

feladat nagy részét (szerencsére) átháríthatjuk olyan fejlesztésszűkökre, mint a Visual Studio 2008, a Microsoft Expression Blend vagy bármely más, külső gyártótól származó termékre. Ha az eszközök elkészítik az alapvető markupt, ezt követően manuálisan végezhetjük el a XAML-definíciók szűk-séges apró simításait.

Noha az eszközök a XAML terjedelmes részt „tálcán kínálják”, érdemes megismerkednünk a XAML-szintaxis alapvető működésével, és meg kell érte-nünk, hogy a rendszer hogyan alakítja át a markupt szabályos „NET-szerel-vényekké”. Tanulmányozzuk a XAML-t működés közben! A következő példánk-ban kizárólag néhány *.xaml fájli felhasználásával készítettünk WPF-alkalmazást.

A MainWindow definíciója XAML-ben

Az első Window-leszármazott osztályunkat (mainwindow) a C#-kódban olyan osztálytípusként definiáltuk, amely kibővíti a system.windows.window alaposz-tályt. Az osztály egyetlen button típust foglal magában, amely ha a gombra kattintunk, meghívja a regisztrált eseménykezelőt. Ugyanezt a window típust XAML nyelvben definiálva (tetelezzük fel, hogy a markupt a mainwindow.xaml nevű fájlban definiáltuk) hasonló eredményt kapunk:

```
<!-- Íme, a window definíció -->
<Window x:Class="SimpleXamlApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="My Xaml App" Height="200" Width="300"
  WindowStartupLocation="CenterScreen">
  <!-- Az ablak tartalmának beállítása -->
  <Button Width="133" Height="24" Name="btnExitApp"
    Click="btnExitApp_Clicked">
    Exit Application
  </Button>
</Window>

<!-- A gomb kattintási eseménykezelőjének megvalósítása -->
<x:Code>
<iCDATA[
private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
{
    // Az aktuális alkalmazás leírójának megszerzése és az
    // alkalmazás leállítása.
    Application.Current.Shutdown();
}
]>
</x:Code>
</Window>
```

```

<!-- A main() metódus hívnyzik! A StartUpuri attribútum a metódus
funktcióját tekintve azonos megjelölője
-->
<Application x:Class="SimpleXamlApp.MyApp"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri="MainWindow.xaml">
</Application>

```

talimat az új MyApp.xaml fájlban:

Bizonyal meghatározhatjuk markkupban. Vizsgáljuk meg a következő tar-

konstruktor. Ennek fényében az alkalmazásobjektum definícióját is minden

osztály markkupját elkészíthetjük, ha az osztály támogatja az alapértelmezett

Emlékezzünk rá, hogy a XAML segítségével bármely nem absztrakt .NET-

Alkalmazásobjektum definíciója XAML-ben

lálható (végereedményben ezt a fájlt fogjuk elkészíteni).

több WPF-alkalmazásban az "igazi kód" egy kapcsolódó C#-kódfájlban ta-

a felhasználói felület markkupját és a programozási logika kapcsolatát. A leg-

műveletet elkülöníti egy helyen, a belső kód nem választja szét egyértelműen

ajánlott a <code> elemben. Bár ez az "egyetlen fájl megközelítés" az összes

Fontos rámutatunk, a szerkesztési funkcionális implementáció nem

közvetlenül értelmezni az adatokat.

csomagoltuk, és így megakadályozzuk, hogy az XML-elemek megpróbálják

példánkban nem feltétlenül lenne szükség) a kódot a CDATA hatókörbe

lőit és egyéb metódusait. Biztonsági intézkedésként (noha erre az aktuális

teszt, hogy közvetlenül a *.xaml fájlban készítsük el az osztály eseménykeze-

A XAML-fájl utolsó említésre méltó pontja a <code> elem, amely lehetővé

tük a metódust, amelyhez fordulni kell, ha a káttintási esemény bekövetkezik.

külső fömerek beállításán túl kezeljük a gomb káttintási eseményét: kijelöl-

implicit módon beállítjuk az ablak content tulajdonságát. A változónév és a

buton példány megjelenését és funkcióját leíró markkupot, amely segítségével

Figyeljük meg, hogy az ablak definíciójának hatókörében elkészítettük a

donságainak közvetlen leképezését.

amelyek – mint látnuk – a system.windows.window típus hasonló nevű tulaj-

tuk a title, a height, a width és a windowstartaruplocation attribútumokat,

foglalkozunk részletesen). A nyitó <window> elem hatókörében meghatároz-

winx/2006/xaml definíciója (az XML-névterekkel a fejezet későbbi részében

ribútumot a XAML-központú XML-névter, a http://schemas.microsoft.com/

gítisével definíciója a származtatott típus nevét. Az x prefix jelzi, hogy az at-

Először is vegyük észre, hogy a <window> gyökerelem a class attribútum se-

Egyetérthetünk abban, hogy az `Application` osztályból származó `C#`-osztálytípus és a XAML leírása közötti leképezés nem annyira áttekinthető, mint a `mainwindow` XAML-definíciója esetén. A példában nyomat sem találjuk a `main()` metódusnak. Mivel a .NET végrehajtható fájlnak rendelkeznie kell programbelső ponttal, helyesen gondoljuk, hogy ez részben a `startup` tulajdonság alapján, a fordítás során generálódik. A hozzárendelt `*.xaml` fájlszámra vonatkozó határozható meg, hogy az alkalmazás indítása során melyik `mainwindow`-ból származó osztályt kell létrehozni.

A `main()` metódus a fordítás során automatikusan elkészül, és a `<code>` elem segítségével – igény szerint – a következőképpen készíthetjük el az `exit` eseménykezelőt (vegyük észre, hogy többé nem statikus metódussal van dolgunk, mivel a metódus példányszintű tagga alakul a `myapp` osztályban):

```
<Application x:Class="SimpleXamlApp.MyApp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="mainwindow.xaml" Exit="AppExit">
    <x:Code>
    <![CDATA[
        private void AppExit(object sender, EventArgs e)
        {
            MessageBox.Show("App has exited");
        }
    ]]>
    </x:Code>
</Application>
```

XAML-fájlok feldolgozása az `msbuild.exe` segítségével

Készen állunk rá, hogy a markuptól érvényes .NET-szerelvényé alakítsuk. Az átalakítás során nem alkalmazhatjuk a `C#`-fordítót, és nem használhatunk reakciófájlt. Pillanatnyilag a `C#`-fordító nem alkalmas a XAML-markup feldolgozására. Azonban az `msbuild.exe` parancssori segédprogram tudja, hogyan kell a XAML-t `C#`-kóddá alakítani, és a kódot menetekben lefordítani, ha felhívjuk a figyelmeztető `*.targets` fájlokra.

Az `msbuild.exe` eszköz lehetővé teszi, hogy összetett fordítószkripteket építsünk (micsoda megjelölés!) XML segítségével. Az XML-definíciók egyik érdekessége, hogy ugyanolyan formátummal rendelkeznek, mint a Visual Studio `*.csproj` fájlok.

Ennek köszönhetően egyetlen fájlt definiálhatunk az automatizált parancssori fordítók számára, valamint a Visual Studio 2008 projekte számára. Vizsgáljuk meg a következő, lenyegre törő `SimpleXamlApp.csproj` fájlt:

```
<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <RootNamespace>SimpleXamlApp</RootNamespace>
    <AssemblyName>SimpleXamlApp</AssemblyName>
    <OutputType>WinExe</OutputType>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="WindowsBase" />
    <Reference Include="PresentationCore" />
    <Reference Include="PresentationFramework" />
  </ItemGroup>
  <ItemGroup>
    <Page Include="MainWindow.xaml" />
    <ApplicatonDefinition Include="MyApp.xaml" />
  </ItemGroup>
  <Import Project="$(MSBuildPath)\Microsoft.CSharp.targets" />
  <Import Project="$(MSBuildPath)\Microsoft.WinFX.targets" />
</Project>
```

A példában a `<PropertyGroup>` elem segítségével meghatározzuk a fordítás néhány alapvető tulajdonságát, például a gyökérnévteret, az előállított szelvény nevét és a kimenet típusát (a `csc.exe` program /target:winexe beállítáसानak megfelelője).

Az első `<ItemGroup>` azoknak a külső szerezvényeknek a gyűjteményét határozza meg, amelyre hivatkoznunk kell. Láthatjuk, hogy ezek azok a kulcsfontosságú WPF-szerelvények, amelyeket a fejezet korábbi részében megvizsgáltunk. A második `<ItemGroup>` sokkal érdekesebb. Vegyük észre, hogy az `<ApplicatonDefinition>` elem `Include` attribútumát ahhoz az `*.xaml` fájlhoz rendeltük, amely az alkalmazásobjektumunkat definiálja. A `<Page>` `Include` elemével listázhatjuk a többi `*.xaml` fájlt, amelyek az alkalmazásobjektum által feldolgozott ablakokat (és oldalakat, amelyeket a XAML-böngészőalkalmazások készítése során gyakran használunk) definiálnak.

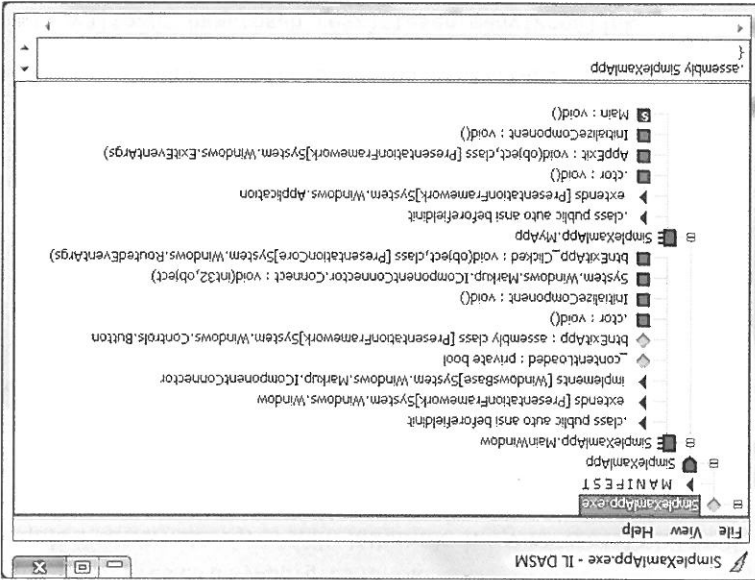
A `*.csproj` fájl „varázssereje” az utolsó `<Import>` elemekben rejlik. Vegyük észre, hogy a fordító szkriptünk két `*.targets` fájlra hivatkozik, amelyek mindegyike további utasításokat foglal magában a fordítási folyamatra vonatkozóan. A `Microsoft.WinFX.targets` fájl foglalja magában a XAML-definíciók megfelelő C#-kódfájllra alakításához nélkülözhetetlen fordítási beállításokat, a `Microsoft.CSharp.targets` pedig a C#-fordítóval folytatott kommunikációhoz szükséges adatokat tartalmazza.

Megjegyzés A könyvben nem vizsgáljuk meg részletesen az msbuild.exe segédprogramot. Ha szeretnék jobban megismerni az eszközt, a .NET Framework SDK dokumentációban keressük az „MSBuild” témakört.

Most feldolgozhatjuk a `SimpleXamlApp.csproj` fájlnkat az `msbuild.exe` programmal:

```
msbuild SimpleXamlApp.csproj
```

A fordítás befejezését követően a szerelvényünket a generált `\bin\Debug` könyvtárban találjuk. Ekkor a várt módon elindíthatjuk WPF-alkalmazásunkat. Egyéltérhetünk abban, hogy néhány sornyi markkup megírása az érve-nyes .NET-szerelvények generálásának elég bizarr módja. Ha a biztonság kedvéért megnyitjuk az `ilasm.exe` programmal a `SimpleXamlApp.exe` fájlt, láthatjuk, hogy a XAML-kódunk (valamilyen módon) átalakult végrehajtható alkalmazássá (lásd a 28.6. ábrát).



28.6. ábra: Markkup átalakítása .NET-szerelvénné? Érdekes...

Markup átalakítása .NET-szerelvénnyé

Annak érdekében, hogy megértsük, a markup hogyan alakult át .NET-szerelvénnyé, kicsit mélyebbre kell ásunk az msbuild.exe folyamatba, és meg kell vizsgálnunk néhány, a fordító által generált fájlt, beleértve a szerelvénnyé fordításidőben begyűjtött fontos bináris erőforrást.

XAML leképezése C#-kódra

Mint említettük, az MSBuild szkriptekben meghatározott *.targetst fájlok több utasítást definiálnak, amelyek alapján a rendszer a XAML-elemeket a fordításhoz C#-kódra alakítja. Amikor az msbuild.exe feldolgozza a *.csproj fájlnkat, két *.g.cs fájlt készít (a "g" *automatikusan generált* fájlt jelöl), amelyeket az \obj\Debug könyvtárba mentett. A *.xaml fájljaink neve alapján a kérdéses C#-fájlok a MainWindow.g.cs és a MyApp.g.cs fájlok.

Ha megnyitjuk a MainWindow.g.cs fájlt, láthatjuk, hogy az osztály kibővíti a Window alaposztályt, és elvárásainknak megfelelően tartalmazza a btnexit-App_Click() metódust. Az osztály egy System.Windows.Controls.Button típusú tagváltozót is definiál. Furcsa módon *nem* találunk olyan kódot, amely a Button vagy a Window típus tulajdonságainak (Height, Width, Title stb.) beállításáról gondoskodik. A rejtélyre hamarosan fény derül.

Végül vegyük észre, hogy az osztály definiál egy bool típusú privát tagváltozót (-content Loaded néven), amelyről a XAML-markup közvetlenül nem tesz említést. Íme, a generált MainWindow típus definíciójának egy része:

```
public partial class MainWindow :
    System.Windows.Window,
    System.Windows.Markup.IComponentConnector
{
    internal System.Windows.Controls.Button btnexitApp;

    // Ezt a tagváltozót hamarosan részletesen megvizsgáljuk.
    private bool _contentLoaded;

    private void btnexitApp_Click(object sender, RoutedEventArgs e)
    {
        // Az aktuális alkalmazás kezelőjének megszerzése,
        // és az alkalmazás leállítása.
        Application.Current.Shutdown();
    }
    ...
}
```

A window-ból származtatott osztály explicit módon megvalósítja a `system.windows.markup.nvtreeconnector` WPFI-terjeszt. Az interfész egyetlen metódust definiál, a `connect` metódust, amelyet az eredeti `mainwindow.xaml` fájlban meghatározott eseményre történő feliratkozást valósítja meg:

```
void System.Windows.Markup.IComponentConnector.Connect(
    int connectionId, object target)
{
    switch (connectionId)
    {
        case 1:
            this.btnExitApp = ((System.Windows.Controls.Button)(target));
            this.btnExitApp.Click += new
                System.Windows.RoutedEventHandler(
                    this.btnExitApp_Clicked);
            return;
        }
        this._contentLoaded = true;
    }
}
```

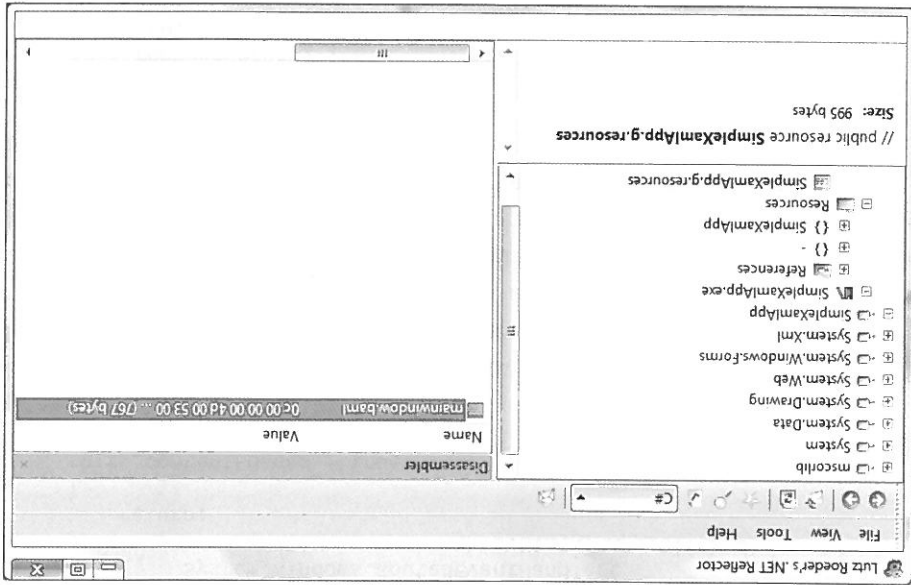
Végl a `mainwindow` osztály megvalósítja az `InitializeComponent` metódust. Ez a metódus feloldja a szerelevényben a beágyazott erőforrás helyét az eredeti `*.xaml` fájl neve alapján. Ha az erőforrást a rendszer megtalálja, az `Application.LoadComponent` meghívásával betölti az aktuális alkalmazásobjektumba. Utoljára pedig a (korábban említett) privát logikai tagváltozót állítjuk igaz értékre, és így biztosítjuk, hogy a rendszer a kért erőforrást az alkalmazás eleje során pontosan egyszer töltse be:

```
public void InitializeComponent()
{
    if (!_contentLoaded)
    {
        return;
    }
    _contentLoaded = true;
    System.Uri resourceLocator = new
        System.Uri("/SimpleXamlApp;component/mainwindow.xaml",
            System.UriKind.RelativeOrAbsolute);
    System.Windows.Application.LoadComponent(this, resourceLocator);
}
```

Ezen a ponton felmerül a kérdés: pontosan mi is az a beágyazott erőforrás?

A BAML szerepe

Amikor az `msbuild.exe` feldolgozza a `*csproj` fájlnkat, `*.baml` kiterjesztéssel létrehozott egy fájlt, amely a kezdeti `mainwindow.xaml` fájlt alapszinten kapta a nevet. Mint azt a nevből kitálalhatjuk, a BAML (Binary Application Markup Language – bináris alkalmazás jelölőnyelv) a XAML bináris megjelenítése. Ezt a `*.baml` fájlt a rendszer (a generált `*.g.resources` fájl segítségével) erőforrásként beágyazza a lefordított szerelvénybe. A BAML révén a WPF-szerelvények (sokkal tömörebb formátumban) magukban foglalhatják a teljes XAML-definíciójukat. Ezt magunk is ellenőrizhetjük, ha a `reflector.exe` programmal a 28.7. ábrán látható módon megnyitjuk a szerelvényünket.



