdonságot *false* értékre, így megakadályozzuk, hogy a vezérlő az adatkötéskor elkészítse a megfelelő oszlopokat. Most hozzáadjuk a már ismert adatok oszlopait (az egész a form *Load* eseményében zajlik):

```
dataGridView1.AutoGenerateColumns = false;

DataGridViewTextBoxColumn tbc = new DataGridViewTextBoxColumn();
tbc.DataPropertyName = "Name";
tbc.DisplayIndex = 0;
tbc.Name = "Name";

DataGridViewTextBoxColumn tbc1 = new DataGridViewTextBoxColumn();
tbc1.DataPropertyName = "Age";
tbc1.DisplayIndex = 1;
tbc1.Name = "Age";

DataGridViewTextBoxColumn tbc2 = new DataGridViewTextBoxColumn();
tbc2.DataPropertyName = "Address";
tbc2.DisplayIndex = 2;
tbc2.Name = "Address";
```

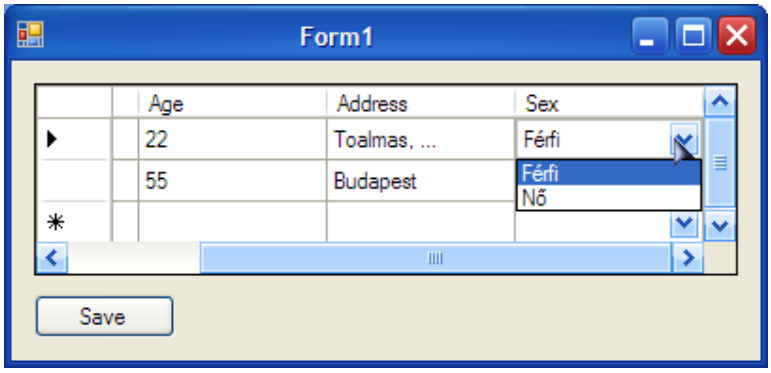
A *DataPropertyName* tulajdonság azt a tulajdonságot jelöli, ahonnan adatkötéskor a megjelenített adatok származnak. A *DisplayIndex* a megjelenítés sorrendjét határozza meg, míg a *Name* az oszlop nevét adja meg, és ez fog megjelenni a fejlécben is, ha nem adjuk meg a *HeaderText* tulajdonság értékét.
Mielőtt továbbmegyünk készítsünk egy stringtömböt:

```
string[] sexList = new string[]
{
      "Férfi", "Nő"
};
```

Ezt fogjuk rákötni a *ComboBox* –ra. Ebben az esetben tulajdonképpen két kötés is lesz, hiszen egyszer megadjuk a vezérlő értékkészletét, másodszor pedig hozzákötjük a *DataTable* oszlopához is. Most jön a *DataGridViewComboBoxColumn*:

```
DataGridViewComboBoxColumn cbt = new DataGridViewComboBoxColumn();
cbt.DataPropertyName = "Sex";
cbt.DisplayIndex = 3;
cbt.Name = "Sex";
cbt.DataSource = sexList;
```

Gyakorlatilag készen vagyunk, semmi mást nem kell módosítanunk. A kész program így néz ki:



Mentéskor az adattábla is módosulni fog, erről könnyen meggyőződhetünk, ha kiíratjuk az adott sor értékeit (pl. egy *MessageBox* –xal).

54.2 DataView

A *DataView* osztállyal egy táblát több szemszögből nézhetjük, illetve rendezhetjük, szűrhetjük (hasonló a funkcionalitása mint az SQL *where* és *order by* parancsainak). Egy *DataView* jellemzően egy *DataTable* példányra „ül rá”. Vegyük a következő példát:

```
private void Form1_Load(object sender, EventArgs e)
{
   DataView view = new DataView(table);
   view.RowFilter = "Name = 'Reiter Istvan'";

   dataGridView1.DataSource = view;
}
```

A *RowFilter* tulajdonság egy kifejezést vár, amely megadja, hogy milyen feltétel alapján válogassa ki az oszlopokat. A *RowStateFilter* pedig a sorok állapotának megfelelően keres.
Látható, hogy a *DataView* köthető, így egy táblát más – más nézőpontból köthetünk rá vezérlőkre.

A *Sort* tulajdonsággal rendezhetjük is az adatokat:

```
private void Form1_Load(object sender, EventArgs e)
{
   DataView view = new DataView(table);
   view.Sort = "Name";
```

```
    dataGridView1.DataSource = view;
}
```

Ebben a példában a rendezés a Name oszlop alapján valósul meg.

54.3 DataSet

Egy *DataSet* példány *DataTable* és *DataRelation* objektumokból áll, vagyis táblákból és a köztük lévő relációkból. Egy *DataSet* tulajdonképpen megfelel egy adatbázisnak, amely a memóriában van. Hozzunk létre egy *DataSet* –et két táblával:

```
DataTable customers = new DataTable("Customers");
DataTable orders = new DataTable("Orders");
DataSet data = new DataSet("ShopDataSet");
```

A példaprogramban egy egyszerű ügyfél – rendelés szituációt fogunk modellezni. Készítsük el az oszlopokat:

```
DataColumn primKeyCust = new DataColumn("CID", typeof(int));
primKeyCust.AllowDBNull = false;
primKeyCust.AutoIncrement = true;
primKeyCust.AutoIncrementSeed = 0;
primKeyCust.AutoIncrementStep = 1;

customers.Columns.Add(primKeyCust);
customers.PrimaryKey = new DataColumn[]
{
    primKeyCust
};

customers.Columns.Add(new DataColumn("Name", typeof(string)));
customers.Columns.Add(new DataColumn("Address", typeof(string)));

DataColumn primKeyOrd = new DataColumn("OID", typeof(int));
primKeyOrd.AllowDBNull = false;
primKeyOrd.AutoIncrement = true;
primKeyOrd.AutoIncrementSeed = 0;
primKeyOrd.AutoIncrementStep = 1;

orders.Columns.Add(primKeyOrd);
orders.PrimaryKey = new DataColumn[]
{
    primKeyOrd
};

orders.Columns.Add(new DataColumn("CID", typeof(int)));
orders.Columns.Add(new DataColumn("OrderedThing", typeof(string)));

DataRow custRow1 = customers.NewRow();
custRow1["Name"] = "Nagy Sándor";
custRow1["Address"] = "Budapest, ...";

DataRow custRow2 = customers.NewRow();
custRow2["Name"] = "Kis Balázs";
```

```
custRow2["Address"] = "Eger, ...";

customers.Rows.Add(custRow1);
customers.Rows.Add(custRow2);

DataRow ordRow1 = orders.NewRow();
ordRow1["CID"] = 0;
ordRow1["OrderedThing"] = "Elektromos gyomorkaparó";

DataRow ordRow2 = orders.NewRow();
ordRow2["CID"] = 1;
ordRow2["OrderedThing"] = "Napelemes szemfenéktörlő";

orders.Rows.Add(ordRow1);
orders.Rows.Add(ordRow2);

shopData.Tables.Add(customers);
shopData.Tables.Add(orders);

shopData.AcceptChanges();
```

Maga a program elég egyértelmű semmi olyan nincs benne, amivel már ne találkoztunk volna. A *DataSet* –en meghívott *AcceptChanges* meghívja az egyes táblák metódusát, amelyek majd a tábla soraira is meghívják.
Kössük hozzá az egyik táblát egy *DataGridView* –hoz:

```
dataGridView1.DataSource = shopData.Tables[1];
```

54.3.1 Relációk táblák között

Amikor egy valamilyen adatbázisból beolvasunk adatokat egy *DataSet* –be, akkor a táblák közti relációk (amik az adatbázisban léteztek) elvesznek. Ebből kifolyólag képesnek kell lennünk reprodukálni ezeket a kapcsolatokat. Erre a feladatra a *DataRelation* osztály szolgál:

```
DataRelation relation = new DataRelation("CustAndOrd",
   shopData.Tables["Customers"].Columns["CID"],
shopData.Tables["Orders"].Columns["CID"]);

shopData.Relations.Add(relation);
```

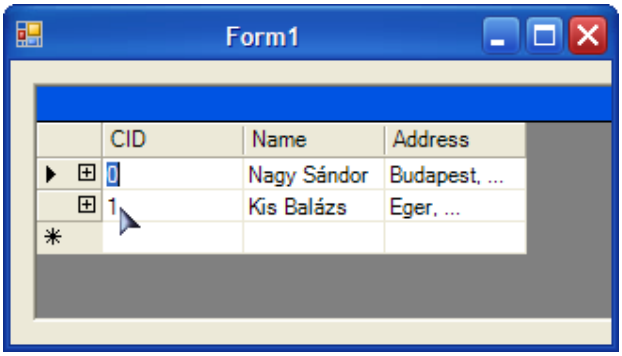
A *DataRelation* itt bemutatott konstruktora (van még öt) három paramétert vár: a reláció nevét, a szülőoszlopot, amelyre majd a harmadik paraméter a gyermekoszlop mutat (ez gyakorlatilag megfelel egy idegenkulcs relációnak). Mi most az egyes vásárlók rendeléseire vagyunk kíváncsiak.
A relációs objektumot hozzá is kell adnunk a *DataSet* –hez.

Természetes az igény, hogy az adatainkat a relációk alapján ábrázoljuk, tehát pl. meg tudjuk nézni, hogy ki mit rendelt. Erre a feladatra két megoldás is van: az első a „régi” *DataGrid* (a *DataGridView* elődje) vezérlő használata, ennek tulajdonképpen ez az egyetlen előnye az új vezérlővel szemben.

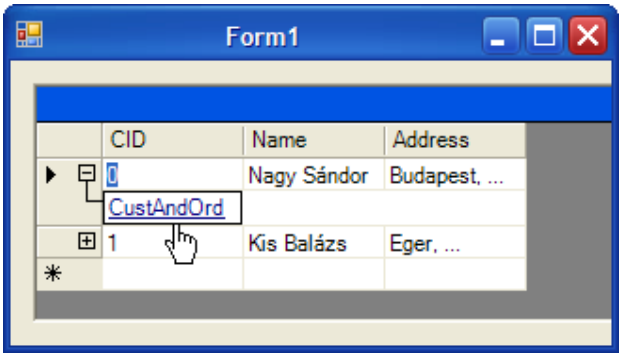
Ajunk hozzá a formhoz egy *DataGrid* –et (ha nincs ott a ToolBox –ban, akkor jobb klikk valamelyik szimpatikus fülön és Choose Items, majd keressük ki a vezérlőt). A form *Load* eseményében fogjuk rákötni az adatforrásokat:

```
private void Form1_Load(object sender, EventArgs e)
{
   dataGrid1.DataSource = shopData;
   dataGrid1.DataMember = "Customers";
}
```

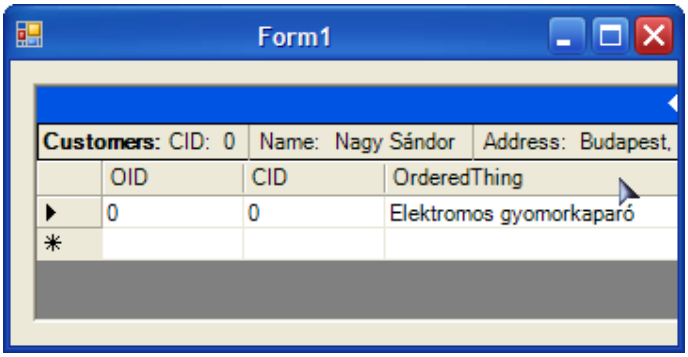
A *DataMember* tulajdonság azt mondja meg, hogy melyik táblát használjuk. Ezután az eredmény a következő lesz:



A sor elején lévő jelre kattintva megjelennek a hozzárendelt relációk:



Erre kattintva pedig megnézhetjük a másik tábla adatait:

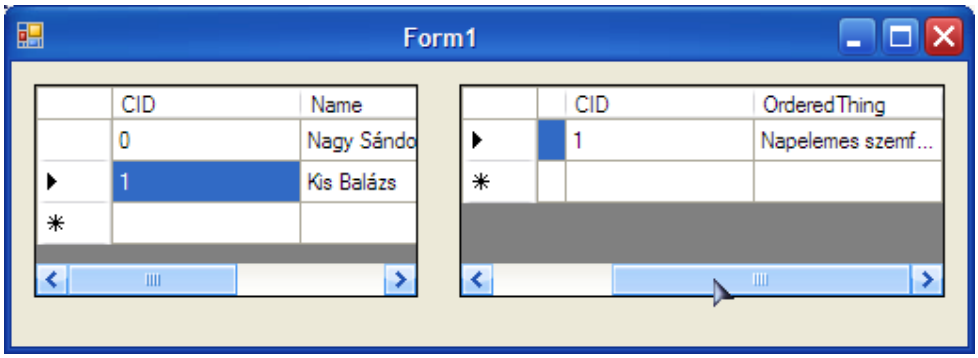


Ez volt az egyszerűbb megoldás. A *DataGridView* nem képes relációk megjelenítésére, ehelyett egy ún. *master-detail* megoldást fogunk létrehozni. Két *DataGridView* lesz a formon, a master –ben navigálva (ez a fenti példában a vásárlók táblája lesz) láthatjuk a detail –ben a hozzá tartozó adatokat (a rendelést).