28.7. ábra: A beágyazott `*.baml` erőforrás megtekintése Lutz Roeder .NET Reflector programjával

Az `Application.LoadComponent()` hívása beolvassa a beágyazott BAML-erőforrást, és feltölti a definiált objektumok hierarchiáját a megfelelő állapotaikkal (például az ablak `Height` és `Width` tulajdonságaival). Valójában, ha a Visual Studio alkalmazásban megnyitjuk a `*.baml` vagy a `*.g.resources` fájlt, láthatjuk az eredeti XAML-attribútumok nyomait. A 28.8. ábra példaként a `StartupLocation.CenterScreen` tulajdonságot emeli ki.

```

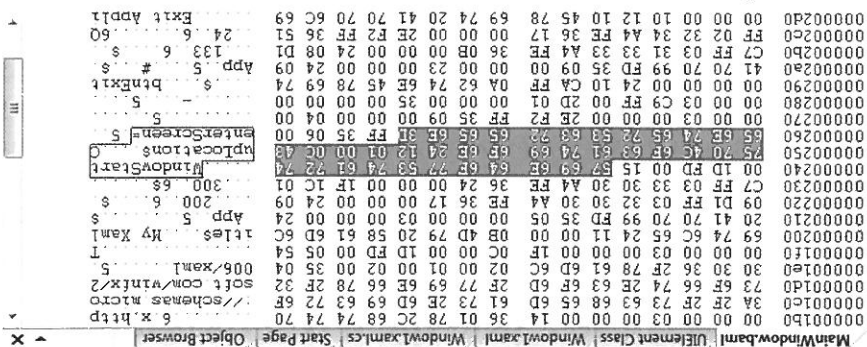
    }
    }
    }
    app.Run();
    app.InitializeComponent();
    SimpleXamlApp MyApp = new SimpleXamlApp.MyApp();
    public static void Main() {
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [System.STAThreadAttribute()]
    }

    new System.Windows.EventDispatcher(this, AppExit);
    this.StartupUri = new System.Uri("MainWindow.xaml",
        System.UriKind.Relative);

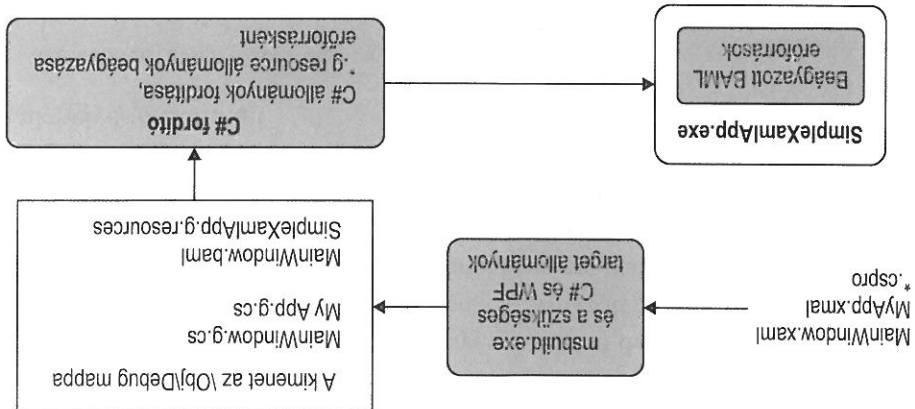
    public void InitializeComponent() {
        this.Exit +=
        new System.Windows.EventDispatcher(this, AppExit);
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public void InitializeComponent() {
            this.Exit +=
            new System.Windows.EventDispatcher(this, AppExit);
            MessageBox.Show("App has exited");
        }
        void AppExit(object sender, EventArgs e)
        {
            MessageBox.Show("App has exited");
        }
    }
    public partial class MyApp : System.Windows.Application
    {
        namespace SimpleXamlApp
    }

```

Az automatikusan generált kód rejtvényének utolsó darabja a MyApp.g.cs fájlban bujlik meg. Láthatjuk az Application-leszármazott osztályunkat a megfelelő main() belépési pont módszerrel. A módszer megvalósítása az Application-leszármazott típuson hívja az InitializeComponent() metódust, amely csereben beállítja a startupuri tulajdonságot, és így lehetővé teszi az objektumok számára, hogy a bináris XAML-definíció alapján meghatározzák tulajdonságaik helyes beállításait.

28.8. ábra: *line a BAML!*

Ejha! Összefoglalva, egy teljes .NET-szerelvényt hoztunk létre kizárólag három XML-dokumentum felhasználásával (a dokumentumok egyikét az msbuild.exe segédprogram használta). Mint láttuk, az msbuild.exe a *.target-fájlokból definiált segédbeállítások segítségével dolgozza fel a XAML-fájlokat (és generálja az *.bam1 fájlt) a fordítási folyamat számára. Ezeket a részletek a háttérben hatódtak végre. A 28.9. ábra bemutatja a *.xaml fájlok fordításiidejű feldolgozásának átfogó képét.



28.9. ábra: A XAML-ból szerelvény fordításiidejű folyamat

Fontos rámutatnunk, hogy amikor a fordító a megfelelő C#-kód és a bináris erőforrás előállításához feldolgozza az összes *.xaml1 fájlnkat, azokra gyakorlatilag a továbbiakban semmi szükség (és soha nem kell őket a végrehajt-ható fájllal együtt szállítani). A fejezet végén látjuk majd, hogy a *.xaml1 fájlt programozott módon végrehajtott olvasásával módunkban áll dinamikusan létrehozni window objektumot. Ebben az esetben a fizikai *.xaml1 fájlt nem szükséges az alkalmazással együtt szállítani.

Forráskód A SimpleXamlApp projektet a forráskódkönyvtár 28. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A kapcsolatok elkülönítése mögöttes kódfájlokkal

A kapcsolatok elkülönítése mögöttes kódfájlokkal

Mielőtt elkezdnenk beleásni magunkat a XAML részleteibe, meg kell vizsgálnunk az alap programozási modell még egy aspektusát: a kapcsolatok elkülönítését. Emlékezzünk rá, hogy a WPF fő célja, hogy elkülönítse a felhasználói felület tartalmát a programozási logikától, és az aktuális példák erről nem gondoskodtak.

Ahelyett, hogy közvetlenül a XAML <code>elem hatókörébe ágyazzanánk az eseménykezelőinket (és az egyedi metódusokat), szerencsésőbb különálló C#-fájlt definiálni a megvalósítási logika meghatározásához, és így a *.xaml fájlok kizárólag a felhasználói felület markupját foglalják magukban. Vizsgáljuk meg a következő mögöttes `mainwindow.xaml` cs kódfájlt (megjegyezés szerint a C# mögöttes kódfájlt neve `*.xaml.cs` formátumú):

```
// MainWindow.xaml.cs
using System;
using System.Windows;
using System.Windows.Controls;

namespace SimpleExamApp
{
    public partial class MainWindow : Window
    {
        // Jegyezzük meg! A metódust a
        //generated MainWindow.g.cs fájl definiálja.
        InitializeComponent();
    }

    private void btnNextApp_Clicked(object sender,
        RoutedEventArgs e)
    {
        // Az aktuális alkalmazás kezelőjének megszerzése,
        // és az alkalmazás leállítása.
        Application.Current.Shutdown();
    }
}
```

```

<Project DefaultTargets="Build" xmlns=
"http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <RootNamespace>SimpleXamlApp</RootNamespace>
    <AssemblyName>SimpleXamlApp</AssemblyName>
    <OutputType>WinExe</OutputType>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="WindowsBase" />
    <Reference Include="PresentationCore" />
    <Reference Include="PresentationFramework" />
  </ItemGroup>
  <ItemGroup>
    <ApplicationDefinition Include="MyApp.xaml" />
    <Compile Include="MyApp.xaml.cs" />
    <Page Include="MainWindow.xaml" />
  </ItemGroup>

```

Mielőtt az msbuild.exe segítségével lefordítanánk a fájljainkat, módosítanunk kell a *.csproj fájljunkat. A (félkövér betűvel szedett) <compile> elemek segítségével biztosítanunk kell, hogy a rendszer az új C#-fájlokat is feldolgozza a fordítás folyamat során:

```

// MyApp.xaml.cs
using System;
using System.Windows;
using System.Windows.Controls;

namespace SimpleXamlApp
{
    public partial class MyApp : Application
    {
        private void AppExit(object sender, EventArgs e)
        {
            MessageBox.Show("App has exited");
        }
    }
}

```

BAML-erőforrás betöltéséhez és feldolgozásához meg kell adnia a metódust. Mivel az események nagy része a MyApp.g.cs fájlban zajlik, a MyApp.xaml.cs fájlban található kód alig több valamivel a következőknél:

Ígény szerint mögöttes kódját is készíthetünk az Application-leszármazott BAAML-erőforrás betöltéséhez és feldolgozásához meghívja a metódust.

MainWindow.g.cs fájlban definiáltuk, az ablakunk konstruktora a beágyazott osztálydefiniációval fűszül össze. Mivel az InitializeComponent() metódust a amely a rendszer a *.g.cs fájlban található ugyanolyan típusú részleges példában részleges osztályt definiáltunk (az eseménykezelő logika számára).

```
</ItemGroup>
<Import Project="$(MSBuildPath)\Microsoft.CSharp.targets" />
<Import Project="$(MSBuildPath)\Microsoft.WinFX.targets" />
</Project>
```

Ha átadjuk a fordítószkriptünk az msbuild.exe segédprogramnak, eredményként ugyanazt a végrehajtható szerelvényt kapjuk. Fejlesztési szempontból azonban a programozási logikából (C#) a leírás (XAML) leírt részletét állítottuk elő. Mivel ez a WPF-fejlesztés ajánlott módja, örömmel vehetjük tudomásul, hogy a Visual Studio 2008 eszközzel létrehozott WPF-alkalmazások mindig az előbbiekből bementatott mögöttes kód-modellel használják.

Forráskód A CodeBehindXamlApp projektet a forráskódkönyvtár 28. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xiv. oldalát.

A XAML-szintaxis

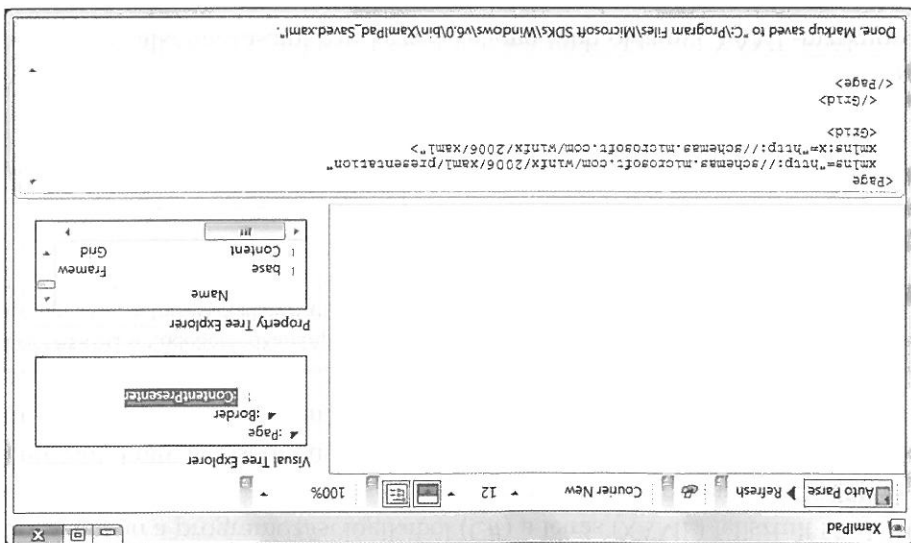
Mint azt a fejezet korábbi részében említettük, rendkívül kicsi az esélye annak, hogy WPF-alkalmazásainkban kézzel kellene több oldalt nyitni XAML-markupot megírniunk, hiszen ezt a feladatot a megfelelő eszközök (Visual Studio 2008, Microsoft Expression Blend stb.) elvégzik helyettünk. Azonban minél többet megtanulunk a jól formázott *.xaml fájlok szintaxisáról, annál felkészültebbek leszünk az automatikusan generált markup beállítására és módosítására, valamint melyek ismereteket szerzünk magáról a WPF-ről. Most pedig ássunk a XAML-alapszintaxis melyére (a következő fejezetekben – ahol szükség lesz rá – további XAML-szintaxis példákkal találkozunk).

XAML-kísérletek a XamlPad segítségével

A XAML tanulmányozása során biztosan létrehozunk valamilyen tartalmat, és az eredményt gyorsan szeretnénk látni. Ehhez a Microsoft az SDK részeként a xamlpad.exe segédprogramot bocsátja rendelkezésünkre.

Megjegyzés Érdekes módon sem a .NET Framework 3.5 SDK, sem a Visual Studio 2008 nem tartalmazza a xamlpad.exe programot. A Windows SDK letöltésével juthatunk hozzá ehhez az eszközhöz, a fejezet legutolsó példájában C# segítségével elkészítjük a xamlpad.exe lecsupa-sztított verzióját.

Ha letöltöttük a Windows SDK-t, hogy hozzájussunk a xamlpad.exe programhoz, az eszközt a Start > All Programs > Microsoft Windows SDK > Tools menüelem segítségével indíthatjuk. A 28.10. ábra a xamlpad.exe kezdeti képernyőjét mutatja, amelyben aktiváltuk a Visual Tree Explorer beállítását (az eszköztár megfelelő gombjával).



28.10. ábra: A XamlPad a XAML-markup valós idejű megjelenítését biztosítja

A XamlPad segítségével az ablak alján található panelben szerkeszthetjük a XAML-markupot, és a felső részben megtekinthetjük a kimenetet. Amikor első alkalommal indítjuk az eszközt, egy üres <Page> deklarációt találunk, amelyet arra használunk, hogy egy XBAP-alkalmazás markuját tartalmazza:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- Ide írhatjuk a XAML-markupot! -->
  </Grid>
</Page>
```

Noha a XamlPad nem teszi lehetővé, hogy a XamlPad nézet ablakában közvetlenül megtekinthetjük egy <Window> elem markuját, a <Page> elemet <Window> elemre módosíthatjuk, és az F5 funkciógombbal egy különálló ablakban megtekinthetjük a tartalmát. Ne feledjük továbbá, hogy a <Page> és a <Window> elem markujja megegyezik!

Megjegyzés A XamlPad nem teszi lehetővé, hogy olyan markupt készítsünk, amely a kód fordítását vonja maga után. Ez magában foglalja a `Class` attribútum definícióját (kódfájl meghatározásához), a `<Code>` elemek használatát, illetve bármely XAML-kulcsszó alkalmazását, amely a forráskód fordítását vonja maga után (mint például a `FileModifier` vagy a `ClassModifier`). Bármilyen erre irányuló próbálkozás markuphibához vezet.

A markupt készítése közben a XamlPad programban feltűnik az IntelliSense hiánya. Ha azonban a Show Visual Tree gombra kattintunk, megjelenik az a felhasználói felület, amely a Visual Studio Properties ablakát utánozza, és segít elképzelni a XAML-markup struktúráját. Sajnálatos módon a Visual Tree ablak segítségével (pillanatnyilag) nem módosíthatjuk magát a XAML-t; csak a XAML-markup írásvédelet megtekintését teszi lehetővé.

Azt se feledjük, hogy a XamlPad segítségével jelenleg nem menthetjük a *.xaml fájlokat; a markupt automatikusan a XamlPad_saved.xaml fájlba menti, és az eszközkövetkező betöltése során megjeleníti (azonban a markupt igény szerint bemásolhatjuk az aktuális alkalmazásba).

XAML-névterek és -kulcsszavak

Mint azt a fejezet korábbi példáiban láttuk, a WPF-központú XAML-fájl egyökeleme (például a `<Window>`, a `<Page>` vagy az `<Application>` tag) rendszert két XML-névterre hivatkozik:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    </Grid>
  </Page>
```

Az első XML-névter, a `http://schemas.microsoft.com/winfx/2006/xaml/presentation` WPF-központú névtereket képez le, és az aktuális *.xaml fájl rendelkezésére bocsátja azokat (`System.Windows.Controls`, `System.Windows.Data`, `System.Windows.Ink`, `System.Windows.Media`, `System.Windows.Navigation` stb.). Ez az egy-többes leképezés gyakorlatilag hardkódolva van a WPF-szerelvényekben (`WindowsBase.dll`, `PresentationCore.dll` és `PresentationFramework.dll`) a szerelvény szintű [`XmlnsDefinition`] attribútum révén. Ha ezeket a WPF-szerelvényeket a `reflector.exe` programmal megnyitjuk, közvetlen betekintést nyerhetünk a leképezésekbe (lásd a 28.11. ábrát).

A második XML-névter, a `http://schemas.microsoft.com/winfx/2006/xaml/XAML-specifikus` kulcsszavakat, valamint a `system.windows.markup` névterben található típusok részhalmozát foglalja magában. Egy helyesen formázott XML-dokumentumnak definiálnia kell egy gyökércsereletemet, amely elsődlegesen névterként egyetlen XML-névteret jelöl ki, amely típkiküszarban a leggyakrabban használt elemeket tartalmazó névter. Ha a gyökércserelemben további másodlagos névtereket kell belefoglalni (mint azt itt láttuk), a névtereket (a lehetőséges névűtközések feloldásának érdekében) egyedi prefix alkatmazásával kell definiálnunk. Megegyezés szerint a prefix egysszerűen „x”, azonban bármilyen egyedi token használhatunk, például a `xaml:specifikus` token is:

Ha a fájázat korábbi részében létrehozott myApp XAML-definíciót módosítani szeretnénk, mert ezt a részletesebb XML-névtér prefixet szeretnénk al-

Az XML-nevter prefixek definíciójának nyilvánvaló hátránya, hogy a XML-specifikus szótlétezés megnevezése nem mindig kell, ha a XML-fájlnak a kérdéses névterben definiált típusok egyike kell hivatkozni. Például a `http://schemas.microsoft.com/winfx/2006/xaml/` egyike a Code XAML-kulcszó, amely – mint láttuk – lehetővé teszi, hogy XAML-dokumentumba ágyazzuk a C#-forráskódot. A másik XAML-kulcszó a `class`, amely segítségével megne-

```
<Application xamlSpec:cstuf:Class="SimpleXamlApp"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:xamlSpec:cstuf="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri="MainWindow.xaml" Exit="AppExit"
"http://schemas.microsoft.com/winfx/2006/xaml"
xamlSpec:cstuf:Code>
<CDATA[
private void AppExit(object sender, EventArgs e)
{
    MessageBox.Show("App has exited");
}
}
]>
</xamlSpec:cstuf:Code>
</Application>
```

Mivel a xamlSpec:cstuf több billentyűleütést követel, maradjunk az x-nél. Mindenesetre a class és a code kulcsszavakon túl az http://schemas.microsoft.com/winfx/2006/xaml XML-névter további XAML-kulcsszavakat biztosít rendeltetésünkre (és a system.windows.markup névter elemeit is). A 28.9. táblázat ezen alapvető kulcsszavakat mutatja be.