Húzzuk rá a két vezérlőt a formra és nevezzük el őket *masterGrid* –nek és *detailGrid* –nek.
Ezenkívűl szükségünk lesz két *BindingSource* objektumra is, ez az osztály az adatkötések finomrahangolásában segít minket. Készítsük el őket is *masterBinding* és *detailBinding* néven. Az első *BindingSource* –höz a C*ustomers* táblát fogjuk kötni, a másodikhoz pedig a táblák közti relációt, így amikor az első griden a kiválasztott sor módosul, akkor a másikon megjelenik a hozzá rendelt adat:

```
private void Form1_Load(object sender, EventArgs e)
{
   masterBinding.DataSource = shopData;
   masterBinding.DataMember = "Customers";

   detailBinding.DataSource = masterBinding;
   detailBinding.DataMember = "CustAndOrd";

   masterGrid.DataSource = masterBinding;
   detailGrid.DataSource = detailBinding;
}
```

Az eredmény:



Részletesebb információkat talál a master-detail kapcsolatokról és a BindingSource osztályról az olvasó a következő oldalakon:

http://msdn.microsoft.com/en-us/library/y8c0cxey.aspx
http://msdn.microsoft.com/en-us/library/system.windows.forms.bindingsource.aspx

54.3.2 Típusos DataSet –ek

Eddig „normális” típustalan *DataSet* –eket használtunk, vagyis fordítási időben semmilyen ellenőrzés nem történik a típusok ellenőrzésére. Térjünk vissza a Persons táblához és helyezzük el azt egy *DataSet* –ben, ezután pedig írjuk a következőt:

```
personsData.Tables["Persons"].Rows[0]["Age"] = "ezegystring";
```

Ezzel a sorral a program ugyan lefordul, de futás közben kivétel generálódik (*ArgumentException*), hiszen stringet nem lehet implicit numerikus értékké alakítani (fordítva viszont működne).
Persze odafigyeléssel meg lehet előzni a bajt, de ha nem akarunk ezzel bajlódni, akkor használhatunk típusos *DataSet* –eket, amelyeket fordításkor ellenőrizhetünk.

A vélemények megoszlanak a típustalan és típusos megoldások közt, igazából egyik használatából sem származik különösebb előny vagy hátrány.

A Solution Explorerben kattintsunk jobb egérgombbal a projectre, majd válasszuk az Add/New Item menüpontokat és az ablakban válasszuk ki a DataSet –et. A példában a neve Persons.xsd lesz. A Visual Studio létrehozza a megfelelő állományokat, ezután pedig grafikus felületen tervezhetjük meg a *DataSet* elemeit. A ToolBox –ból hozzáadhatunk táblákat és relációkat is (meg mást is, de erről a következő fejezet szól majd). Készítsük is el a Persons DataSet –et egy táblával, amelynek legyen a neve PersonsTable. Az egyes oszlopok (amik most fizikailag sorok) tulajdonságait a Properties ablakban állíthatjuk.

Húzzunk most a formra egy *DataGridView* –ot, majd hozzuk létre a *DataSet* példányt:

```
Persons personsDataSet = new Persons();

Persons.PersonsTableRow row = personsDataSet.PersonsTable.NewPersonsTableRow();
row.Name = "Béla";
row.Age = 20;
row.Address = "Budapest";

personsDataSet.PersonsTable.Rows.Add(row);

personsDataSet.AcceptChanges();

dataGridView1.DataSource = personsDataSet.Tables[0];
```

Látható, hogy a tábla adatait tulajdonságokon keresztűl adhatjuk meg, így már fordításkor kiderülnek a rossz típus használatából fakadó hibák.
Ami még fontos, az az, hogy a táblák a *DataSet* példányosításakor azonnal létejönnek, nem kell kézzel elkészíteni illetve hozzáadni azokat.

55. Adatbázis adatainak lekérdezése és módosítása

Azt már tudjuk, hogyan kapcsolódjunk egy adatbázis szerverhez és az sem okozhat gondot, hogy a lekérdezett adatokkal dolgozzunk. Ami hiányzik, az az ún. *connected layer*, ami tulajdonképpen az adatok kezeléséért felelős.

Egy adatbázison kétféleképpen tudunk lekérdezést végrehajtani, vagy a klienstől küldjük a lekérdezést, vagy pedig a szerveren hívunk meg egy tárolt eljárást. Akárhogy is döntünk, mindkét esetben a *DbCommand* absztrakt osztály egy leszármazottját fogjuk használni, ez MSSQL szerver esetén az *SqlCommand* lesz.
Készítsünk egy Console Application –t (az egyszerűség kedvéért) és nyissunk egy kapcsolatot:

```
using (SqlConnection connection = new SqlConnection())
{
   connection.ConnectionString = @"Data Source=(local)\SQLEXPRESS;Initial
Catalog=Test;Integrated Security=True";

   connection.Open();
}
```

Az adatbázisunkban legyen egy *Persons* nevű tábla, amelynek egy *ID* és egy *Name* oszlopa van. Írjunk egy lekérdezést, amellyel ezeket le tudjuk kérdezni:

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = connection;
cmd.CommandText = "select ID, Name from Persons";
```

A C*onnection* tulajdonság az *SqlConnection* –re mutat, a *CommandText* pedig a lekérdezést fogja tartalmazni. Most küldjük is el a szerver felé a lekérdezést:

```
SqlDataReader reader = cmd.ExecuteReader();

while (reader.Read())
{
   Console.WriteLine("ID: {0}, Name: {1}", reader.GetInt32(0), reader.GetString(1));
}
```

Az *SqlDataReader* sorokat olvas ki az adatbázisból, méghozzá azokat, amelyeket az *ExecuteReader* metódus visszaad. Az *SqlCommand* utasításait háromféle módon hathatjuk végre: az *ExecuteReader* visszaadja az összes sort, ami a feltételnek megfelelő, Az *ExecuteScalar* csakis az első sort míg az *ExecuteNonQuery* csakis azoknak az oszlopoknak a számát, amelyekre a lekérdezés hatással volt (ahogy a nevében is benne van, ezt nem lekérdezésre, hanem törlésre, módosításra, stb. használjuk).
Most meg is kell jelenítenünk az adatokat, ehhez egy ciklust használunk, ami addig megy, amíg van mit olvasni. Az *SqlDataReader Get**** metódusai a megfelelő típusú adatokat adja vissza a paramétereként megadott oszlopból.

Most nézzük meg, hogy mit kell tennünk, ha tárolt eljárást akarunk használni. Tegyük fel, hogy a szerveren már van is egy tárolt eljárás *GetPersonById* néven, amely egy paramétert kap, a keresett személy azonosítóját:

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = connection;
cmd.CommandText = "GetPersonById";
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add("@PersonID", SqlDbType.Int).Value = 1;
```

55.1 Connected és Disconnected rétegek közti kapcsolat

Az előző részben már megismerkedtünk a kapcsolat nélküli réteggel. Ahhoz, hogy a kettőt össze tudjuk kapcsolni egy olyan osztályra van szükségünk, amely „hídat" képez köztük. Erre a *DataAdapter* osztályt fogjuk használni. Ez – ahogy azt már megszokhattuk – egy absztrakt osztály, amelynek leszármazottai az egyes adatbázis szerverek sajátosságaihoz specializálódnak. Mi most értelemszerűen az *SqlDataAdapert* osztály példányait fogjuk használni. Az adatbázis legyen ugyanaz, mint az előző fejezetben, de most hozzunk létre egy Windows Application –t, mivel a *DataGridView* vezérlőt fogjuk használni az adatok megjelenítésére.
Először egy típustalan *DataSet* –et használunk, utána a típusos társának használata is bemutatásra kerül.

```
SqlCommand selectCmd = new SqlCommand();
selectCmd.Connection = connection;
selectCmd.CommandText = "select ID, Name from Persons";

SqlDataAdapter adapter = new SqlDataAdapter();
adapter.SelectCommand = selectCmd;

adapter.Fill(personsDataSet);
```

Maga az adapter gyakorlatilag semmiről nem tud semmit, egyetlen dolga van, hogy közvetítsen. A *SelectCommand* tulajdonsága egy *SqlCommand* példányt vár, ez fog az adatok lekérdezésére szolgálni (ugyanígy van *UpdateCommand*, *DeleteCommand*, stb.). A *Fill* metódusa meghívja a *SelectCommand* utasítását és feltölti a paramétereként megadott *DataSet* –et (a háttérben tulajdonképpen az előző fejezetben bemutatott *DataReader* osztályok működnek). A *DataSet* mellett a tábla nevét is megadhatjuk, ha az már létezik, akkor az adatokat hozzáfűzi, ha még nem, akkor pedig létehozza:

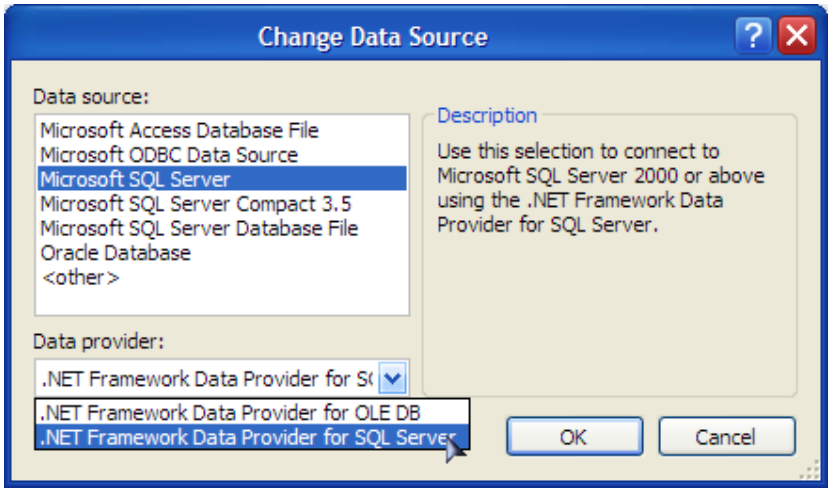
```
adapter.Fill(personsDataSet, "Persons");
```

Az adatok visszaírása az adatbázisba is nagyon egyszerű. Meg kell írnunk a szükséges törlés, módosítás SQL lekérdezéseket és ezeket *SqlCommand* formájában megadni az adapternek. Ezután az *Update* metódussal egyszerűen meghívjuk ezeket a lekérdezéseket:

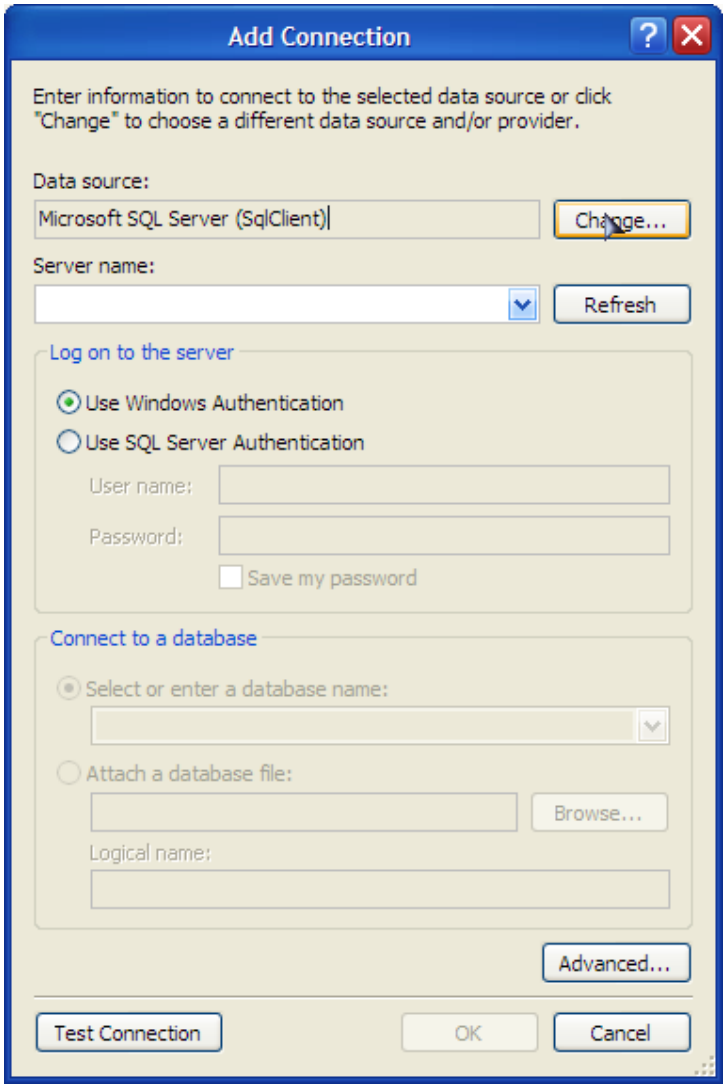
```
adapter.Update(personsDataSet, "Persons");
```

Most a *personsDataSet Persons* táblájának módosításait írtuk vissza.

Típusos *DataSet* –ek esetében még egyszerűbb lesz a dolgunk, hiszen grafikus felületen varázslók segítségével konfigurálhatjuk be az egészet. A projecthez adjunk hozzá egy *DataSet* –et, ezt s szokásos jobb klikk a projecten Add/New Item műveletekkel tudjuk megtenni. Ezután megjelenik a tervező nézet, amely egyelőre teljesen üres. Most szükségünk lesz a Server Explorer ablakra, ha alapértelmezetten nem látható, akkor a View menüben megtaláljuk (vagy használjuk a Ctrl+Alt+S billentyűkombinációt). Kattintsunk jobb egérgombbal a Data Connections elemen és válasszuk ki, hogy Add Connection. Ha korábban már dolgoztunk valamilyen adatbázisal, akkor rögtön az Add Connection ablak jelenik meg, ha a Data Source sorban nem a következő áll: Microsoft SQL Server (SqlClient), akkor kattintsunk a Change gombra, ekkor megjelenik az adatforrást kiválasztó ablak. Szintén ez jelenik meg akkor, ha még nem dolgoztunk adatbázissal:

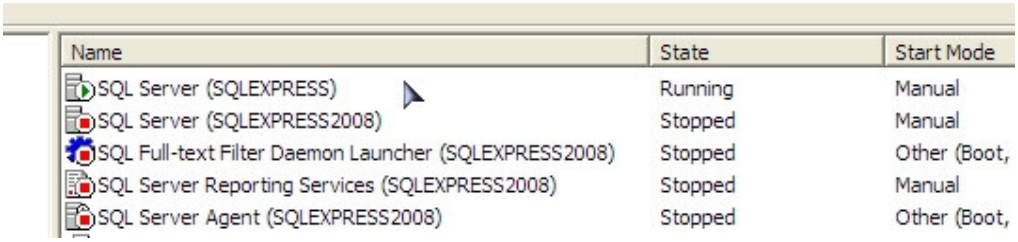


Válasszuk ki a Microsoft SQL Server –t, azon belül pedig a „.NET Framework Data Provider for SQL Server” –t. OK után visszatérünk az előző ablakhoz:



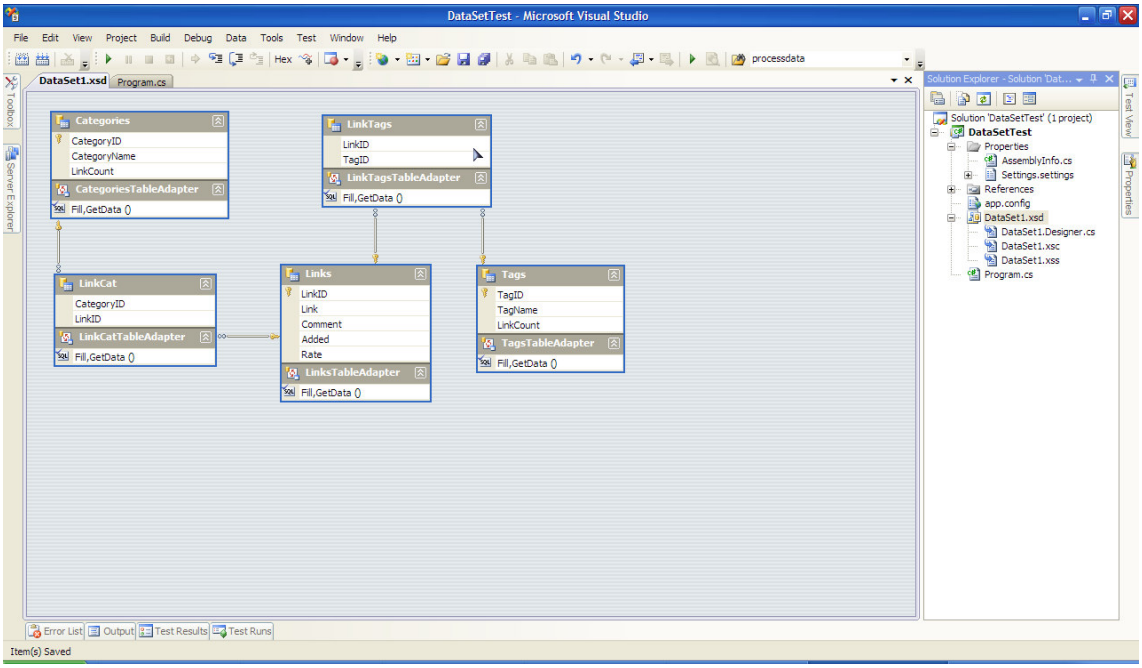
A Server Name sorba kell írnunk a számítógép azonosítóját és az SQL Server példányának nevét. Elöbbit megtudhatjuk, ha a parancssorba (Start/Futtatás cmd) beírjuk a hostname parancsot. Az SQL Server Express neve alapértelmezetten SQLEXPRESS lesz, ezt meg kellett adnunk telepítéskor (kivéve ha a Visual Studio telepítette, akkor alapértelmezés szerint ment). Könnyebben is hozzá tudunk jutni ezekhez az információkhoz, ha megnyitjuk az SQL Server Management Studio –t, ekkor a kapcsolódás ablakban megjelennek a szükséges információk:

Térjünk vissza a Visual Studio –hoz. Írjuk be a megszerzett adatokat, majd kattintsunk az alul lévő Test Connection gombra. Ha hibaüzenetet kapunk, akkor ellenőrizzük, hogy elindítottuk –e az SQL Servert. A Start menü programjai közül keressük ki az SQL Servert, azon belül pedig a Configuration Tools mappát. Itt találjuk az SQL Server Configuration Manager –t. Ha nincs elindított szerver (az ikon piros a példányok neve mellett), akkor jobb egérgombbal kattintsunk a szerverpéldányon és válasszuk ki, hogy Start.

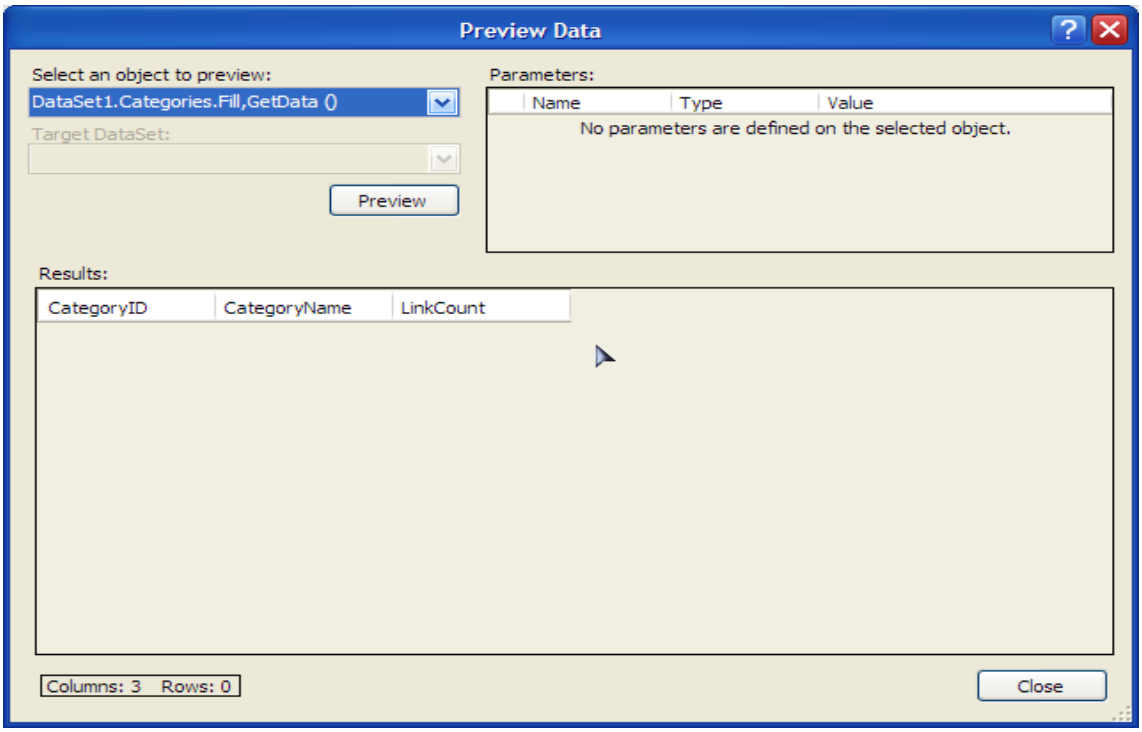


A képen a Configuration Manager látható, fölül zöld ikonnal és futó státusszal egy működő SQL Express 2005 látható, alul ínaktív 2008 –as társa.