XAML-kulcsszó Jelentés

Array	.NET-tímbtípust képvisel a XAML-ben.
ClassModifier	Lehetővé teszi a class kulcsszóval jelölt (belső vagy nyilvános) osztály láthatóságának meghatározását.
DynamicResource	Lehetővé teszi, hogy olyan WPF-erőforrásra hivatkozzunk, amelynek változásait ellenőrizni kell.
FieldModifier	Lehetővé teszi, hogy meghatározzuk egy típus tag (belső, nyilvános, privát, védett) láthatóságát a gyökér bármely megnevezett részlemez (például a <Window> elemen belül egy <Button> elem) számára. „Megnevezett elemet” a NameXAML kulcsszóval definiálunk.
Key	Lehetővé teszi, hogy kulcs értéket határozzunk meg egy XAML-elemhez – amelyet egy szótár elemben helyezünk el.
Name	Lehetővé teszi, hogy meghatározzuk adott XAML-elem generál C#-névét.
Null	Nullreferenciát jelöl.
Static	Lehetővé teszi, hogy egy típus statikus tagjára hivatkozzunk.
StaticResource	Lehetővé teszi, hogy olyan WPF-erőforrásra hivatkozzunk, amelynek változásait nem kell ellenőrizni.

XAML-kulcsszó Jelentés

Type	A C# typeof operátor XAML-megfelelője (a megadott név alapján System.Type osztályt hoz létre).
TypeArguments	Lehetővé teszi, hogy az elemet generikus típusként hozzunk létre a meghatározott típusparaméterrel (például a List<int> elemet szemben a List<bool> elemmel).

28.9. táblázat: XAML-kulcsszavak

Több említett kulcsszót látnunk majd működés közben. Egy egyszerű példán keresztül vizsgáljuk meg a következő XAML <window> definíciót, amely a ClassModifier és a FieldModifier, valamint a Name és a Class Kulcsszavakat alkalmazza (emlékezzünk rá, hogy a xamlpad.exe nem teszi lehetővé, hogy olyan XAML-kulcsszót használjunk, amelynek alkalmazása a kód fordítását vonja maga után, mint például a code, a FieldModifier vagy a ClassModifier):

```
<!-- Ez az osztály most belső osztály lesz.
Kódfájl használata esetén a részleges osztályt is belső
osztályként kell definiálni! -->
<window x:class="MyWPFApp.MainWindow" x:classmodifier="internal"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
<!-- Ez a gomb a *.g.cs fájlban nyilvános lesz -->
<button x:name="myButton" x:fieldmodifier="public">
    OK
</button>
</window>
```

Alapértelmezés szerint a C#/XAML típusdefiníciók nyilvánosak, míg a tagok alapértelmezés szerint belsők. A XAML-definíciónk alapján azonban az eredményként kapott, automatikusan generált fájl belső osztálytípust tartalmaz egy nyilvános Button típussal:

```
internal partial class MainWindow : System.Windows.Window,
    System.Windows.Markup.IComponentConnector
{
    public System.Windows.Controls.Button myButton;
    ...
}
```


Ezen a ponton a content tulajdonság implicit vagy explicit módon elvégzett beállítása pusztán személyes preferencia kérdésének tűnhet. A történet igazal-
masabbá válik, ha elgondolkodunk azon, hogy a XAML segítségével hogyan
alakítanánk a button tartalmát objektumra egyetlen egyszerű sztring helyett
(grafikus rendelés, scrollbar, vagy TextBox stb.). Mint a fejezet korábbi ré-
szében már említettük, a megoldás a tulajdonság-lem szintaxis alkalmazása.

```
<Button x:Name="myButton"
        Height="50" Width="100" Content="OK">
</Button>
```

A *tulajdonság-lem* szintaxis lehetővé teszi, hogy összetett objektumokat rendel-
jünk egy tulajdonsághoz. Íme a „görgőtősav a gombban” eset XAML-leírása,
amely tulajdonság-lem szintaxis segítségével állítja be a content tulajdonságot:

```
<Button x:Name="myButton" Height="100" Width="100">
<Button.Content>
<Scrollbar Height="50" Width="20"/>
</Button.Content>
</Button>
```

Vegyünk észre, hogy ebben az esetben a <Button.Content> beágyazott elemben
definiáltuk a scrollbar típust. A tulajdonság-lem szintaxis mindig lebont-
hatjuk <TypeName.TulajdonságNév> mintára; jelen esetben nyilván a <Button> a
típus, a content pedig a tulajdonság. A 28.12. ábra a xamlpad.exe programban
látható kimenetet mutatja.

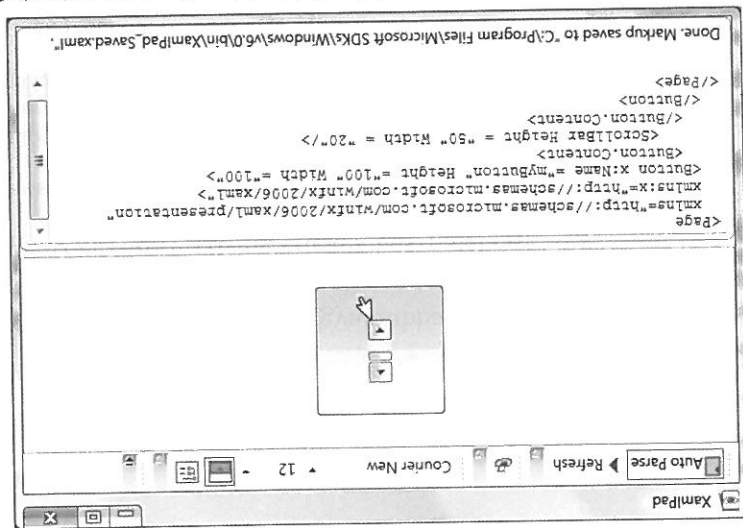
Emlékezzünk rá, hogy a contentControl-ból származtatott osztály gyer-
meklemével a rendszer automatikusan beállítja a content tulajdonságot, tehát
a következő definíció is helyes:

```
<Button x:Name="myButton" Height="100" Width="100">
<Scrollbar Height="50" Width="20"/>
</Button>
```

A tulajdonság-lem szintaxis nem korlátozódik a content tulajdonság beállítá-
sára. Ezt a XAML-szintaxist bármikor alkalmazhatjuk, ha típus tulajdonságnál
összetett objektumot kell beállítanunk. Vizsgáljuk meg például a button típus
background tulajdonságát.

A tulajdonságot a WPF API-ban található bármely Brush típuson beállíthatjuk. Ha egyszerű esetben típusot szerelnénk használni, ehhez csak a következő markus szükséges, mivel a Brush-leszármazott típus (például Background) való tulajdonságokhoz rendelt sztringérték automatikusan eset típusa alakul:

```
<!-- Itt a "Green" Brushes.Green típusra képeződik le -->
<Button x:Name="myButton" Height="100" Width="100"
        Background="Green">
</Button>
<Button.Content>
<ScrollBar Height="50" Width="20"/>
</Button.Content>
</Button>
```



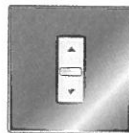
28.12. ábra: A tulajdonságlelem szintaxis lehetővé teszi, hogy összetett tulajdonságokat rendeljünk a tulajdonságokhoz

Ha kifinomultabb esetben (például LinearGradientBrush esetén) van szükségünk, a név/érték szintaxis nem lesz elegendő. Figyelembe véve, hogy a LinearGradientBrush teljes osztálytípus, a tulajdonságlelem szintaxis segítségével kell a típusnak a kezdeti értékeket átadnunk:

```
<Button x:Name="myButton" Height="100" Width="100">
<Button.Content>
<ScrollBar Height="50" Width="20"/>
</Button.Content>
</Button>
<Button.Background>
<LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
<GradientStop Color="Blue" Offset="0"/>
<GradientStop Color="Yellow" Offset="0.25"/>
</LinearGradientBrush>
```

```
<gradientstop color="green" offset="0.75" />
<gradientstop color="red" offset="0.50" />
</lineargradientbrush>
</Button.Background>
</Button>
```

Pillanatnyilag ne foglalkozzunk a `LinearGradientBrush` típus konfigurálásával (a 30. fejezet a WPF grafikus renderelési szolgáltatásait vizsgálja). Egyetlen csak figyeljünk meg, hogy a tulajdonságlelem szintaxis segítségével hataroztuk meg a `Button` típus `content` és `background` tulajdonságait. A 28.13. ábrán ezen különleges gomb renderelése látható.



28.13. ábra: Szokványos gomb típus

Noha a tulajdonságlelem szintaxisal leggyakrabban összetett objektumokat (például a `LinearGradientBrush` objektumot) rendelünk tulajdonság értékekhez, megengedett az egyszerű sztring értékek használata is:

```
<Button x:Name="myButton" height="100" width="100">
  <Button.Content>
    <ScrollBar height="50" width="20"/>
  </Button.Content>
  <Button.Background>
    pink
  </Button.Background>
</Button>
```

Ebben az esetben nem nyertünk semmit, inkább feleslegesen bonyolítottuk a folyamatot, miközben csak a következőket kellett volna betennünk:

```
<Button x:Name="myButton" height="100" width="100"
  background="pink">
  <Button.Content>
    <ScrollBar height="50" width="20" />
  </Button.Content>
</Button>
```

A XAML csatolt tulajdonságai

A tulajdonságlelem szintaxison kívül a XAML definiál egy olyan szintaxist, amellyel *csatolt tulajdonságokat* határozhatunk meg. A csatolt tulajdonságok sokoldalúak, és egyik céljuk, hogy lehetővé tegyék a különböző gyermek-elemek számára egy olyan tulajdonság egyedi értékeinek meghatározását, amelyet valójában a szülőelemben definiáltunk. A csatolt tulajdonság-szintaxis alkalmazásának legelterjedtebb módja a felhasználói felület-elemek elhelyezése a WPF-paneltípusok (grid, dockpanel stb.) valamelyikén. A következő fejezetben részletesen megvizsgáljuk ezeket a paneleket. Egyelőre tekintsük meg a csatolt tulajdonság-szintaxis következő példáját:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <DockPanel LastChildIsTrue="true">
    <!-- Elemek dokkolása a panelhez csatolt tulajdonságok
        használatával -->
    <label DockPanel.Dock="Top" Name="lblInstruction"
      FontSize="15">Enter car information</label>
    <label DockPanel.Dock="Left" Name="lblMake">Make</label>
    <label DockPanel.Dock="Right" Name="lblColor">Color</label>
    <label DockPanel.Dock="Bottom"
      Name="lblPetName">Pet Name</label>
    <Button Name="btnOK">OK</Button>
  </DockPanel>
</Page>
```

A példában dockpanel típust definiáltunk, amely négy, a tárolóban dokkolt label típust foglal magában. Vegyük észre, hogy a csatolt tulajdonság szintaxisa <szülőtípus.szülotulajdonság> (például dockpanel.dock). A button típus nem határoz meg dokkolási területet; el foglajja a dockpanel fennmaradó területét, mivel a nyitó <dockpanel> definícióban a lastchildIsTrue tulajdonság van beállítva. A csatolt tulajdonságokkal kapcsolatban néhány dologgal tisztában kell lennünk. Nem univerzális szintaxissal állunk szemben, amelyet bármely szülő bármely elemén alkalmazhatunk. A következő XAML-definíció például nem elemezhető anélkül, hogy hibát ne kapnánk:

```
<!-- A Height tulajdonság beállítására Button objektumon csatolt
tulajdonsággal -->
<Button x:Name="myButton" Width="100"
  Height="100">
  <ScrollBar Button.Height="100" Height="50" Width="20"/>
</Button.Content>
```

```
<Button.Background>
    pink
</Button.Background>
</Button>
```

A csatolt tulajdonság valójában a WPF-specifikus fogalom, a *függetlenség tulajdonság* speciális formáját képviseli. Röviden, a függetlenség tulajdonságok lehetővé teszik, hogy egyező értéket több bemenet alapján számítsuk ki. A függetlenség tulajdonságoknak, és emellett fogva a csatolt tulajdonságoknak is „regisztrálniuk” kell, hogy mely tulajdonságokat mely objektumok állíthatják be (ezt az előzőleg bemutatott `scrollbar/button` példában nem tettük meg).

A WPF a háttérben több technológia – például az adatkötés, a stílusok és témák és az animációs szolgáltatások – használatára esetén alkalmazza a függetlenség tulajdonság-mechanizmust. A függetlenség tulajdonság megvalósításával önirányító ellenőrzést, alapértelmezett értékeket és visszahívási mechanizmust biztosíthatunk, és lehetővé teszi, hogy futásidejű információk alapján állítsuk be a tulajdonságok értékét.

A függetlenség tulajdonságokkal kapcsolatos furcsaság, hogy a függetlenség tulajdonság értékeknek *beállítása* pontosan úgy néz ki, mint egy „normál”. NET-tulajdonság beállítása. Ezért az esetek többségében nem vesszük észre, hogy beállítottuk egy függetlenség tulajdonság értékét.

Azonban a függetlenség tulajdonságok megvalósítása a szintalak mögött való-
jában teljesen eltérő. A legtöbb WPF-alkalmazásunkban *nem* kell egyedi függetl-
ségi tulajdonságokat készítenünk. Ez csak abban az esetben válhat gyakori fel-
adattá, ha egyedi WPF-vezérlőelemeket kell készítenünk. Ez a XAML megjele-
nésnek köszönhetően szintén nem számít mindennapos tevékenységnek.
A 29. fejezetben részletesen megvizsgáljuk a függetlenség tulajdonságokat.

XAML-típusátalakítók

Gyakorlati szempontból, amikor értéket rendelünk az attribútumokhoz (pél-
dái `background = "pink"`), vagy a nyitó és a záró elemek hatókörében impli-
cit módon beállítjuk a tartalmat (példái `<button>ok</button>`), feltehetően nem
várjuk, hogy az értékek sztringadatok. Ha azonban ezt jobban átgondoljuk, rájö-
hetünk, hogy ez nem minden esetben igaz. Gondoljunk például a `button` ti-
pus `background` tulajdonságának definíciójára (amelyet a `control` összeszálytól
örököltünk):

```
// A System.Windows.Controls.Control.Background tulajdonság.  
public Brush Background  
{  
    ...  
}
```

Mint látjuk, a tulajdonság a `system string` helyett a `brush` típust csomagolja be! Felmerül a kérdés, hogy (ebben az esetben) a „pink” értéket mi alakítja `solidcolorbrush` objektummá, amely a `rozaszaszinek` megfelelő RGB értékekkel rendelkezik? Íme, egy másik példa, vizsgáljuk meg egy meghatározott méreteekkel rendelkező `biborszínű` `ellipszis` `XAML`-definícióját:

```
<Ellipse F111 = "purple" width = "100.5" height = "87.4">  
</Ellipse>
```

Ha megvizsgálunk az `ellipse` típus `width` és `height` tulajdonságait, láthatunk, hogy `prototípusuk` szerint `string`ek helyett `double` típusú adatokkal dolgoznak.

A színtalak mögött a `XAML`-elemzők különböző *típusátalakítók* segítségével alakítják a `string`adatokat a megfelelő `mögöttes objektumma`. Ha például a „pink” értéket olyan tulajdonsághoz rendeljük, amely `prototípusa` szerint `ecset` típusokkal dolgozik, a rendszer a `colorconverter` és a `brushconverter` típusokat alkalmazza. Több *átalakító* létezik: a `sizeconverter` (az előző `ellipse` típus `width` és `height` tulajdonságainak beállítására), a `rectconverter`, a `vectorconverter` és így tovább.

A nevetől függetlenül az összes `típusátalakító` a `system.componentmodel.typeconverter` `összesítőből` származik. Ez a típus több `virtuális metódust` definiál, például a `canconvertto()`, a `convertto()`, a `canconvertfrom()` és a `convertfrom()` `metódusokat`, amelyeket `sármaztatott típusban` `felüldefiniálhatunk`, és így `gondoskodhatunk` a `mögöttes fordítástól`.

Tulajdonképpen nem kell tudnunk, hogy *melyik* `típusátalakító` képezi le a `XAML-sztringadatainkat` a megfelelő `mögöttes objektumra`. Legalábbis elég megérténünk, hogy az *átalakítókat* a rendszer a `háttérben` a `XAML-definíciók` `egyszersítésére` használja.

Megjegyzés Noha maga a `XAML` viszonylag új technológia, a `típusátalakítók` koncepciója a .NET platform megjelenése óta létezik. A színtalak mögött a `Windows Forms` és a `GDI+` különböző `átalakítókat` használ. Például a `GDI+ system.drawing` névter a `FontConverter` típusátalakítót definiálja, amely a „Wingdings” sztringet a `Wingdings` betűkészletet alkalmazó `Font` objektumra képezi le.

A XAML markuppóvitményei

A típusátalakítók érdekes szerkezetek abból a szempontból, hogy nincs fizikai bizonyítéka a XAML szintén velük folytatott együttműködésnek. Ehelyett a típusátalakítók egy rendszer a szintaksa mögött a *.xaml fájlfeldolgozás során alkalmazza. Ezzel szemben a XAML támogatja a *markuppóvitményeket* is. A típusátalakítókhoz hasonlóan a markuppóvitmények lehetővé teszik, hogy egy egyszerű címkézési értéket futásidőben objektummá alakítsunk. A különbség azonban az, hogy a markuppóvitmények nagyon speciális XAML-szintaxissal rendelkeznek.

Mivel a típusátalakítók és a markuppóvitmények ugyanazt a célt szolgálják, felmerülhet a kérdés, hogy miért készíthetünk típusdefiniciókat két különböző módszerrel. A markuppóvitmények a típusátalakítókhoz hasonlóan működést biztosítanak, és lehetővé teszik, hogy a XAML nyelvet letisztult módon, új funkcionálisitással bővítsük.

A markuppóvitmények segítségével egy másik típuson a statikus tulajdonság visszatérési értékekhez rendelhetjük egy tulajdonság értékét, címkézéssel adathalmazt deklarálhatunk, vagy típusinformációkat gyűjthetünk. Valójában, a XAML-kulcsszavak egy részhalma (például az `Array`, a `Null`, a `Static` és a `Type`) ábrázolható markuppóvitményekkel a rendszer belsőleg olyan osztályként ábrázolja, amely a `MarkupExtension` osztályból származik (megegyezés szerint a `MarkupExtension` osztályból származó osztályok nevében az `extension` előtagot használják).

Vizsgáljuk meg működés közben a markuppóvitmények Tételvezérlő fel, hogy a `Label` típusok készíthetők beállítjuk a `content` tulajdonságot, és a `system.Environment` statikus tagjaival így jelentünk meg információkat arról a számtalról, amelyen az alkalmazásunkat futtatjuk. Így a teljes markuppó, magyarázattal:

```
<Page
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:corlib="clr-namespace:System;assembly=mscorlib">
  <StackPanel>
    <Label Content="{x:Static corlib:Environment.MachineName}" />
    <Label Content="{x:Static corlib:Environment.OSVersion}" />
    <Label Content="{x:Static corlib:Environment.ProcessorCount}" />
  </StackPanel>
</Page>
```


Vegyük észre, hogy a <Page> definíció új XML-névter deklarációval rendelkezik, amelyet a corlib névtérprefixummal láttunk el (a prefixum neve bármely más XML-névter prefixumához hasonlóan tetszőleges). Az XML-névterhez rendelt érték azonban egyedi, mivel egy clr-namespaces nevé regisztrált token (amely lehetővé teszi, hogy a típus definíciót tartalmazó „NET-névterre mutassunk), valamint egy assembly nevé másik token (amely a névteret magában foglaló szerelvény „barátságos” nevet képviseli) használatunk.

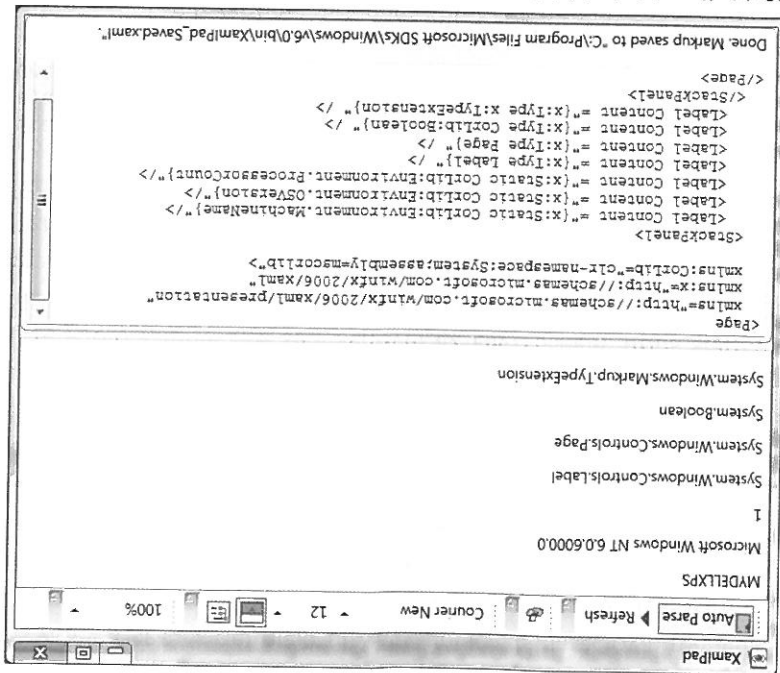
Az alkalmazott XML-névterekben figyeljük meg, hogy az egyes label típusok a static jelölő kiterjesztés segítségével az Environment típus statikus tagját indítják. Mint lájuk, a markkupbővíítőket mindig kapcsos zárójel fogja közre. Legegyeszerűbb formájában a jelölő kiterjesztés két értéket vesz fel: a jelölő kiterjesztés nevet (ebben az esetben static) az az érték követi, amelyet hozzárendelünk (például corlib.system.Environment.OSVersion).

```
<!-- A "static" jelölő kiterjesztés segítségével a Content
tulajdonságot a statikus tulajdonság értékére állítjuk -->
<label content="{x:Static corlib:Environment.OSVersion}" />
```

Íme, egy másik példa. Tételizzuk fel, hogy a különböző típusok teljesen meghatározott nevét szereznénk megszerezni, hogy aztán az értékeket label típus sok content tulajdonságaihoz rendeljük. Ebben az esetben a Type jelölő kiterjesztést alkalmazhatjuk:

```
<Page
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:corlib="clr-namespaces:system;assembly=mscorlib">
  <StackPanel>
    <label content="{x:Static corlib:Environment.MachineName}" />
    <label content="{x:Static corlib:Environment.OSVersion}" />
    <label content="{x:Static corlib:Environment.ProcessorCount}" />
    <label content="{x:Type Label}" />
    <label content="{x:Type Page}" />
    <label content="{x:Type Corlib:Boolean}" />
    <label content="{x:Type x:TypeExtension}" />
  </StackPanel>
</Page>
```

A példában a WPF label típusának, button típusának, valamint Boolean adat-típusának az mscorlib.dll szerelvényben teljesen meghatározott nevét kapjuk, ráadásaként pedig a Type jelölő kiterjesztés teljesen meghatározott nevét. Ha az oldalt megtekintjük a xamlpad.exe segédprogramban, a 28.14. ábrához hasonló képet kapunk.



28.14. ábra: Markuppótvírtők alkalmazása a tulajadonságokon statikus lagok értékeinek beállítására és típusinformációk megszerzésére

Az erőforrások és adatkötesek előzetes áttekintése

A XAML-szintaxis bevezetésének összefoglalásaként az utolsó példában nem csak az array jelölő kiterjesztés (amelyet az arrayextension osztálytípus képvisel) használatát vizsgáljuk meg, de tanulmányozunk néhány egyszerű deklaratív adatkövetést is, és áttekintjük a WPF-erőforrásokat. Az array jelölő kiterjesztés lehetővé teszi, hogy adatömböt rendeljünk egy meghátrázott tulajdonsághoz. Amikor a XAML segítségével definiálunk ilyen ömböt, a type jelölő kiterjesztéssel határozzuk meg, hogy milyen típusú ömböt hozunk létre (sztring ömböt, bittérkép ömböt stb.). Vizsgáljuk meg a következő <Page> definíciót:

```
<Page>
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:Corlib="clr-namespace:System;assembly=mscorlib">
    <StackPanel>
      <Label Content="{x:Array Type = Corlib:String}" />
    </StackPanel>
  </Page>
```

Há a xamlpad.exe segítségével megtekintjük a renderelt markupot, a megtekintési panelben láthatjuk kiterítve a `system.string[]` értéket. A kapcsolószerű szintaxis alkalmazásával nem tudjuk adatokkal feltölteni a tömböt. Ehhez részlemek segítségével létre kell hoznunk a saját tömbünket, amelynek típusa megegyezik az adott tömb típusával. Vizsgáljuk meg a következő XAML-definíció részletet:

```
<x:Array Type="Corlib:String">
  <Corlib:String>Sun Kil Moon</Corlib:String>
  <Corlib:String>Red House Painters</Corlib:String>
  <Corlib:String>Barnard Lakes</Corlib:String>
</x:Array>
```

A példában sztringtömböt hoztunk létre. Az `x:Array` típus hatókörében három szöveges értéket adunk hozzá, majd lezárjuk a definíciót. Noha ez szabályos XAML-markup, felmerülhet a kérdés, hogy hová helyezzük a tömbdeklarációnkat? Ha közvetlenül a `<Page>` elem hatókörébe helyezzük, akkor implicit módon beállítjuk a `<Page>` elem `content` tulajdonságát, és (ismét) láthatjuk a `system.string[]` értéket a xamlpad.exe megtekintési panelében (azonban nem ezt akartuk):

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:corlib="clr-namespace:System;assembly=mscorlib">
  <!-- Hmm, éppen most állítottuk be a content tulajdonságot! -->
  <x:Array Type="Corlib:String">
    <Corlib:String>Sun Kil Moon</Corlib:String>
    <Corlib:String>Red House Painters</Corlib:String>
    <Corlib:String>Barnard Lakes</Corlib:String>
  </x:Array>
</Page>
```

Ígazából az a célunk, hogy nevet adjunk a tömbnek, majd a markup valamely más pontján hivatkozzunk rá (például egy `ListBox` feltöltéséhez). Pontosan ezt tehetjük, ha a tömbünket *erőforrás elem*ben definiáljuk. Tisztaban kell lennünk azzal, hogy a WPF „erőforrások” nem mindig arra képezhetők le, mint amit rendszerint elvárunk (sztringtáblákra, ikonokra, bittérképekre stb.) A WPF-erőforrások segítségével bármelyik egyedi markupidarabot ábrázolhatunk, például a sztringtömbünket is (a WPF-erőforrásokkal kapcsolatban bővebb információkat a 30. fejezetben találunk).

Vizsgáljuk meg az utolsó <Page> definíciót, amely a key jelölő kiterjesztés segítségével a <stackpanel> elemhez adja a „goodmusic” sztringtömb-erőforrás típus ItemSource tulajdonságát a tömbünkre (vegyük észre, hogy ezúttal ugyanarra a kulcsra hivatkozunk):

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:corlib="clr-namespace:System;assembly=mscorlib">
  <StackPanel>
    <StackPanel.Resources>
      <x:Array Type="String" x:Key = "GoodMusic">
        <Corlib:String>Sun kit Moon</Corlib:String>
        <Corlib:String>Red House Painters</Corlib:String>
        <Corlib:String>Besnard Lakes</Corlib:String>
      </x:Array>
    </StackPanel.Resources>
    <Label Content="Really good music"
      <ListBox Width="200"
      ItemsSource="{StaticResource GoodMusic}">
    </StackPanel>
  </Page>
```

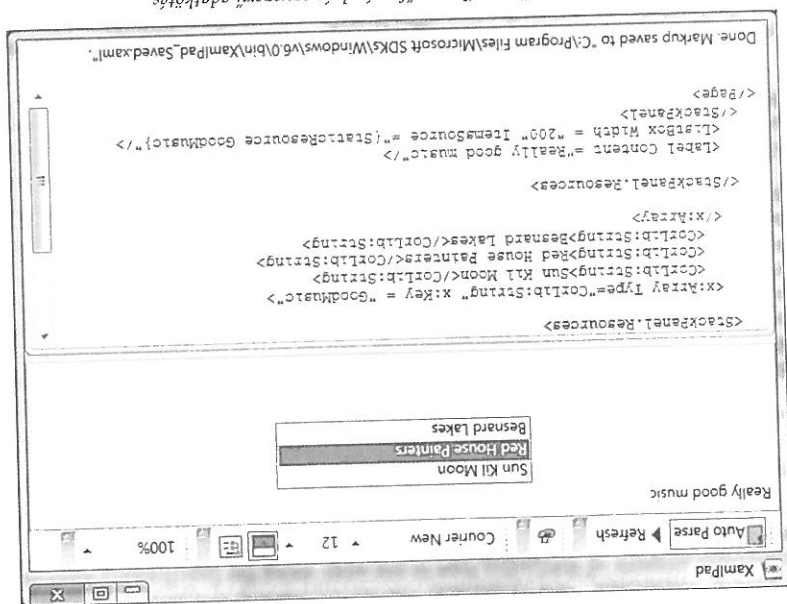
Mint látnuk, a <stackpanel> hatókörében beágyazott <stackpanel.Resources> elemet helyezzük el. Ez az elem biztosít helyet a sztringtömbünk számára. A staticresource markupsbővíítők olyan erőforrásokat képviselnek, amelyek a kezdeti kötetet követően nem változnak (innen a „státikus” erőforrás fogalma). Ha olyan erőforrással dolgozunk, amely az első kötés után módosul (például egy meghatározott rendszerszín), a másik jelölő kiterjesztést, a dynamicresource kiterjesztést használhatjuk. A 28.15. ábra a xamlpad.exe programban látható képet mutatja.

Megjegyzés A Key és a staticresource markupsbővíítők azért nem minősítették az x prefixummal (az itt vizsgált többi jelölő kiterjesztéssel ellentétben), mert azokat a http://schemas.microsoft.com/winfx/2006/xaml/presentation gyökér XML-névterében definiáltuk (mivel WPF-központúak).

Összefoglalva, eddig megvizsgáltunk több példát, amelyek bemutatták a XAML-szintaxis alapvető szempontjait. Nem vitás, hogy a XAML rendkívül érdekes technológia, amely lehetővé teszi, hogy deklaratív módon leírjuk .NET-objektumok hierarchiáját. Ez rendkívül hasznos a grafikus felhasználói

felületek konfigurálása során, de ne feledjük, hogy a XAML *bármely* szerelvény *bármely* típusát leírhatja, ha a típus az alapértelmezett konstruktori magában foglaló nem absztrakt típus.

Megjegyzés A XAML-t is feldolgozhatjuk futásidőben, erre a fejezet későbbi részében látunk példát.



28.15. ábra: Markupbővívők, statikus erőforrások és egyszerű adatkötés

A WPF és a XAML-alapszintaxis bevezető áttekintésének végére értünk. A következő fejezet ezen információkra épül, és bemutatja a WPF-elrendezéskezelőket, valamint az általuk tartalmazott vezérlőelemet. Azonban mielőtt tovább haladnánk, téjünk ki a Visual Studio 2008 WPF-projekt sablonokra, és vizsgáljuk meg a Microsoft Expression Blend szerepét.

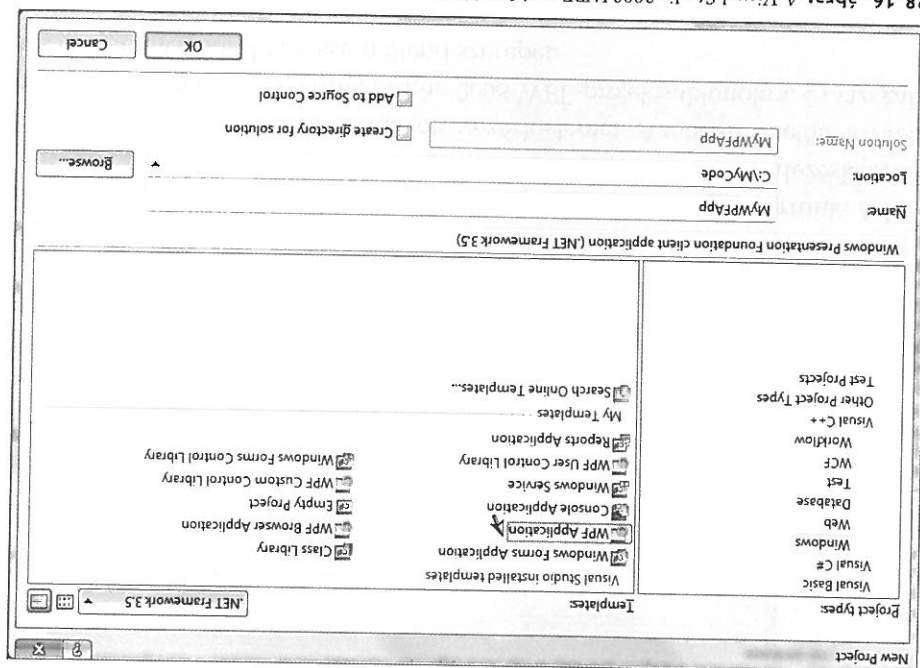
WPF-alkalmazások készítése a Visual Studio 2008 segítségével

A fejezet során egyszerű szövegszerkesztővel, a parancssoros fordítóval és a xamlpad.exe segítségével készítettük példánkat. Ennek az volt az oka, hogy a WPF-alkalmazások alapszintaxisát vizsgáltuk, és nem akartuk, hogy egy gra-

fikus tervező vonzó tulajdonságai eltereljük a figyelmiünket. Most, hogy látuk, hogyan lehet nyers WPF-alkalmazásokat készíteni, vizsgáljuk meg, a Visual Studio 2008 hogyan egyszerűsítheti a WPF-alkalmazások fejlesztését.

WPF-projektsablonok

A Visual Studio 2008 New Project párbeszédablaka WPF-központú projekt-munkaterületet definiál, amelyek mindegyikét a Visual C# gyökökér window csomópontja foglal magában. Mint azt a 28.16. ábrán láthatjuk, a WPF Application, a WPF User Control Library, a WPF Custom Control Library és a WPF Browser Application (például XBAP) sablonok közül választhatunk.

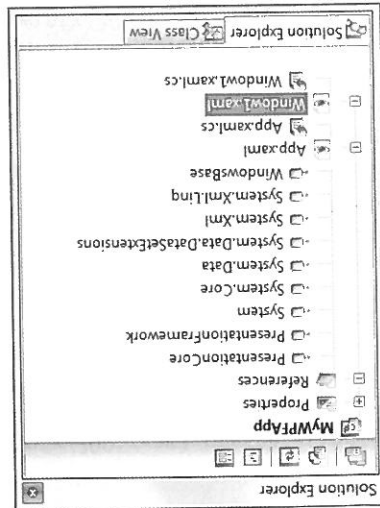


28.16. ábra: A Visual Studio 2008 WPF projektsablonjai

Amikor WPF asztali alkalmazást készítünk, a WPF Application projekt típust kell választanunk. Ekkor beállítódna az egyes WPF-szerelevények (a PresentationCore.dll, a PresentationFoundation.dll és a WindowsBase.dll) hivatkozásai, és a kódfájlok, illetve a XAML-definíciók alapján rendelkezésünkre bocsátja a studio a window és az Application típusokból származtatott típusokat (lásd a 28.17. ábrát).

A kezdeti ablak nevének módosítása

28.17. ábra: A WPF Application projekt típus kezdeti fájjai



Termékszintű projekt esetén egészen biztosan szeretnénk más nevet adni a Window-leszármazott típusunknak (és a fájlnak, amely a típust definiálja), hogy a Window1 alapértelmezett név helyett hozzáillőbb nevet kapjon. A WPF-alkalmazás számára szükséges összes mozgó részt figyelembe véve, ez jóval bonyolultabb, mint az első pillantásra tűnhet. Íme a folyamat részletes ismertetése.

Ha először a jobb gombbal a kezdeti Window1.xaml fájlt kattintunk, majd a Rename menüelemet választjuk, örömmel láthatjuk, hogy a rendszer a kapcsolódó Window1.xaml.cs fájlt is átnevezte a választásunknak megfelelően (például MainWindow.xaml-re). Azonban a *.xaml.cs fájlban az neve továbbra is Window1 marad. Ha a jobb gombbal a *.xaml.cs fájlban az osztály nevére kattintunk, és kiválasztjuk a Refactor > Rename lehetőséget, akkor alkalmasabb nevet (MainWindow) választhatunk a típus számára. Ha ezt követően megpróbáljuk futtatni a programunkat, az eredmény futásidejű kivétel lesz.

Ennek egyik oka az, hogy a nyitó <window> elem class attribútuma még mindig az eredeti window osztálynévre hivatkozik, és ezt az új osztálynévnek megfelelően módosítanunk kell:

```
<window x:
  class="MyWPFApplication.MainWindow"
  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>
</window>
```

Továbbá, az `<Application>` deklarációban a `StartupUri` tulajdonságot szintén módosítanunk kell, hogy a kezdeti `Window` típust magában foglaló, átnevezett XAML-fájlt határozza meg (`MainWindow.xaml`).

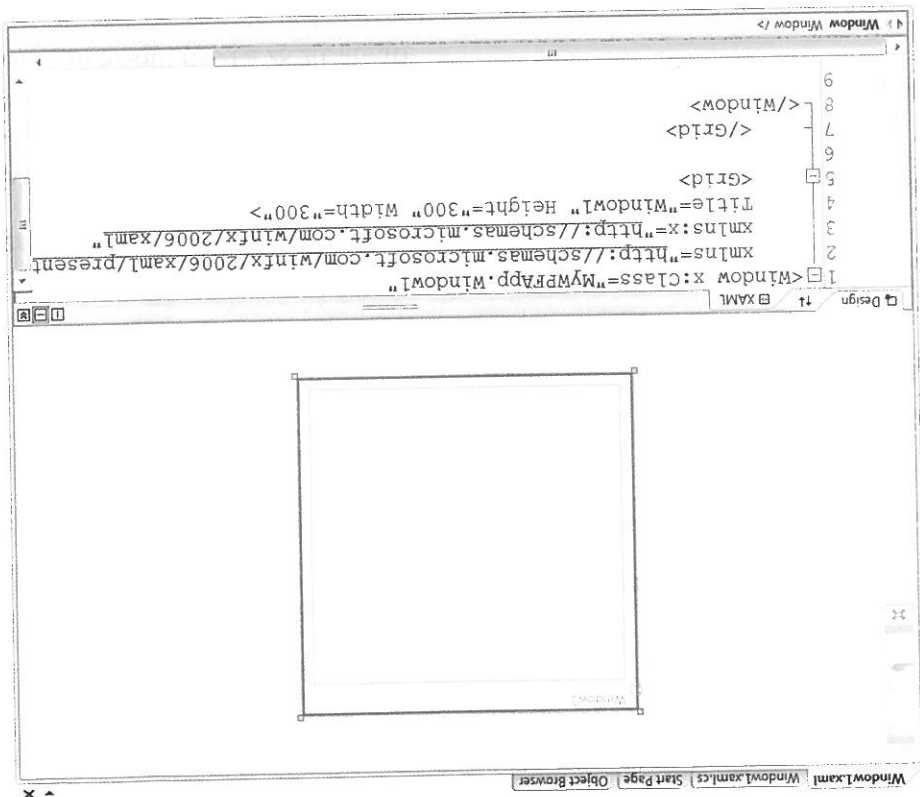
```
<Application x:Class="MyWPFApp.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
  <Application.Resources>
  </Application.Resources>
</Application>
```

Ezt követően már hiba nélkül lefordíthatjuk és futtathatjuk az alkalmazásunkat.

Megjegyzés Ha új `Window` típusokat illesztünk a WPF-projektbe, a kezdeti fájlunk nevének használatával a rendszer a megfelelő nevet osztja ki a fájlok és a típusdefiniciók számára, tehát nincs szükség további konfigurációra.

A WPF-tervező

A Windows Forms alkalmazásokhoz hasonlóan (lásd a 27. fejezetet) a Visual Studio 2008 is biztosít egy olyan eszköztárat, amely több WPF-vezérlőelemet, valamint egy vizuális tervezőt foglal magában. A tervező segítségével összeállíthatjuk a felhasználói felületet, a `Properties` ablakban pedig beállíthatjuk a kiválasztott vezérlőelem tulajdonságait. A `*.xaml` fájlok tervezője két panele van osztva. A felső panel alapértelmezés szerint a készülő ablak megjelenését és funkcionálitását mutatja, míg az alsó panelen a XAML-definíció látható (lásd a 28.18. ábrát).



28.18. ábra: A WPF-tervező

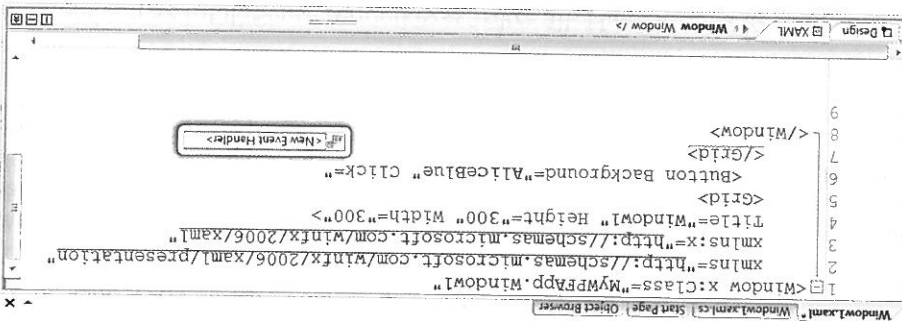
Megjegyzés A vizuális tervező megjelenítési paneljeit az osztott ablak beágyazott gombjai-val – például a Swap Panes gomb (amelyet a felfelé/lefelé nyílak jeleznek), a Horizontál és a Vertikal osztógombok stb. – átrendezhetjük. Szánjunk rá pár percet, és alakítsuk ki a szá-munkra legkényelmesebb konfigurációt.

Amikor a XAML-panelben XAML-markupot készítettünk, várakozásainknak megfelelően az IntelliSense rendelkezősűnkre áll. Ha például a kezdeti `<Grid>` típus hatókörén belül Button deklarációt kezdünk írni, a képernyőn megjelenik azon tulajdonságok és események listája, amelyeket a típus támogat. Továbbá, amikor kiválasztunk egy tulajdonság tagot, a 28.19. ábrán látható módon a lehetséges értékek listája jelenik meg.

A Windows Formstól eltérően a WPF-alkalmazásokban az események kezelése *nem* úgy történik, hogy a Properties ablakban a villám ikonnal rendelkező gombra kattintunk (valójában ez a gomb a WPF-alkalmazások fejlesztése során nem áll rendelkezésünkre).

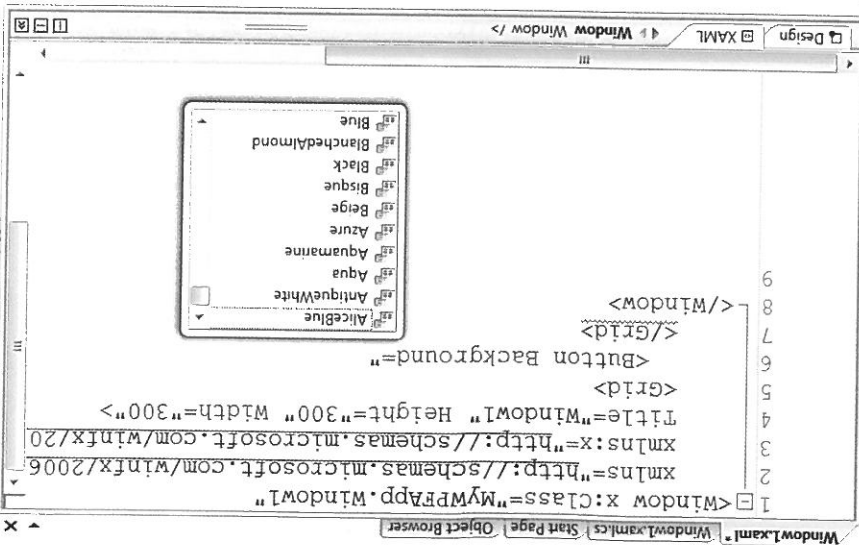
Ha kézzel írjuk be egy esemény nevet (idézőjelek között – ahogyan a XAML megköveteli), meghatározhatunk egy tetszőleges metódusnevet. Ha inkább azt szeretnénk, hogy az IDE alapértelmezett nevet generáljon (amely *Vezelő_Neve_EseményNev* formátumú), kattintsunk kérészer a <New EventHandler> előugró menüelemre! Mindkét esetben az IDE a forráskódhoz adja a megfelelő eseménykezelőt:

28.20. ábra: Események kezelése a vizuális tervező segítségével



Amikor a WPF-vezérlőeseményeit kezeljük, a megszokott C#-szintaxis segítségével kézzel megírhatjuk a teljes forráskódot; ha azonban a XAML-szerkesztőbe beírjuk egy esemény nevet, azzal aktiváljuk a New Event Handler előugró menüt (lásd a 28.20. ábrát).

28.19. ábra: A XAML IntelliSense



```
private void Button_Click(object sender, RoutedEventArgs e)
{
}
```

Megjegyzés Emlékezzünk rá, hogy ha vezérlőelem típusú tagváltozót szeretnénk definiálni az IDE-vel, értéket kell rendelnünk a Name tulajdonsághoz. Ha névtelen vezérlőelemek eseté-nyét kezeljük, az eseménykezelő neve egyszerűen *VezérlőNeve_EseményNeve[.Sorszám]* (pél-dául *Button_Click_1*, *Button_Click_2* stb.).

Miután megvizsgáltuk a Visual Studio 2008 alapvető eszközeit, amelyekkel WPF-alkalmazásokat kezelhetünk, készítsünk példaprogramot az IDE segít-ségével, amely bemutatja a XAML futásidőben megvalósított elemzésének fo-lyamatát.

XAML feldolgozása futásidőben: a SimpleXamlPad.exe

A WPF API támogatja a XAML-leírások programozott módon végrehajtott betöltését, elemzését és mentését. Ez a lehetőség sokszor rendkívül hasznos-nak bizonyulhat. Tételizzzuk fel például, hogy öt különböző XAML-fájlnk van, amelyek egy window típus megjelenését és funkcionálitását írják le. Felte-ve, hogy minden vezérlőelem (és a szükséges eseménykezelők) neve minden-fájlbán megegyezik, az ablakon (az alkalmazásnak átadott indítási argumen-tum alapján) dinamikusan alkalmazhatunk „grafikus arcúlatokat” (skineket). A XAML-lel futásidőben folytatott kommunikáció a *XamlReader* és a *XamlWriter* típusokra épül, amelyek mindegyikét a *System.Windows.Markup* névtér definiálja. Annak illusztrálására, hogy programozott módon hogyan nyerhetünk ki egy window objektumot külső *.xaml fájlból, létrehozunk egy WPF-alkalmazás projektet (*SimpleXamlPad* néven), amely a fejezet korábbi ré-szében vizsgált *xamlpad.exe* alapműködését utánozza.

Az alkalmazásunk egészen biztosan nem lesz olyan sokoldalú, mint a *xamlpad.exe*, de lehetővé teszi, hogy betrhassunk XAML-definíciókat, megte-kinthessük az eredményeket és a XAML-kódot külső fájlba menthessük. Miu-tán a Visual Studio 2008 segítségével létrehoztuk a *SimpleXamlPad* projektet, a kiindulási ablakot nevezzük át *MainWindow*-ra (a korábban leírt módon), és a következőképpen módosítsuk a XAML-definíciót:

Megjegyzés A következő fejezet részletesen ismerteti a vezérlőelemeket és a paneleket, most ne nyugtalankodjunk a vezérlőelem-deklarációk részletei miatt.

```
<window x:class="SimpleXamlPad.MainWindow"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:s="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Simple XAML Viewer" Height="338" Width="1041"
Loaded="Window_Loaded" Closed="Window_Closed"
WindowState="CenterScreen"
>
<DockPanel LastChildIsTab="True" >
<!-- Ez a gomb elindít egy ablakot a definiált XAML-lel -->
<Button DockPanel.Dock="Top" Name="btnViewXaml" Width="100"
Height="40" Content="View Xaml"
Click="btnViewXaml_Click" />
<!-- Ez a terület, ahová gépelhetünk -->
<TextBox AcceptsReturn="True" Name="txtXamlData"
FontSize="14" Background="Black" Foreground="Yellow"
BorderBrush="Blue" VerticalScrollBarVisibility="Auto"
AcceptsTab="True" >
</TextBox>
</DockPanel>
</window>
```

Először is vegyük észre, hogy a kezdeti <grid> típusú <dockpanel> típusra cseréljük, amely egy Button (btnViewXaml névvel) és egy TextBox (txtXamlData néven) típust foglal magában, és a kód kezeli a Button típus kattintási eseményét. Azt is figyeljük meg, hogy a Window Loaded és Closed eseményeit a nyitó <window> elem kezeli. Ha a tervező segítségével kezeljük eseményeinket, a következők forráskódot találjuk a MainWindow.xaml.cs fájlunkban:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void btnViewXaml_Click(object sender, RoutedEventArgs e)
    {
    }

    private void Window_Closed(object sender, RoutedEventArgs e)
    {
    }

    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
    }
}
```

Mielőtt folytatnánk, feltétlenül importáljuk a `MainWindow.xaml.cs` fájlunkba a következő névtereket:

```
using System.IO;
using System.Windows.Markup;
```

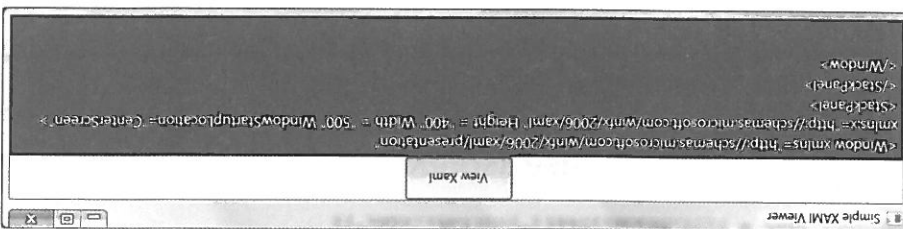
A Loaded esemény megvalósítása

A `foablak loaded` eseményének feladata annak meghatározása, hogy az alkalmazást tartalmazó könyvtárban pillanatnyilag megtalálható-e a `YourXaml.xaml` fájl. Ha ez a fájl létezik, beolvassuk az adatokat, és a rendszer azokat a `foablak`-ban egy `TextBox`-ba helyezi. Ha nem létezik, a `TextBox`-ot egy üres `ablak` alapértelmezett XML-leírásával töltjük meg (ez a leírás pontosan ugyanaz a `makup`, mint egy kezdeti `ablak`definiáció, egyetlen különbség, hogy a `Window` `content` tulajdonságának `[implicit]` beállításához a `<Grid>` helyett a `<StackPanel>` elemet alkalmazunk).

Megjegyzés Az XML-névteret képviselő, éppen készülő sztringet nem könnyű feladat beírni a beágyazott idézőjelekhez szükséges vezérlőkérekekre miatt (gépeljünk körültekintően!).

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Az alkalmazás főablakának betöltése során
    // helyezzünk el egy szerű XML-szöveget a szöveges blokkban!
    if (File.Exists(System.Environment.CurrentDirectory +
        "\\YourXaml.xaml"))
    {
        txtXamlData.Text = File.ReadAllText("YourXaml.xaml");
    }
    else
    {
        txtXamlData.Text =
            "<Window xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'>\n"
            + "<xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'>\n"
            + "Height='400' Width='500'\n"
            + "WindowState='CenterScreen'>\n"
            + "<StackPanel>\n"
            + "</StackPanel>\n"
            + "</Window>";
    }
}
```

A megközelítés segítségével a SimpleXamlPad.exe alkalmazás be tudja tölteni az előző munkamenet során létrehozott XAML-t, vagy szükség szerint megadja a markup alapértelmezett blokkját. Ezt követően futtathatjuk a programunkat, és a 28.21. ábrán látható képernyő jelenik meg a TextBoxban.



28.21. ábra: A SimpleXamlPad.exe első futtatása

A Button kattintási eseményének megvalósítása

Amikor a Button-ra kattintunk, először a YourXaml.Xaml fájlba menjünk a TextBox aktuális adatait. Ekkor a File.Open() metódussal beolvassuk a rögzített adatokat, hogy megszerezzünk egy Stream-leszármazott típust. Erre azért van szükség, mert a XamlReader.Load() megköveteli, hogy az elemezni kívánt XAML-t a stream (nem pedig egyszerű System.String) típusból származtatott típus képviselje.

Ha betöltöttük a létrehozni kívánt <Window> XAML-markupot, a memóriába belőli XAML alapján hozzuk létre a System.Windows.Window egy példányát, és modális párbeszédablakként jelenítjük meg:

```
private void btnViewXaml_Click(object sender, RoutedEventArgs e)
{
    // A szöveges blokk adatainak kitérása helyi *.xaml fájlba.
    File.WriteAllText("YourXaml.Xaml", txtXamlData.Text);

    // Ez az ablak dinamikusan XAML-kódot kápl.
    Window myWindow = null;

    // A helyi *.xaml fájl megnyitása.
    try
    {
        using (Stream sr = File.Open("YourXaml.Xaml", FileMode.Open))
        {
            // A XAML összekapcsolása a Window objektummal.
            myWindow = (Window)XamlReader.Load(sr);
            myWindow.ShowDialog();
        }
    }
}
```

Ha a gombra kattintunk, feltűnik egy ablak, amely a XAML-definíciókat jeleníti meg (de lehet, hogy egy üzenetdobozban elemzési hibát kapunk – na- miní meg (de lehet, hogy egy üzenetdobozban elemzési hibát kapunk – na- gyon figyelmesen gépeljünk!). A 28.22. ábra a lehetséges kimenetet mutatja. Nagyszerű! Egészen biztosan eszünkbe jut több lehetséges javítás az alkal- mazással kapcsolatban, de ehhez tudnunk kell, hogyan dolgozzunk a WPF- vezérlőelemekkel és azokkal a panelekkel, amelyekben ezek vannak. Mielőtt a következő fejezetben megvizsgálalnánk a WPF-vezérlőelemmodellt, a fejezetet a Microsoft Expression Blend alkalmazás gyors áttekintésével zárjuk.

```
<StackPanel>
  <Rectangle Fill = "Green" Height = "40" Width = "200" />
  <Button Content = "OK!" Height = "40" Width = "100" />
  <Label Content = "{x:Type Label}" />
</StackPanel>
```

Most pedig indítsuk el a programunkat, és írjunk XAML-markupot a szöve- ges területre. Ne feledjük, hogy ez a program (a xamlpad.exe programhoz ha- sonlúan) nem teszi lehetővé, hogy bármilyen kódgenerálással kapcsolatos XAML-attribútumot (például a class attribútumot vagy bármilyen esemény- kezelőt) meghatározzunk. Tesztelés céljából írjuk be a következő XAML- markpupot a <stackpanel> hatókörébe:

Az alkalmazás tesztelése

```
private void Window_Closed(object sender, EventArgs e)
{
    // A szöveges blokk adatainak kitírása helyi *.xaml fájlba.
    File.WriteAllText("YourXaml.xaml", txtXamlData.Text);
}
```

Végül, a Window típus Closed eseménye biztosítja, hogy a TextBox legfrissebb adatát rögzítse a YourXaml.xaml fájl:

A Closed esemény megvalósítása

Vegyünk észre, hogy a logika jelentős részét try/catch blokkba csomagoljuk. Így módon, ha a YourXaml.xaml fájl helytelen markpupot tartalmaz, az eredmény- ként kapott üzenetdobozban láthatjuk a hibákat, amelyeket okoztunk.

```
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

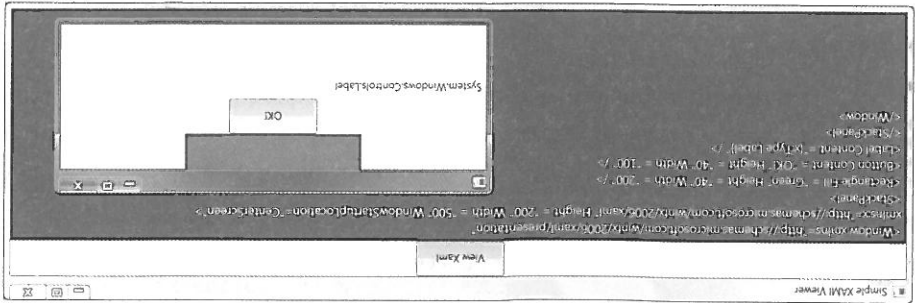
A Microsoft Expression Blend szerepe

Miközben az új technológiák, mint például a XAML és a WPF, tanulmányozása sok fejlesztő számára izgalmas, kevesen lelkesedünk a gondolatért, hogy ablakok, 3D képek, animációk és más összetevők leírásához több ezer sornyi markaput készítsunk. Az összetett entitások sokoldalú XAML-leírásának elkészítése még a Visual Studio 2008 segítségével is fáradszt és hibára hajlamosító folyamat. A Visual Studio 2008 jól felszerelt alkalmazás, amellyel forráskódot készíthetünk, és *finomhangolhatjuk* a XAML-leírások automatizálását szolgáló eszközök által generált XAML-definíciókat.

Emlékezzünk rá, hogy a WPF egyik legnagyobb előnye a kapcsolatok elkülönítése. Azonban a WPF nem pusztán a fájlok (például C#-kódfájlok és XAML-fájlok) szintén alkalmazza a kapcsolatokat elkülönítést. Valójában a WPF-alkalmazás tiszteltében tartja a kapcsolatokat elkülönítést azon eszközök szintén, amelyek segítségével az alkalmazásainkat készíthetjük. Ez fontos, mert a profi WPF-alkalmazások rendszerint megkövetelik, hogy lehetséges grafikus-művész szolgáltatásait vegyük igénybe az alkalmazás megjelenésének és funkcionálitásának megfelelő kialakításához. A nem technikai beállítottságu emberek nem szívesen használják a Visual Studio 2008 alkalmazást XAML írásához. Ezen problémák megoldására a Microsoft új termékcsaládot készített, amely az Expression-családba tartozik. Az Expression család tagjainak részletes ismeretöjtét a <http://www.microsoft.com/expression/webhelyen/talajluk>; röviden, az Expression Blend a sokoldalú WPF front end alkalmazások fejlesztésére rendezkedett be.

Forráskód A SimpleXamlPad projektet a forráskódkönyvtár 28. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

28.22. ábra: A SimpleXamlPad.exe működés közben



28. fejezet: A WPF és az XAML

Az Expression Blend előnyei

Az Expression Blend első fontos előnye, hogy a mód, amellyel a grafikusművész elkészíti a felhasználói felületet (nem egyezik meg, de) hasonló más multimédia-alkalmazásokhoz, például az Adobe Photoshop vagy a Macromedia Director programokhoz. Például az Expression Blend támogat az animációkhoz történetkereketek készítésére alkalmas eszközöket, szinkronizáló segédprogramokat, elrendezési és grafikus átalakító eszközöket és így tovább. Az Expression Blend továbbá biztosít olyan, a forráskód világa közeli tulajdonságokat, mint az adatkötések létrehozásának támogatása vagy a triggerek. Ettől függetlenül egy grafikusművész rendkívül sokoldalú felhasználói felületet készíthet anélkül, hogy akár egyetlen sornyi XAML-markupot vagy C#-kódot látna. A 28.23. ábrán az Expression Blend működés közben látható.



28.23. ábra: Az Expression Blend a háttérben a felhasználó számára láthatatlannul generálja a XAML-markupot

Az Expression Blend másik fontos előnye, hogy ugyanazokat a projekt-munkaterületeket használja, mint a Visual Studio 2008! Ha a grafikusművész katevületeket használja, a C#-fejlesztő megnyithatja ugyanazt a projektet, és belenyúlhat a forráskódba, hozzáadhat eseménykezelőket, finomhangolhatja a XAML-t és így tovább. A grafikusművész az Expression Blend eszközzel hasonlóképpen megnyithat létező Visual Studio 2008 WPF-projek-

teket, és kicsinosíthat egy jellegtelen front endet. Vagyis röviden, a WPF rendkívül rugalmas próbálkozás a kapcsolódó kódfájlok és a fejlesztőeszközök együttműködésének támogatására.

Bár az igaz, hogy Expression Blendhez hasonló eszköz használatának ismerete többé-kevésbé kötelező, ha a WPF segítségével élővonalbeli és sokoldalú alkalmazásokat készítünk, a könyv jelen kiadása nem foglalkozik ennek részleteivel. Ezen könyv célja az, hogy megvizsgálja a WPF mögöttes programozási modelljét, de nem merül el művészetelméleti részletekben. Ha mélyebben foglalkoztat bennünket a témakör, a Microsoft Expression család tagjainak próbaverzióját letölthetjük a támogatási webhelyről. Erdemes letölteni az Expression Blend próbaverzióját is, és megtapasztalni, hogy az eszköz mire képes.

Összefoglalás

A Windows Presentation Foundation (WPF) egy olyan interfész-eszközrendszer, amely a .NET 3.0 verziójának megjelenése óta áll rendelkezésünkre. A WPF fő célja, hogy a korábban egymáshoz nem kapcsolódó asztali technológiákat (2D grafika, 3D grafika, ablak- és vezérlőelem-fejlesztés stb.) összeolvassza és egységesítse egyetlen programozási modellben. A WPF-programok rendszerint a XAML-t (Extensible Application Markup Language – bővíthető alkalmazás jelölőnyelv) alkalmazzák, amely lehetővé teszi, hogy markup segítségével deklaráljuk WPF-elemünk megjelenését és funkcionálitását.

Mint azt a fejezetben láttuk, a XAML deklaratív szintaxisa segítségével leírhatjuk .NET-objektumaink hierarchiáját. A fejezetben a XAML tanulmányozása során megismerkedtünk a szintaxis újabb elemeivel, például a tulajdonság-elem szintaxisával, a csatolt tulajdonságokkal, valamint a típusátalakítók és a XAML-markupbővítmények szerepével. A fejezet végén megvizsgáltuk, hogyan tudunk a.XamlReader és a.XamlWriter típusok segítségével programozni módon együttműködni a XAML-definíciókkal, bepillantást nyertünk a Visual Studio 2008 WPF-specificus funkcióiba, és röviden áttekintettük a Microsoft Expression Blend szerepét.

Programozás WPF-vezérlőelemekkel

Az előző fejezetben megismerkedtünk a WPF programozási modell alapjaival, beleértve az ablak- és alkalmazásstípusok vizsgálatát, valamint a XAML több jellegzetességét. Itt a WPF-vezérlőelemek csoportját mutatjuk be részletesen, és bővíthetjük jelenlegi ismereteinket. A fejezetet a beépített WPF-vezérlőelemek vizsgálatával kezdjük, ezt követi majd két jelentős, a vezérlőelemekhez kapcsolódó WPF-téma: a függőségi tulajdonságok és a továbbított események.

Ha a központi programozási modellen már túljutottunk, a fejezet további részében több érdekes példával illusztráljuk, hogyan használhatjuk a WPF-vezérlőelemeket alkalmazásainkban. Például megismerjük, hogyan rendezzük a vezérlőelemeket a különböző WPF-tárolókban (canvas, grid, stack-panel, wrappanel stb.), és hogyan hozzunk létre főablakot menürendszerrel, állapotsávval és eszköztárral. A fejezet végén megvizsgáljuk a *vezérlőutasítási* sok használatát (amelyekkel beépített viselkedéseket adhatunk a különböző felhasználóifelhelyt-elemekhez és bemeneti parancsokhoz), valamint megismerkedünk a WPF *adatkötési* modellel.

Megjegyzés Sok vezérlőelem XAML-definíciója „laza (háttérkód nélküli) XAML-fájlként” szerepel a lehetőhető forráskódokban. A megjelölt kimenet megtekintéséhez az adott *.xaml fájlból átmásolhatjuk a markupt a 28. fejezetben létrehozott SimpleXamlPad.exe alkalmazásba. Vagy a <window> és </window> elemeket átirhatjuk <Page> és </Page> elemekre, majd a fájlt a kétszer rákattintva megtekinthetjük az Internet Explorerben.

A WPF vezérlőelem-könyvtár vizsgálata

Hacsak nem vagyunk teljesen járatlanok a grafikus felhasználói felület készítésében, a WPF-vezérlőelemek beépített készlete nem tartalmaz sok meglepő dolgot számunkra, függetlenül attól, hogy korábban melyik GUI-eszközrendszert használtuk (MFC, Java AWT/Swing, Windows Forms, VB 6.0, Mac OS X [Cocoa], GTK+/GTK# stb.). A 29.1. táblázatban láthatók a kapcsolódó működésük szerint csoportosított alapvető WPF-vezérlőelemek.

WPF-vezérlőelem	Példák tagokra	Jelentés
-----------------	----------------	----------

Alapvető felhasználói bemenni vezérlőelemek	Button, RadioButton, ComboBox, CheckBox, Expander, ListBox, Slider, ToggleButton, TreeView, ContextMenu, ScrollBar, TextBox, RepeatButton, RichTextBox, Label	Mint elvárható, a WPF a vezérlőelemek egész csatládjával rendelkezik a felhasználói interfész legbonyolultabb elemeinek felépítéséhez.
---	---	--

Ablakkereteket díszítő vezérlőelemek	Menu, Toolbar, StatusBar, ToolTip, ProgressBar	A felhasználói felületek ezen elemei a Window objektum keretét díszítik bemeneti eszközökkel (például Menu típusú) és felhasználói információk elemekkel (állapotsáv, eszközleírás stb.).
--------------------------------------	--	---

Média-vezérlőelemek	Image, MediaElement, SoundPlayerAction	Ezek támogatják az audio- / videólejátszást és képmegjelenítést.
---------------------	--	--

Elrendezést szabályozó vezérlőelemek	Border, Canvas, DockPanel, Grid, Gridview, GroupBox, Panel, StackPanel, Viewbox, WrapPanel	A WPF több olyan vezérlőelemet biztosít, amelyek segítségével a többi vezérlőelemet csoportosíthatjuk és rendszerezhetjük a megfelelőbb elrendezés érdekében.
--------------------------------------	--	---

29.1. táblázat: Alapvető WPF-vezérlőelemek

A 29.1. táblázatban bemutatott GUI-típusokon túl a WPF további vezérlőelemeket biztosít speciális dokumentumfeldolgozáshoz (documentviewer, flowdocument-reader stb.), valamint a Tablet PC fejlesztésekhez hasznos) Ink API-t támogató típusokat és különböző rögzített párbeszédablakokat (passwordbox, printdialog, dialogbox, openfiledialog és savefiledialog és a savefiledialog típusokat a PresentationFramework.dll szerialvény Microsoft.Win32 névtérben definiálja.

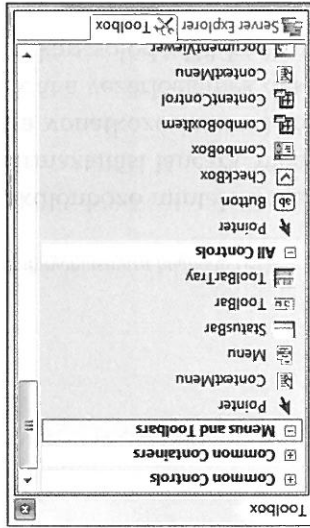
Megjegyzés A FileDialog, az OpenFileDialog és a SaveFileDialog típusokat a PresentationFramework.dll szerialvény Microsoft.Win32 névtérben definiálja.

Há Windows Forms tapasztalatokkal kezdünk a WPF tanulmányozásába, feltűnhet, hogy a beépített vezérlőelemek jelenlegi kínálata valamivel szegényesebb, mint a Windows Forms esetében (például a WPF nem rendelkezik „létógomb” – spin button – vezérlőelemekkel). A jó hír az, hogy sok ilyen hiányzó vezérlőelem gyorsan kifejlezhető a XAML segítségével, sőt, a projektek közötti újrafelhasználás érdekében akár még felhasználói vagy egyedi vezérlőelemként is modellezhető.

Megjegyzés A könyv jelen kiadása nem tér ki az egyedi WPF felhasználói vezérlőelemekre és WPF vezérlőelem-könyvtárakra. Ha szeretnénk többet megtudni a témakörrel, további információért lapozzunk fel a .NET Framework 3.5 SDK dokumentációját.

A WPF-vezérlőelemek és a Visual Studio 2008

Há Visual Studio 2008 segítségével hozunk létre új WPF-alkalmazás projekteket (lásd az előző fejezetet), a vezérlőelemek többségét a 29.1. ábrán látható módon (kategóriánként csoportosítva) megtaláljuk a Toolbox menüben. A Windows Forms projektekhez hasonlóan ezeket a vezérlőelemeket áthúzzuk a vizuális tervezőbe, és a Properties ablakban konfigurálhatjuk őket. Továbbá emlékezzünk vissza a 28. fejezetről, hogy ha az eseményeket a XAML-szerkesztő segítségével kezeljük, az integrált fejlesztői környezetet a tomatikus generálja a megfelelő eseménykezelőt a kódálljunkban.

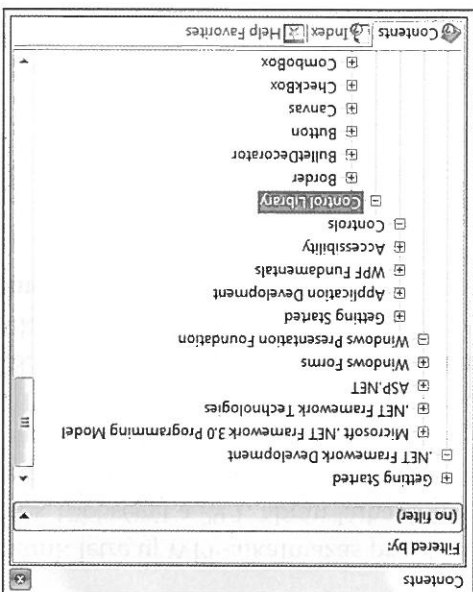


29.1. ábra: A Visual Studio 2008 Toolbox mutatja a beépített WPF-vezérlőelemeket

A részletek a dokumentációban találhatóak

A várakozásainkkal ellentétben ennek a fejezetnek *nem* az a célja, hogy végigvezessen minden egyes WPF-vezérlőelem minden egyes tagján. Inkább az alapvető vezérlőelemekről nyújtunk átfogó képet, különös hangsúlyt fektetve azokra az alapul szolgáló programozási modellekre (függetlenségi tulajdonságok, továbbított események, parancsok stb.) és kulcsfontosságú szolgáltatásokra, amelyek a legtöbb WPF-vezérlőelem esetén azonosak.

Az egyes vezérlőelemek konkrét működésére vonatkozó ismereteinket a .NET Framework 3.5 SDK dokumentációból melyíthetjük el – konkrétan a .NET Framework Development > Windows Presentation Foundation > Controls útvonalon érhető el (lásd a 29.2. ábrát).



29.2. ábra: Az egyes WPF-vezérlőelemek részletes leírása csak egy gombnyomásra található (F1)

Itt megtaláljuk az összes vezérlőelem teljes leírását, különböző mintakódokat (XAML és C# nyelven), valamint a vezérlőelem származtatási láncára, megvalósított interfészeire és alkalmazott attribútumaira vonatkozó információkat. A kis helyreigazítást követően gyorsan tekinthetjük át a vezérlőelemek deklarálását és konfigurálását a XAML-ben, valamint a kapcsolódó C#-kódban való használatát.

Vezérlőelemek deklarálása a XAML-ben

Hosszú évek során a fejlesztőket arra tanították, hogy a vezérlőelemeket viszonylag rögzített és kiszámítható elemeként kezeljék. Például a `Label` eszközöknek mindig szöveges tartalma van, és szinte soha nincs látható keretük (bár lehetne). A gombok szöveges tartalommal rendelkező szűrke téglalapok, és időnként akár beágyazott képekkel is rendelkezhetnek. Amikor egy projektet azt igényelte, hogy a „standard” eszközöket (például gombon típusú) egyédivé tegyék (például kör alakú képként megjelenő gombon vezérlő), a fejlesztők gyakran arra kényszerültek, hogy az egyedi vezérlőelemet külön ködolással állítsák elő.

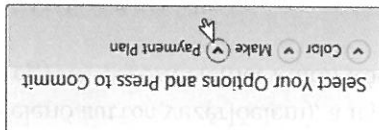
A WPF alapjaiban változtatja meg a vezérlőelemekről alkotott nézetelmét. Nemcsak arra van lehetőségünk, hogy a jelölőkkel kifejezzük a vezérlőelem megjelenését és használatát, de nagyon sok WPF-vezérlőelem (különösen a `contentControl` leszármazottai) úgy készül, hogy tetszőleges *tartalommal* rendelkezhesse. Emlékezzünk vissza a 28. fejezetből arra, hogy a `contentControl` tulajdonságot beállíthatjuk mind explicit (az elemek nyitó címkéjében, attribútumként), mind pedig implicit módon, ha a gyökeri gyermekelemeként megadjuk a beágyazott tartalmat.

Tételezzük fel, hogy létrehozunk egy `ControlReview` nevű új Visual Studio 2008 WPF alkalmazás projektet. Ahelyett, hogy felteleleznénk, hogy „minden Button vezérlőelem egy szűrke téglalap, amely szöveggel és esetleg képpel rendelkezik”, a XAML segítségével a következő implicit tartalmat adhatjuk meg a gombon típushoz (feltelelezve, hogy a következőket a kezdeti `<window>` elem `<grid>` elemén belül deklaráltuk):

```
<!-- Egyedi gomb beépített választási lehetőségeket! -->
<Button Name="btnPurchaseOptions" Height="100" Width="300">
  <StackPanel>
    <Label Name="lblInstructions" Foreground="DarkGreen"
      Content="Select your options and press to commit"/>
    <StackPanel Orientation="Horizontal">
      <Expander Name="colorExpander" Header="Color">
        <!-- Tételezzük fel, hogy itt szerepelnek az elemek... -->
      </Expander>
      <Expander Name="makeExpander" Header="Make">
        <!-- Tételezzük fel, hogy itt szerepelnek az elemek... -->
      </Expander>
      <Expander Name="paymentExpander" Header="Payment Plan">
        <!-- Tételezzük fel, hogy itt szerepelnek az elemek... -->
      </Expander>
    </StackPanel>
  </Button>
```

```
</StackPanel>
</StackPanel>
</Button>
```

Vegyük észre, hogy ez a `<Button>` típus három `<Expander>` típust tartalmaz (szinten a fejezet későbbi részében tárgyaljuk) egy `halmozásban rendeztünk` el. Mielőtt nagyon belmélyednénk az egyes vezérlők működésébe, nézzük meg a 29.3. ábrát, amelyen a megjelölt kimenet látható.