Ismét teszteljük a kapcsolatot a Visual Studio –ban. Ha működik, akkor ki tudjuk választani a használni kívánt adatbázist. Nyomjuk meg az Ok gombot, ekkor az adatbázis megjelenik a Server Explorer –ben. Ezután ha használni akarjuk, akkor csak nyissuk le a fülét, automatikusan csatlakozik majd (ha van futó SQL Server). Tehát nyissuk le a fület és keressük ki a táblák közül azokat amelyeket használni akarunk. Húzzuk át ezeket a DataSet tervezőjébe. Ezután létrejönnek a táblák és automatikusan a hozzájuk tartozó relációk is:



Minden egyes táblához elkészül egy *TableAdapter* objektum is. A táblákat tesztelhetjük is, kattintsunk jobb egérgombbal a tervezőn és válasszuk a Preview Data menüpontot. Ebben az ablakban megtekinthetjük a táblák tartalmát az adapterek segítségével. Egyelőre csak annyit tudunk tenni, hogy minden oszlopot lekérdezünk, de ezt bővíthetjük is. Válasszuk ki valamelyik adaptert, kattintsunk rajta jobb egérgombbal és válasszuk ki a Preview Data menüpontot. Ekkor a megjelenő ablakban megnézhetjük az adott tábla tartalmát, illetve az adapteren létrehozott lekérdezések alapján mást is csinálhatunk:



Egy dataset nem csak egy adatbázisból tartalmazhat adatokat. A ToolBox –ból húzzunk egy TableAdapter –t a tervezőbe, ekkor megjelenik egy varázsló, amelynek megadhatjuk az új táblák helyét, illetve, hogy milyen műveleteket generáljon. Ugyanígy arra is van lehetőség, hogy adatbázistól függetlenül DataTable objektumokat adjunk a dataset –hez. Az egyes táblákat teszőleges kóddal is bővíthetjük, mivel a generált osztályok mind parciális osztályok, vagyis attól függetlenül kiegészíthetőek. Ehhez kattintsunk jobb gombbal valamelyik táblára és válasszuk, hogy View code.

A táblák generálása után a típusos dataset –ekkel való munka nagyon hasonló a hagyományosokéhoz, azzal a különbséggel, hogy mostantól IntelliSense támogatás és fordítási idejű típusellenőrzés is jár a csomaghoz.

56. XML

A .NET Framework erősen épít az XML –re, a típusos *DataSet* –ek, a LINQ To SQL, stb… is ezt a formátumot használja az adatsémák felépítésre.

56.1 XML file –ok kezelése

A szükséges osztályok a *System.Xml* névtérben vannak. A két legalapvetőbb ilyen osztály az *XmlReader* és az *XmlWriter*. Ezek absztrakt osztályok, amelyek segítségével villámgyorsan hatjhatóak végre a műveletek (persze van nekik számos specializált változatuk is, ezekről is lesz szó).

Elsőként nézzük meg, hogyan tudunk beolvasni egy XML filet. A megnyitáshoz az *XmlReader* egy statikus metódusát a *Create* –et fogjuk használni, ez egy stream –től kezdve egy szimpla filenévig mindent elfogad:

```
XmlReader reader = XmlReader.Create("test.xml");
```

Függetlenül attól, hogy az *XmlReader* egy absztrakt osztály tudjuk példányosítani. Ennek oka az, hogy ilyen esetekben valójában egy *XmlTextReaderImpl* típusú objektum jön létre, amely az *XmlReader* egy belső, *internal* elérésű leszármazottja (tehát közvetlenül nem tudnánk példányosítani).

Miután tehát megnyitottuk a filet, végig tudunk iterálni rajta (a példában egy *RichTextBox* vezérlőt használunk a megjelenítésre):

```
while (reader.Read())
{
   switch (reader.NodeType)
   {
      case XmlNodeType.Element:
         richTextBox1.AppendText("<" + reader.Name + ">");
         break;
      case XmlNodeType.EndElement:
         richTextBox1.AppendText("</" + reader.Name + ">");
         break;
      case XmlNodeType.Text:
         richTextBox1.AppendText(reader.Value);
         break;
      default:
         break;
   };
}
```

Az *XmlReader NodeType* tulajdonsága egy *XmlNodeType* felsorolás egy tagját adja vissza, a példában a legalapvetőbb típusokat vizsgáltuk és ezek függbényében írtuk ki a file tartalmát.

A következő példában használni fogjuk az *XmlWriter* osztály egy leszármazottját, az *XmlTextWriter* –t, ugyanis a filet kódból fogjuk létrehozni:

```
XmlTextWriter writer = new XmlTextWriter("newxml.xml", Encoding.UTF8);
writer.Formatting = Formatting.Indented;
```

A *Formatting* tulajdonság, az Xml file -októl megszokott hierachikus szerkezetet fogja megteremteni (ennek az értékét is beállíthatjuk).

```
writer.WriteStartDocument();
writer.WriteComment(DateTime.Now.ToString());
```

Elkezdjük az adatok feltöltését, és beszúrunk egy kommentet is. A file tartalma most a következő:

```
<?xml version="1.0" encoding="utf-8"?>
<!--2008.10.17. 20:00:59-->
```

A file személyek adatait fogja tartalmazni:

```
writer.WriteStartElement("PersonsList");

writer.WriteStartElement("Person");
writer.WriteElementString("Name", "Reiter Istvan");
writer.WriteElementString("Age", "22");
writer.WriteEndElement();
```

A *StartElement* egy új „tagcsoportot” kezd a paramétereként megadott névvel, a *WriteElementString* pedig feltölti azt értékekkel, kulcs érték párokkal.

A file:

```
<?xml version="1.0" encoding="utf-8"?>
<!--2008.10.17. 20:02:05-->
<PersonsList>
 <Person>
  <Name>Reiter Istvan</Name>
  <Age>22</Age>
 </Person>
</PersonsList>
```

Az egyes tagekhez attribútumokat is rendelhetünk:

```
writer.WriteAttributeString("Note", "List of persons");
```

Első paraméter az attribútum neve, utána pedig a hozzá rendelt érték jön. Ekkor a file így alakul:

```
<?xml version="1.0" encoding="utf-8"?>
<!--2008.10.17. 20:05:24-->
<PersonsList Note="List of persons">
 <Person>
  <Name>Reiter Istvan</Name>
  <Age>22</Age>
 </Person>
```

```
</PersonsList>
```

Az *XmlReader* rendelkezik néhány *MoveTo* előtaggal rendelkező metódussal, ezekkel a file egy olyan pontjára tudunk navigálni, amely megfelel egy feltételnek. Néhány példa:

```
bool l = reader.MoveToAttribute("Note");
```

Ez a metódus a praméthereként megadott attribútumra állítja a reader –t, és igaz értékkel tér vissza, ha létezik az attribútum. Ellenkező esetben a pozíció nem változik, és a visszatérési érték hamis lesz.

```
reader.MoveToContent();
```

Ez a metódus a legközelebbi olyan csomópontra ugrik, amely tartalmaz adatot is, ekkor az *XmlReade*r (vagy leszármazottainak) *Value* tulajdonsága ezt az értéket fogja tartalmazni.