29.3. ábra: Egyedi Button vezérlőelem XAML-deklarációja

Egyszerű összehasonlításként nézzük meg, hogy ugyanezt a vezérlőelemet hogyan építettük volna fel Windows Forms használatával. Ebben az API-ban ilyen vezérlőelemet csakis úgy hozhattunk volna létre, ha építettünk egy `Button` típusból származtatott egyedi típust, amely manuálisan kezeli a grafikus tartalmak megjelenítését, módosítja a belső vezérlőelemek gyűjteményét, felülír bizonyos eseménykezelőket, stb.

Az asztali alkalmazások leíró nyelvének megszűléseivel az egyedi WPF-vezérlőelemek létrehozásának egyetlen nyomós oka az, ha egyedi viselkedéseket (események, virtuális metódusok felülírása, további interfésztípusok támogatása stb.) támogató vezérlőre van szükségünk, vagy mindenképpen egyedi tervezési idejű konfigurációs segédprogramokat kell támogatnunk. Ha csupán egyedi rendterelést létrehozásával kell törödnünk, a XAML is megteszi.

Együttműködés a vezérlőelemekkel a forráskódfájlokban

Emlékezzünk vissza az előző fejezetből, hogy a WPF típus tulajdonságait attritűtumokkal állíthatjuk be az elem nyitó címkéjén belül (vagy a tulajdonság-elem szintaxis alkalmazásával). Az esetek többségében a XAML-elemek attritűtumai közvetlenül a vezérlőelem osztályreprezentációjának tulajdonságaira és eseményekre képezhetők le a `system.windows.controls` névtérben. Eppen ezért mindig lehetőségünk van a vezérlőelemeket csak teljesen a jelölőben, vagy teljesen a kódban definiálni, esetleg a kettő kombinációját alkalmazni.

Megjegyzés A vezérlőelemhez a kapcsolódó kódfejletben kizárólag akkor van közvetlen hozzáférésünk, ha a XAML-definíció nyitó elemében a Name attribútum segítségével deklaráltuk.

Mivel az előző XAML-jelölés olyan típusokat tartalmaz, amelyekhez hozzárendeltük a name attribútumot, a típusokhoz közvetlen hozzáférésünk van a kódfejlétben, és szabadon kezelhetünk deklarált eseményeket is. Például a Label vezérlőelem fontsize tulajdonságának értékét a következőképpen módosíthatjuk:

```
public partial class MainWindow : System.Windows.Window
{
    public MainWindow()
    {
        InitializeComponent();
        // A label elem fontsize tulajdonságának módosítása.
        lblInstructions.FontSize = 14;
    }
}
```

Ez azért lehetséges, mert a XAML-definíciójukban Name attribútummal rendelkező vezérlőelemek tagváltozóként szerepelnek az automatikusan generált *.g.cs fájlban (lásd a 28. fejezetet):

```
public partial class MainWindow : System.Windows.Window,
    System.Windows.Markup.IComponentConnector
{
    //XAML-markup alapján definiált tagváltozók.
    internal System.Windows.Controls.Button btnPurchaseOptions;
    internal System.Windows.Controls.Label lblInstructions;
    internal System.Windows.Controls.Expander colorExpander;
    internal System.Windows.Controls.Expander makeExpander;
    internal System.Windows.Controls.Expander paymentExpander;
    ...
}
```

Amikor egy adott vezérlőelem eseményeit kívánjuk kezelni, a XAML-definícióban a következőképpen rendelhetünk metódusnevet az adott eseményhez:

```
<Button Name="btnPurchaseOptions"
        Click="btnPurchaseOptions_Click"
        Height="100" Width="300">
...
</Button>
```

A kapcsolódó kód[1] tartalmazza a metódus definícióját, amelynek formátuma az alapul szolgáló metódusreferencián alapul (ismét emlékeztünk vissza arra, hogy a Visual Studio 2008 IDE automatikusan frissíti a kód[1]unkat):

```
private void btnPurchaseOptions_Click(object sender,
    RoutedEventArgs e)
{
    MessageBox.Show("Button has been clicked");
}
```

Fontos megjegyezni, hogy az eseményeket nyugodtan kezelhetjük teljes mértékben a forráskódban. Ha például az előző kattintási esemény XAML-definícióját töröljük, a Window elem konstruktorát a következőképpen módosíthatnánk:

```
public MainWindow()
{
    InitializeComponent();

    // A Label elem FontSize tulajdonságának módosítása.
    lblInstructions.FontSize = 14;

    // A gomb kattintási eseményének kezelése.
    btnPurchaseOptions.Click +=
        new RoutedEventArgsHandler(btnPurchaseOptions_Click);
}
```

Most, hogy az alapvető vezérlőelem-model még frissen el emlekeztettünkben, a következő feladatunk, hogy megvizsgáljuk a WPF vezérlőelem-model két fontos (de kissé bonyolult) jellemzőjét: a függőségi tulajdonságokat és a továbbított eseményeket. Bár ezeknek a fogalmaknak a részletei a napi WPF programozási feladataink során általában rejtve maradnak, minél inkább tisztában vagyunk ezekkel az alacsony szintű részletekkel, a későbbiekben annál felkészültebben vehetjük bele magunkat a bonyolultabb WPF programozási feladatokba.

Forráskód A ControlReview kód[1]okat a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A függőségi tulajdonságok szerepe

A függőségi tulajdonságok szerepe

Mint sejtethető, a Windows Presentation Foundation API-k a .NET-típusrendszert minden egyes tagját (osztályok, struktúrák, interfészek, metódusreferenciák, felsorolások) és minden egyes lehetséges típustagját (tulajdonságok, metódusok, események, konstans adatok/írásvédett mezők stb.) használják a megvalósításaikban. A WPF azonban bevezetett egy új programozási mechanizmust, amelynek neve *függőségi tulajdonság*.

Megjegyzés A függőségi tulajdonságok WPF-specifikus programozási szerkezetek. A mai napig egyetlen .NET programozási nyelvnek sincs saját szintaxisa a tulajdonságok ezen sajátos típusának definiálására. Viszont a „propdp” C#-kódreszlet új függőségi tulajdonság vázát fogja generálni (a kódreszleteket lásd a 2. fejezetben).

A „normális” .NET-tulajdonságokhoz (amelyet a WPF-szakirodalom gyakran *CLR-tulajdonságoknak* nevez) hasonlóan a függőségi tulajdonságokat beállítottuk deklaratív módon XAML segítségével, vagy programozott módon a kódfejléiben. Ráadásul léteznek függőségi tulajdonságok (a CLR-tulajdonságokhoz hasonlóan) adatmezők beágyazására, és konfigurálhatókat őket írásvédettként, csak írhatóként vagy írható-olvashatóként, és így tovább.

A helyzetet még érdekesebbé teszi, hogy a legtöbb esetben szerencsére fogalmunk sincs arról, hogy CLR-tulajdonság helyett tulajdonképpen egy függőségi tulajdonságot állítottunk be! Például a WPF-vezérlőelemek a `Height` és a `Width` tagjaikat a `FrameworkElement` elemtől, míg a `Content` tagot a `ContentElement`-től öröklik – ezek mind függőségi tulajdonságok:

```
<!-- Éppen most állítottunk be három függőségi tulajdonságot! -->  
<Button Name = "btnMyButton" Height = "50" Width = "100"  
Content = "OK"/>
```

Ezeket a hasonlóságokat látva meglepőnek találhatjuk, hogy a WPF miért vezette be új fogalmat egy általánosan használt elvre. A válasz a függőségi tulajdonság mögött megbúvó megvalósításban rejlik. Implementálásuk után a függőségi tulajdonságok számos hatékony szolgáltatást biztosítanak, amelyeket a különböző WPF-technológiák, többek között az adatkötés, az animációs szolgáltatások, a témák és stílusok stb. használnak. Dióhéjban a függőségi tulajdonságok – a CLR-tulajdonságoknál megismert egyszerű adatbeágyazáson túl – a következő előnyöket nyújtják:

- A függőségi tulajdonságok öröközhetnek értékeiket a szülőelem XAML-definiációjából.
- A függőségi elemek támogatják annak lehetőségét, hogy az értékeket külső típusokkal állítsuk be (emlékezzünk vissza a 28. fejezetről arra, hogy a csatolt tulajdonságok éppen ezt teszik, mivel a csatolt tulajdonságok a függőségi tulajdonságokra épülnek).
- A függőségi tulajdonságok lehetővé teszik a WPF számára az érték kiszámítását több külső érték alapján.
- A függőségi tulajdonságok teremtik meg a visszahívási értéktételek és triggerek (elég gyakoriak animációk, stílusok és témák létrehozása során) számára szükséges infrastruktúrát.
- A függőségi tulajdonságok lehetővé teszik az adataik statikus tárolását (ezzel csökkenthető a memóriafelhasználás).

A függőségi tulajdonságok egyik legjelentősebb eltérése, hogy lehetővé teszik a WPF számára az érték kiszámítását több tulajdonságbeágyazott értéke alapján. Az egyéb kérdéses tulajdonságok közé tartozhatnak az operációs rendszer tulajdonságai (beleértve a rendszerszintű felhasználói beállításokat), az adatkötesen és az animáción/forगतókönyv-logikán, az erőforrásokon és a stílusokon alapuló értékek, vagy más XAML-elemekkel való szülő/gyermek kapcsolatot alapján ismert értékek.

Egy másik jelentős eltérés az, hogy a függőségi tulajdonságokat beállíthatjuk a tulajdonság értékváltozásának megfigyelésére, és ez alapján külső műveletek végrehajtásának kikényszerítésére. Például egy függőségi tulajdonság értékének változása kiválthatja azt, hogy a WPF megváltoztassa a Window objektumban a vezérlőelemek elrendezését, újrakössön külső adatforrásokhoz, vagy végighaladjon egy egyedi animáció lépésén.

Létező függőségi tulajdonság vizsgálat

A teljes igazsághoz hozzátartozik, hogy elég kicsi annak az esélye, hogy manuálisan kelljen függőségi tulajdonságot létrehozni WPF-projektjeink során. Valóban kizárólag akkor kell ezt megtennünk, ha olyan egyedi WPF-vezérlőelemet hozunk létre, amelyhez alosztályként már létező vezérlőelemet adunk, és így akarjuk annak viselkedését módosítani. Ebben az esetben, ha olyan tulajdonságot hozunk létre, amelynek WPF adatkötő motorral, témamotorral

vagy animációs motorral kell dolgoznia, vagy, ha a tulajdonságnak üzenetszó-rással közölnie kell, hogy megvalósított, a függőségi tulajdonság a megfélelő eszköz. Minden más esetben a normál CLR-tulajdonság is megteszi.

Bár ez teljességgel igaz, azért nem árt tisztában lenni a függőségi tulajdonságok alapvető felépítésével, mivel ezáltal a WPF egyes „miszlikus” tulajdonságai érthetőbbé válnak, és jobban megismerjük az alapul szolgáló WPF programozási modellt. A függőségi tulajdonság belső felépítésének bemutatására vizsgáljuk meg a következő C#-kódot, amely a `FrameworkElement` osztálytípus `Height` tulajdonságának megvalósítását mutatja be:

```
public class FrameworkElement : UIElement, IFrameworkInputElement,
    InputElement, ISupportInheritance, IHaveResources
{
    ...
    // Figyeljük meg, hogy ez DependencyProperty
    // típusú statikus mező.
    public static readonly DependencyProperty HeightProperty;

    // A statikus DependencyProperty mezőt a statikus konstruktorban
    // hozzuk létre és „regisztráljuk”.
    static FrameworkElement()
    {
        HeightProperty = DependencyProperty.Register(
            "Height",
            typeof(double),
            typeof(FrameworkElement),
            new FrameworkPropertyMetadata((double) 1.0 / (double) 0.0,
                FrameworkPropertyMetadataOptions.AffectsMeasure,
                new PropertyChangedCallback(
                    FrameworkElement.OnTransformity)),
            new ValueCallback(
                FrameworkElement.IsWidthHeight));
    }

    // Vegyük észre, hogy a Height tulajdonságnak még vannak get/set
    // blokkjai. Visszont a megvalósítás az örökölt
    // GetValue()/SetValue() metódusokat használja.
    public double Height
    {
        get { return (double) base.GetValue(HeightProperty); }
        set { base.SetValue(HeightProperty, value); }
    }
}
```

Mint láthatjuk, a függőségi tulajdonságok jóval több kiegészítő logikát köve-
telnek a tipikus CLR-tulajdonságoknál! Lassuk részleteseen, hogy mi is történik:
először is a függőségi tulajdonságokat a `system.windows.dependencyproperty`
osztálytípussal képviseljük, és szinte mindig publikus, statikus, írásvédelem

zöként deklaráljuk. Ne feledjük, hogy a függőségi tulajdonságok egyik előnye, hogy nem kötődnek közvetlenül az objektum példányához (ami a memóriátakarekosságot segít), ezért használunk statikus adatokat.

A függőségi tulajdonságok regisztrálása

Mivel a függőségi tulajdonságokat statikusan deklaráljuk, a kezdőértékeiket a típus statikus konstruktorában kapják meg. Azonban az egyszerű numerikus mezőktől eltérően a `DependencyProperty` objektum indirekt módon jön létre a statikus `DependencyProperty.Register` metódus visszatérési értékének rögzítésével. Ezt a metódust többszörösen túlterhelhettük; ebben a példában a `Register` metódust a következőképpen hívjuk meg:

```
HeightProperty = DependencyProperty.Register(
    "Height",
    typeof(double),
    typeof(FrameworkElement),
    new FrameworkPropertyMetadata(
        new FrameworkPropertyMetadataOptions.AffectsMeasure,
        new PropertyChangedCallback(
            FrameworkElement.OnTransform),
        new ValueCallback(
            FrameworkElement.IsWidthHeightValid));
```

A `Register` első argumentuma az osztály azon „normál” CLR-tulajdonságának neve, amely a `DependencyProperty` mezőt használja (jelen esetben a `Height`), míg a második argumentum az alap adattípusra vonatkozó típusinformáció, amelyhez kötöttük (itt `double` típusú). A harmadik argumentum annak az osztálynak a típusinformációját határozza meg, amelyhez a tulajdonság tartozik (esetünkben a `FrameworkElement`). Bár ez feleslegesenek tűnhet (hiszen a `HeightProperty` mezőt már definiáltuk a `FrameworkElement` osztályban), ez a WPF nagyon hasznos sajátossága, hiszen lehetővé teszi, hogy egy típus tulajdonságokat „csatoljon” egy másik típushoz (még akkor is, ha az osztálydefiníció lezárt).

Megjegyzés Emlékezzünk vissza, hogy a C# 2008 bővítő metódusai (lásd a 13. fejezet) lehetővé teszik új tagok hozzáadását a lezárt típusokhoz. A bővítő metódusok jelentik a legkövetlenebb módszert a típusok bővítésére olyan új funkcionálisokkal, amelyeknek nem kell WPF-központú szolgálatásokban részt venniük (pl. animáció).

A `register()` módszernek átadott utolsó argumentumok azok, amelyek ténylegesen megadják a függőségi tulajdonságok sajátos izét. Itt tudjuk megadni a `FrameworkPropertyMetadata` objektumot, amely leírja annak részleteit, hogy a WPF hogyan kezelje a tulajdonságot visszahívási értesítések esetén (ha a tulajdonságnak értesítenie kell másokat az értéke változásakor), az értéket hogy hitelesítsük, valamint azokat a különféle lehetőségeket is itt adhatjuk meg (amelyeket a `FrameworkPropertyMetadataOptions` felsorolás képvisel), amelyek azt szabályozzák, hogy a szoban forgó tulajdonság mire van hatással (dolgozik-e adatkötéssel, örökölheto-e, stb.).

Wrapper tulajdonság definíciója a `DependencyProperty` mezhöz

Amint a `DependencyProperty` objektum konfigurálásának részleteiről gondoskodtunk egy statikus konstruktorban, az utolsó feladatunk, hogy a mezt becsomagoljunk egy típusú CLR-tulajdonságba (esetünkben a `Height`-ba). Vagy beállítanak egy osztályszintű `double` tagváltozót, hanem ezt indirekt módon teszik a `System.Windows.DependencyObject` alaposztály `GetValue()` és `SetValue()` módszerainak alkalmazásával:

```
public double Height
{
    get { return (double) base.GetValue(HeightProperty); }
    set { base.SetValue(HeightProperty, value); }
}
```

Megjegyzés Az igazat megvallva, nem szükséges burkolo tulajdonságot létrehozni a `DependencyProperty` mezhöz, ha a mezo publikus, mivel statikusan is elérhetjük az örökölt `GetValue()/SetValue()` publikus metódusok hívása során. A gyakorlatban viszont a legtöbb függőségi tulajdonság rendelkezik barátságos burkolóval, mivel ez nagyon XAML-barát.

Most, hogy már láttuk, hogy a függőségi tulajdonságok hogyan épülnek fel a felszín alatt, tartsuk észben, hogy teljességgel lehetséges normal CLR-tulajdonságok használatát, amelyek ugyanazokat a szolgáltatásokat támogatják, mint a WPF függőségi tulajdonságok (értékesítések, statikus memóriafoglalás stb.). Viszont ehhez nem kevés háttérkódolás szükséges, amelyet manuálisan kell létrehozunk, és sok helyen megismételünk. A beépített `DependencyProperty` típus (és további intranszituív tulajdonságok) alkalmazásával ugyanezeknek a szolgáltatásoknak a készen elérhető megvalósítását kapjuk.

Mivel a függőségi tulajdonságok különböző WPF-központú típusok felhasználásával állnak elő, egyértelműen lehetőségnk van saját függőségi tulajdonságaink létrehozására – bár erre nem lesz szükség a könyvben található példákhoz. A következő kód összefoglalja a függőségi tulajdonságok deklarációjának központi elemeit (itt a tulajdonságot a statikus írásvédtett `DependencyProperty` típus deklarálása során regisztráljuk):

```
public class MyOwnerClass : DependencyObject
{
    // A DependencyProperty használata a MyProperty háttértáraként.
    // Ez lehetővé teszi az animációt, a stílus alkalmazását,
    // a kötetstb...
    public static readonly DependencyProperty MyProperty =
        DependencyProperty.Register("MyProperty", typeof(int),
            typeof(OwnerClass), new UIPropertyMetadata(0));

    // XAML-barát burkoló a statikus írásvédtett mezőhöz.
    // Ez azért szükséges, mert nem tudunk módszereket
    // (GetValue/SetValue) hívni XAML-ben.
    public int MyProperty
    {
        // A GetValue/SetValue a DependencyObject
        // alapszóllyból származik.
        get { return (int)GetValue(MyProperty); }
        set { SetValue(MyProperty, value); }
    }
}
```

Ha további részleteket szeretnénk erről a WPF programozási szerkezetről megtudni, olvassuk el a .NET Framework 3.5 SDK dokumentáció „Custom Dependency Properties” témakörét.