A *MoveToElement* metódus visszalép arra a csomópontra, amely azt az attribútumot tartalmazza, amelyen a reder áll (értelemszerűen következik, hogy az előző kettő metódust gyakran használjuk együtt):

```
XmlReader reader = XmlReader.Create("newxml.xml");

while (reader.Read())
{
   if (reader.HasAttributes)
   {
      for (int i = 0; i < reader.AttributeCount; ++i)
      {
         reader.MoveToAttribute(i);
         richTextBox1.AppendText(reader.Name + " " + reader.Value);
      }

      reader.MoveToElement();
   }
}
```

A *HasAttribute* metódussal megtudakoljuk, hogy van –e a csomóponton attribútum, utána pedig az *AttributeCount* tulajdonság segítségével (amely azoknak a számát adja vissza) végigiterálunk rajtuk. Miután végeztünk visszatérünk a kiindulási pontra (hogy a *Read* metódus tudja végezni a dolgát).

Említést érdemelnek még a *MoveToFirstAttribute* és a MovetoNextAttribute metódusok, amelyek nevükhöz méltóan az első illetve a „következő” attributumra pozícionálnak. Módosítsuk egy kicsit az előző példát:

```
for (int i = 0; i < reader.AttributeCount; ++i)
{
   //reader.MoveToAttribute(i);
   reader.MoveToNextAttribute();
   richTextBox1.AppendText(reader.Name + " " + reader.Value);
```

```
}
```

Végül, de nem utolsósorban a *Skip* metódus maradt, amely átugorja egy csomópont gyermekeit és a következő azonos szinetn lévő csomópontra ugrik.

Most ismerkedjünk meg az *XmlReader* egy leszármazottjával az *XmlTextReader* –rel. Használata teljesen ugyanolyan, mint amit az *XmlReader* esetében már láttunk. Az objektum létrehozása a hagyományos úton zajlik (neki is megadhatunk pl. egy filestream –et):

```
XmlTextReader reader = new XmlTextReader("newxml.xml");
```

A példában az előzőleg létrehozott file –t fogjuk használni. Gyakorlatilag az *XmlTextReader* tulajdonságai megegyeznek annak az osztályéval, amelyet az *XmlReader* példányosításánál létrehoztunk.

56.2 XML DOM

Eddig az XML állományokat hagyományos file –ként kezeltük, ennél azonban van egy némileg kényelmesebb lehetőségünk is. Az XML DOM (Document Object Model) a file –okat a hierarchikus felépítésük szerint kezeli. Az osztály, amelyen keresztűl elérjük a DOM –ot az az *XmlDocument* lesz. Egy másik fontos osztály, amelyből egyébként maga az *XmlDocument* is származik, az az *XmlNode*. Egy XML file egyes csomópontjait egy-egy *XmlNode* objektum fogja jelképezni.
Ahogy azt már megszokhattuk az *XmlDocument* számos forrásból táplálkozhat, pl. stream vagy egyszerű filenév. A forrás betöltését az *XmlDocument Load* illetve *LoadXml* metódusaival végezzük:

```
XmlDocument xdoc = new XmlDocument();
xdoc.Load("test.xml");
```

Egy *XmlDocument* bejárható foreach ciklussal a *ChildNodes* tulajdonságon keresztűl, amely egy *XmlNodeList* objektumot ad vissza. Persze nem biztos, hogy léteznek gyermekei, ezt a *HasChildNodes* tulajdonsággal tudjuk ellenőrizni. A következő példában egy *XmlDocument* első szinten lévő gyermekeit járjuk be:

```
foreach (XmlNode node in xdoc.ChildNodes)
{
   richTextBox1.AppendText(node.Name);
}
```

Az *XmlDocument* a *ChildNodes* tulajdonságát az *XmlNode* –tól örökli (ugyanígy a *HasChildNodes* –t is), így az egyes *XmlNode* –okat bejárva a teljes „fát' megkaphatjuk.

Most nézzük meg, hogyan tudjuk manipulálni a file –t. Hozzunk létre kódból egy teljesen új dokumentumot:

```
XmlDocument xdoc = new XmlDocument();
```

```
XmlElement element = xdoc.CreateElement("Test");
XmlText text = xdoc.CreateTextNode("Hello XML DOM!");

XmlNode node = xdoc.AppendChild(element);
node.AppendChild(text);

xdoc.Save("domtest.xml");
```

Az *XmlText* és *XmlElement* osztályok is az *XmlNode* leszármazottai. A Save metódussal pedig el tudjuk menteni a dokumentumot, akár filenevet, akár egy stream –et megadva. A fenti kód eredménye a következő lesz:

```
<Test>Hello XML DOM!</Test>
```

A *CloneNode* metódussal már létező csúcsokat „klónozhatunk”:

```
XmlNode source = xdoc.CreateNode(XmlNodeType.Element, "test", "test");
XmlNode destination = source.CloneNode(false);
```

A metódus egyetlen paramétert vár, amely jelzi, hogy a csúcs gyermekeit is átmásoljuk –e.

Két másik metódusról is megemlékezünk, első a *RemoveChild*, amely egy létező csúcsot távolít el a csúcsok listájából, a másik pedig a *ReplaceChild*, amely felülír egy csúcsot.

Az *XmlDocument* eseményeket is szolgáltat, amelyek segítségével teljes körű felügyeletet nyerhetünk. Kezeljük le például azt az eseményt, amikor egy új csúcsot adunk hozzá:

```
XmlNodeChangedEventHandler handler = null;
handler = (sender, e) =>
   {
      MessageBox.Show(e.Node.Value);
   };
xdoc.NodeInserting += handler;
```

56.3 XML szerializáció

Mi is az a szérializálás? A legegyszerűbben egy példán keresztűl lehet megérteni a fogalmat, képzeljük el, hogy írtunk egy játékot, és a játékosok pontszámát szeretnénk eltárolni. Az elért pontszám mellé jön a játékos neve és a teljesítés ideje is. Ez elsőre semmi különös, hiszen simán kiírhatjuk az adatokat egy file –ba, majd visszaolvashatjuk onnan. Egy hátulütő mégis van, mégpedig az, hogy ez egy elég bonyolult feladat: figyelni kell a beolvasott adatok típusára, formátumára, stb…
Nem lenne egyszerűbb, ha a kiírt adatokat meg tudnánk feleltetni egy osztálynak? Ez megtehetjük, ha készítünk egy megfelelő osztályt, amelyet szérializálva kiírhatunk XML formátumba és onnan vissza is olvashatjuk (ezt deszérializálásnak hívják). Természetesen a szérializálás jelentősége ennél sokkal nagyobb és nemcsak XML segítségével tehetjük meg. Tulajdonképpen a szérializálás annyit tesz, hogy a

memóriabeli objektumainkat egy olyan szekvenciális formátumba konvertáljuk, amelyből vissza tudjuk alakítani az adatainkat.

Már említetük, hogy több lehetőségünk is van a szerializációra, ezek közül az XML a legáltalánosabb, hiszen ezt más környezetben is felhasználhatjuk.

Kezdjük egy egyszerű példával (hamarosan megvalósítjuk a pontszámos példát is), szérializáljunk egy hagyományos beépített objektumot:

```
FileStream fstream = new FileStream("serxml.xml", FileMode.Create);
XmlSerializer ser = new XmlSerializer(typeof(DateTime));
ser.Serialize(fstream, DateTime.Now);
```

Ezután létrejön egy XML file, benne a szérializált objektummal:

```
<?xml version="1.0"?>
<dateTime>2008-10-23T16:19:44.53125+02:00</dateTime>
```

Az *XmlSerializer* a *System.Xml.Serialization* névtérben található.
A deszérializálás hasonlóképpen működik:

```
FileStream fstream = new FileStream("serxml.xml", FileMode.Open);
XmlSerializer ser = new XmlSerializer(typeof(DateTime));
DateTime deser = (DateTime)ser.Deserialize(fstream);
```

Most már eleget tudunk ahhoz, hogy készítsünk egy szérializálható osztályt. Egy nagyon fontos dolgot kell megjegyeznünk, az osztálynak csakis a publikus tagjai szérializálhatóak a *private* vagy *protected* elérésűek automatikusan kimaradnak (emellett az osztálynak magának is publikus elérésünek kell lennie). Ezeken kívűl még szükség lesz egy alapértelmezett konstruktorra is (a deszérializáláshoz, hiszen ott még nem tudja, hogy milyen objektumról van szó).

```
public class ScoreObject
{
   public ScoreObject()
   {

   }

   private string playername;

   [XmlElement("PlayerName")]
   public string PlayerName
   {
      get { return playername; }
      set { playername = value; }
   }

   private int score;

   [XmlElement("Score")]
   public int Score
   {
```

```
        get { return score; }
        set { score = value; }
    }

    private DateTime date;

    [XmlElement("Date")]
    public DateTime Date
    {
        get { return date; }
        set { date = value; }
    }
}
```

Az egyes tulajdonságoknál beállítottuk, hogy az miként jelenjen meg a file –ban (sima element helyett lehet pl. attribútum is). Enélkül is lefordulna és működne, de így jobban kezelhető. Rendben, most szérializáljuk:

```
ScoreObject so = new ScoreObject();
so.PlayerName = "Player1";
so.Score = 1000;
so.Date = DateTime.Now;

FileStream fstream = new FileStream("scores.xml", FileMode.Create);
XmlSerializer ser = new XmlSerializer(typeof(ScoreObject));
ser.Serialize(fstream, so);
```

És az eredmény:

```
<?xml version="1.0"?>
<ScoreObject xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <PlayerName>Player1</PlayerName>
  <Score>1000</Score>
  <Date>2008-10-23T16:34:32.734375+02:00</Date>
</ScoreObject>
```

56.4 XML és a kapcsolat nélküli réteg

Az előző fejezetben szérializációval értük el, hogy eltároljuk az objektumainkat. Mi van akkor, ha az objektum jóval bonyolultabb, esetleg nem is egy objektumról van szó?
Ebben a fejezetben megtanuljuk, hogyan tudunk egy *DataTable*/*DataSe*t objektumot XML formátumba menteni. Készítsünk egy egyszerű táblát:

```
DataTable table = new DataTable("Persons");

DataColumn primcol = new DataColumn();
primcol.ColumnName = "ID";
primcol.DataType = typeof(int);
primcol.AutoIncrement = true;
primcol.AutoIncrementSeed = 0;
primcol.AutoIncrementStep = 1;
```

```
table.Columns.Add(primcol);
table.PrimaryKey = new DataColumn[]
{
   primcol
};

table.Columns.Add(new DataColumn("Name", typeof(string)));
table.Columns.Add(new DataColumn("Address", typeof(string)));

DataRow row1 = table.NewRow();
row1["Name"] = "Kovács János";
row1["Address"] = "Budapest, ...";
table.Rows.Add(row1);

DataRow row2 = table.NewRow();
row2["Name"] = "Nagy Ede";
row2["Address"] = "Baja, ...";
table.Rows.Add(row2);
```

Ezután a *WriteXml* metódus segítségével kiírhatjuk a tábla adatait:

```
table.WriteXml("persons.xml");
```

A file:

```
<?xml version="1.0" standalone="yes"?>
<DocumentElement>
 <Persons>
  <ID>0</ID>
  <Name>Kovács János</Name>
  <Address>Budapest, ...</Address>
 </Persons>
 <Persons>
  <ID>1</ID>
  <Name>Nagy Ede</Name>
  <Address>Baja, ...</Address>
 </Persons>
</DocumentElement>
```

Ahogy azt látjuk, ez a megoldás csak a tábla adatait írja ki. Ha szeretnénk visszaolvasni a táblát, akkor annak a szerkezetét, a sémáját is el kell tárolnunk:

```
table.WriteXml("persons.xml", XmlWriteMode.WriteSchema);
```

Az eredmény sokkal összetettebb lesz, hiszen az egyes oszlopok felépítését is kiírjuk:

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
 <xs:schema id="NewDataSet" xmlns=""
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-
microsoft-com:xml-msdata">
  <xs:element name="NewDataSet" msdata:IsDataSet="true"
msdata:MainDataTable="Persons" msdata:UseCurrentLocale="true">
```

```
    <xs:complexType>
     <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element name="Persons">
       <xs:complexType>
        <xs:sequence>
         <xs:element name="ID" msdata:AutoIncrement="true" type="xs:int" />
         <xs:element name="Name" type="xs:string" minOccurs="0" />
         <xs:element name="Address" type="xs:string" minOccurs="0" />
        </xs:sequence>
       </xs:complexType>
      </xs:element>
     </xs:choice>
    </xs:complexType>
    <xs:unique name="Constraint1" msdata:PrimaryKey="true">
     <xs:selector xpath=".//Persons" />
     <xs:field xpath="ID" />
    </xs:unique>
   </xs:element>
  </xs:schema>
  <Persons>
   <ID>0</ID>
   <Name>Kovács János</Name>
   <Address>Budapest, ...</Address>
  </Persons>
  <Persons>
   <ID>1</ID>
   <Name>Nagy Ede</Name>
   <Address>Baja, ...</Address>
  </Persons>
</NewDataSet>
```

Ezt a file –t már tudjuk használni:

```
DataTable backupTable = new DataTable();
backupTable.ReadXml("persons.xml");
```

Amennyiben a tábla adataira nincs szükség, csak a sémára, akkor használjuk a *WriteXmlSchema*/*ReadXmlSchema* metódusokat.

DataSet –ek esetében gyakorlatilag pontosan ugyanúgy járunk el, ahogy azt a táblák esetében már láttuk.

Figyelem! Még nincs vége: a következő frissítés 2009. Április 12. –én várható.