Továbbított események

Nem a tulajdonságok az egyetlen .NET programozási szerkezetek, amelyeknek egy kis frissítésre volt szükségük, hogy megfelelően működjenek a WPF API-val. A szabványos CLR-eseménymodell is finomodott egy kicsit, hogy biztosítsa az események feldolgozhatóságát oly módon, hogy az megfelelő legyen egy objektumra XAML-leírása számára. Tetelezzük fel, hogy létrehoztuk a `WPFCtrlEvents` nevű új WPF-alkalmazás projektet. Most frissítsük a kezdő ablak kezdeti XAML-leírását a következő <button> típus hozzáadásával a kezdeti <grid> típushoz:


```

<Button Name="btnClickme" Height="75" Width = "250"
  Click="btnClickme_Clicked">
<StackPanel Orientation="Horizontal">
  <Label Height="50" FontSize="20">Fancy Button!</Label>
  <Canvas Height="50" Width="100">
    <Ellipse Name="outerEllipse" Fill="Green" Height="25"
      Width="50" Cursor="Hand" Canvas.Left="25"
      Canvas.Top="12"/>
    <Ellipse Name="innerEllipse" Fill="Yellow" Height="15"
      Width="36" Canvas.Top="17" Canvas.Left="32"/>
  </Canvas>
</StackPanel>
</Button>

```

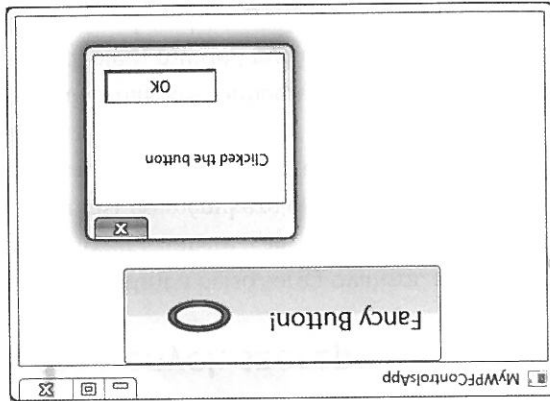
Figyeljük meg, hogy a <Button> esemény nyitó definíciójában úgy kezeltük a Click eseményt, hogy megadtuk a metódus nevét, mely akkor fut le, ha az esemény bekövetkezik. A kattintási esemény a routedeventhandler metódusregrációval működik, amely olyan eseménykezelőt vár, amely objektumot vesz fel első paraméterként, és System.Windows.RoutedEventArgs típust másodikként:

```

public void btnClickme_Clicked(object sender, RoutedEventArgs e)
{
    // Csinaljunk valamit, ha a felhasználó a gombra kattint.
    MessageBox.Show("Clicked the button");
}

```

A 29.4. ábra mutatja az elvart kimenetet, amikor rákattintunk a jelenlegi vezérlőelemre (megjelenítési okok miatt módosítottam a kezdeti <Grid> típust <StackPanel> típusra, ami megmagyarázza, hogy a gomb miért az ablak tetejének közepén van, ahelyett, hogy az ablak közepén helyezkedne el).



29.4. ábra: Események kezelése egy összetett Button típus esetén

A továbbiított buborék eseményminitának köszönhetően nem kell foglalkozni a StackPanel, majd a Button típushoz.

Ha a jelenlegi példában a felhasználó rákattint a belső sárga oválra, a kattintási esemény "kibugyan" a következő hatókör színtérre (canvas), majd a StackPanel, végül a Button típushoz, ahol a kattintási eseménykezelője található. Ehhez hasonlóan, ha a felhasználó rákattint a labai objektumra, az esemény "kibugyan" a StackPanel, majd a Button típushoz.

A továbbított buborékesemények szerepe

Megjegyzés A függőségi tulajdonságokhoz hasonlóan a továbbított események a WPF-specifikus segédtypusok használatával megvalósított WPF-specifikus események kezeléséhez el kellene sajátítaniunk a C#-szintaxis, amelyet a továbbított események

kapnánk, amelyet nehéz lenne karbantartani a jövőben.

A Windows Forms eseménymodellben az ehhez hasonló egyedi vezérlő-
elem megkövetelne tőlünk, hogy a gomb minden eleme számára létrehozzuk
a kattintási eseményt. Szerencsére a WPF *továbbított eseményei* automatikusan
gondját viselik ennek. A továbbított események automatikusan továbbítanak
egy eseményt feléle (vagy leéle) az objektumfan, megfelelő kezelőt keresve.
Konkrétában, a továbbított események három „bejárás stratégia” hasz-
nálhatnak. Ha egy esemény a kezdőponttól fölfelé mozog az objektumfan, ak-
kor az eseményt *buborekeseménynek* nevezzük. Ezzel ellentétben, ha egy ese-
mény a kezdőponttól lefelé halad a kapcsoltódó részelemek felé, akkor *lefutó*
esemény. Végül, ha egy eseményt csak az eredeti elem váltja ki és kezelei (amit
normális CLR-eseményként jellemezhetünk), akkor a neve *közvetlen esemény*.

Most vizsgáljuk meg a button típusunk jelenlegi összetételét. Több beágyazott elemet tartalmaz, a felhasznált felületének megjelenítéséhez (canvas, `Ellipse`, `Label` stb.). Képezzük el, hogy a WPF-eseménykezelés milyen unalmas lenne, ha arra kényserülnénk, hogy kezezzünk egy kattintási eseményt mindezen részlem esetében. Végül is a végfelhasználó bárhova kattinthat a gomb hátatrainak hatókörén belül (a label objektumra, az ovális zöld területre, a gombfelszínre stb.). Nemcsak munkáigényes lenne, ha a button típus minden aspektusához külön eseménykezelőt hoznánk létre, hanem olyan utaltatos kódot kapnánk, amelyet nehéz lenne karbantartani a jövőben.

egyetlen módon szeretnénk kezelni. Először kezeljük a mousedown eseményt ezen részem esetében (a grafikusunk rendszerelt típusok, mint az `Ellipse` nem támogatják a „kattintás” eseményt, azonban figyelhetjük az egér gombjának tevékenységét a mousedown, a mouseup stb. események révén):

```
<Button Name="btnClickme" Height="75" Width = "250"
Click="btnClickme_Clicked">
<StackPanel Orientation="Horizontal">
<Label Height="50" FontSize="20">Fancy Button!</Label>
<Canvas Height="50" Width="100">
<Ellipse Name="outerEllipse" Fill="Green"
Height="25" MouseDown="outerEllipse_MouseDown"
Width="50" Cursor="Hand" Canvas.Left="25"
Canvas.Top="12">
<Ellipse Name="innerEllipse" Fill="Yellow" Height="15"
Width="36" Canvas.Top="17" Canvas.Left="32">
</Canvas>
</StackPanel>
</Button>
```

Ezután implementálunk egy megfelelő eseménykezelőt, amely szemléltetés-képpen egyszerűen megváltoztatja a főablak `Title` tulajdonságát:

```
public void outerEllipse_MouseDown(object sender, RoutedEventArgs e)
{
// Az ablak címének megváltoztatása.
this.Title = "You clicked the outer ellipse!";
}
```

Ezzel most már különböző műveletek történnek attól függően, hogy a végfelhasználó hova kattintott (ami lenyegében a külső ellipszis és a gomb hatókörén belül minden más hely).

Megjegyzés A továbbított buborékesemények mindig a kezdőponttól a következő tartalmazó hatókör felé mozognak. Így, ha ebben a példában az `innerEllipse` objektumra kattintanánk, akkor az esemény a Canvas felé tartana, *nem* pedig az `outerEllipse` objektum felé, ugyanis mindkét `Ellipse` típus a Canvas hatókörében van.

A buborékesemények folytatása és leállítása

Jelenleg, ha a felhasználó az `outerEllipse` objektumra kattint, kiváltja az `Ellipse` típushoz regisztrált `MouseDown` eseménykezelőt, amikor a buboréklogika megáll (ezért nem láthatjuk, hogy a rendszer végrehajtja a `Button` típus kattintási eseménykezelőjét). Legtöbbször ez a kívánt hatás; azonban, ha arra

szereplőnk utasítani a WPF-et, hogy folytassa az esemény mozgatását fölfelé az objektumtán, beállíthatjuk a routedeventargs típus handled tulajdonságának értékét false-ra:

```
public void OuterMouseMove(object sender, RoutedEventArgs e)
{
    // Az ablak címének megváltoztatása.
    this.Title = "You clicked the outer ellipse!";

    // A továbbítás ("bugyborékolás") folytatódik!
    e.Handled = false;
}
```

Ebben az esetben láthatjuk, hogy az ablak címe megváltozik, amit a MessageBox megjelenítése követ a Button típus kattintási eseménykezelőjén belül. Röviden, a továbbított buborékesemények lehetővé teszik, hogy egy összetett tartalomcsoport akár egyetlen logikai elemként (pl. egy Button elemként), akár külön elemként (pl. egy Ellipse a Button típuson belül) szerepeljen.

A továbbított leftú események szerepe

A továbbított események tulajdonképpen *buborék* (ahogy az előbb leírtuk) vagy *leftú* jellegűek lehetnek. A leftú események (amelyek mind a `PreventMouseUp` előtaggal kezdődnek, pl. `PreventMouseUp`) a kezdőelemtől az objektumtában lévő belsőbb hatókörök felé haladnak. Nagyjából a WPF alapoztaly-könyvtárakban lévő összes buborékeseménynek van kapcsolódó leftú esemény párja, amely a buborék megfellelő *előt* indul el. Például mielőtt a `MouseDown` buborékeseményt a rendszer kiváltana, először a `PreventMouseUp` leftú esemény következik be.

Egy leftú esemény kezelése pont ugyanúgy néz ki, mint bármilyen más esemény kezelésének folyamata; egyszerűen rendeljük hozzá az eseménykezelő-`Id` nevét XAML-ben (vagy, ha kell, használjuk a megfellelő C# eseménykezelő-szintaxist a kód fájlban), majd implementáljuk a kezelőt a kód fájlban. A leftú és buborékesemények kölcsönös kapcsolatot szemléltessük a `PreventMouseUp` esemény kezelésével az `OuterEllipse` objektum esetében:

```
<Ellipse Name = "OuterEllipse" Fill = "Green" Height = "25"
MouseDown = "OuterEllipse_MouseDown"
PreventMouseUp = "OuterEllipse_PreventMouseUp"
Width = "50" Cursor = "Hand" Canvas.Left = "25"
Canvas.Top = "12"/>
```

Ezután tervezzük át az aktuális C#-osztálydefiniációt az eseménykezelők módosításával (minden típusnál), hogy információit fűzzenek egy sztringhez, és végül megjelenítsek az értéket. Ez lehetővé teszi, hogy megvizsgáljuk a hátterben kiváltott események folyamatait:

```
public partial class MainWindow : System.Windows.Window
{
    // Ezt használjuk arra, hogy tároljuk az eseményhez kapcsolódó
    // tevékenység adatait.
    string mouseActivity = string.Empty;

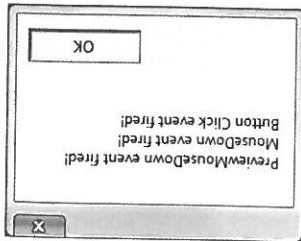
    public MainWindow()
    {
        InitializeComponent();
    }

    public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
    {
        // A végso sztring megjelenítése.
        mouseActivity += "Button Click event fired!\n";
        MessageBox.Show(mouseActivity);
        // A sztring törlése a következő teszthez.
        mouseActivity = string.Empty;
    }

    public void outerEllipse_MouseDown(object sender,
        RoutedEventArgs e)
    {
        // Adat hozzáadása a sztringhez.
        mouseActivity += "MouseDown event fired!\n";
        // A "bugborékolás" folytatódik!
        e.Handled = false;
    }

    public void outerEllipse_PreviewMouseDown(object sender,
        RoutedEventArgs e)
    {
        // Adat hozzáadása a sztringhez.
        mouseActivity = "PreviewMouseDown event fired!\n";
        // A "bugborékolás" folytatódik!
        e.Handled = false;
    }
}
```

Ha futtatjuk a programot, és nem kattintunk sehova a külső ellipszis határain belül, akkor egyszerűen a „Button Click event fired!” üzenet jelenik meg az üzenetdobozban. Azonban, ha a külső ellipszisziképen belülré kattintunk, akkor a 29.5. ábrán látható üzenetdoboz jelenik meg.



29.5. ábra: Először a lefutó, majd a buborékesemény

Elgondolkozhatunk, hogy vajon miért hajlamosak párban járni a WPF-események (egy lefutó és egy buborékesemény). A válasz az, hogy az események előmozdítással lehetőségünk van bármilyen speciális logika (adatellenőrzés, buborékesemény leállítás stb.) végrehajtására, mielőtt a buborékrész elindulna. Az esetek nagy többségében *nem lesz szükségünk* a Preview prefixumos lefutó események kezelésére, egyedül csak a (nem Preview prefixumos) buborék-

eseményekkel kell törődnünk.

A függőségi tulajdonság kézi megírásának feladatahoz hasonlóan, a lefutó események kezelése általában csak akkor szükséges, ha lezármatatunk egy már meglévő WPF-vezérlőelemet. Ehhez kapcsolódik, hogy ha egyedi WPF-vezérlőelemet készítünk, akkor létrehozhatunk egyedi továbbított eseményeket (akár buborék-, akár lefutó eseményeket) egy olyan technika használatával, amely hasonlít az egyedi függőségi tulajdonság készítéséhez. További részletekért forduljunk a „How to: Create a Custom Routed Event” témakörhöz a .NET Framework 3.5 SDK dokumentációjában.

Forráskód A WPFCtrlEvents kódrajlokat a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A Button típusok használata

Most, hogy áttekintettük a függőségi tulajdonságok és a továbbított események részleteit, abban a helyzetben vagyunk, hogy jobban megérthetjük magukat a WPF-vezérlőelemeket, kezdve a gombtípusokkal. Összönsően tisztában lehetünk a gombtípusok szerepével. Ezek a felhasználói felület olyan elemei, amelyeket megnyomhatunk az egér vagy a billentyűzet révén (az Enter vagy a szökőz billentyűvel), amikor ezek aktívak. A WPF-ben a ButtonBase szülőosztályként szolgál a három alapvető leszármazott típus (Button, RepeatButton és ToggleButton) számára.

A ButtonBase típus

Mint minden szülőosztály, a ButtonBase egy polymorf interfészt biztosít a leszármazott típusok számára (az alaposztályából, a ContentControl osztályból örökölt tagok mellett). Például a ButtonBase definiálja a kattintási eseményt is. Emellett ez a szülőosztály határozza meg az IsPressed tulajdonságot, amellyel végrehajthatunk egy műveletet, ha a leszármazott típust megnyomták, de még nem engedték fel. Ezenkívül a 29.2. táblázat bemutatja a ButtonBase absztrakt ösztály néhány további érdekes tagját.

A ButtonBase tagja		Jelentés
ClickMode	A ClickMode felsorolás értékeinek alapján ezzel a tulajdonsággal határozhatjuk meg, hogy mikor tüzeljen a kattintási esemény.	
Command	Ahogy a fejezet későbbi részében szó lesz róla, a felhasználói felület több elemének van társított „parancsa”, amelyet a Command tulajdonság beállításával rendelhetünk hozzá.	
CommandParameter	Ezzel a tulajdonsággal paramétereket adhatunk át a Command tulajdonság által megadott elemnek.	
CommandTarget	Ez a tulajdonság lehetővé teszi, hogy meghatározzuk a Command tulajdonság által megadott utasítás fogadóját.	

29.2. táblázat: A ButtonBase típus tagjai

Az utastításközponthoz tagok mellett (amelyeket a fejezet végén vizsgálunk meg) a legérdekesebb tag talán a `ClickMode`, amellyel megadhatjuk egy gombra kattintás három különböző módját. Ehhez a tulajdonsághoz a kapcsolódó `System.Windows.Controls.ClickMode` felsorolásból bármilyen értéket hozzárendelhetünk:

```
public enum ClickMode
{
    Release,
    Press,
    Hover
}
```

Tételezzük fel például, hogy a következő XAML-kóddal írunk le egy `Button`-t, melynek a `ClickMode` tulajdonságát `ClickMode.Hover` értékre állítottuk:

```
<Button Name = "btnHoverClick" ClickMode = "Hover"
        Click = "btnHoverClick_Click"/>
```

Ezáltal a kattintási esemény azonnal elindul, amint az egérkurzor bekerül a `Button` típus határain belülre. Noha lehet, hogy nem ez a legszerezencesebb eljárás egy tipikus nyomógomb esetében, a „lebegés” (`hover`) mód hasznos lehet egyedi stílusok, sablonok vagy animációk készítésekor.

A Button típus

Az első lezármazott típus, a `Button`, két azonnali érdeklődésre számot tartó tulajdonságot biztosít számunkra: az `IsCancel` és az `IsDefault` tulajdonságot, amelyek nagyon hasznosak, ha `OK` és `Mégse` gombokat tartalmazó párbeszédablakokat hozunk létre. Amikor az `IsCancel` értéke `true`, a gomb kattintási eseménye elsül, ha a felhasználó megnyomja az `Esc` billentyűt. Ha az `IsDefault` értéke `true`, a gomb kattintási eseménye elsül, ha a felhasználó megnyomja az `Enter` billentyűt. Tekintsük meg a két `Button` típus következő XAML-leírását:

```
<!-- Tételezzük fel, hogy ezeket egy window típus <StackPanel>
      osztályán definiáljuk -->
<Button Name = "btnOK" IsDefault = "true" Click = "btnOK_Click"
        Content = "OK"/>
<Button Name = "btnCancel" IsCancel = "true"
        Click = "btnCancel_Click" Content = "Cancel"/>
```


Ha szeretnénk tömöríteni a mögöttes kódjainkat, hogy egyetlen kezelőt használjon minden egyes eseményhez, módosíthatjuk a XAML-definiációt, hogy mind a checked, mind az unchecked esemény ugyanarra a kezelőre mutasson (vagyis a toggleonoffbuttonpressed eseménykezelőre), majd használjuk az isChecked tulajdonságot, hogy világossak az üzenetek:

```
protected void toggleonoffbuttonpressed(object sender,
    RoutedEventArgs e)
{
    if (toggleonoffbutton.IsChecked == false)
    {
        toggleonoffbutton.Content = "off!";
    }
    else
    {
        toggleonoffbutton.Content = "on!";
    }
}
```

Végül ne feledjük, hogy a togglebutton támogatja a háromállapotú működést (az isztreestate tulajdonság és az indeterminate esemény révén), és lehetővé teszi annak ellenőrzését, hogy a vezérlő bekapcsol, nem bekapcsolt vagy egyik sem. Noha furcsának tűnhet, hogy egy gomb ilyen módon figyelje önmagát, azonban értelmet nyer a dolog azon típusok esetében, amelyek a togglebutton típusból származnak, mint például a hamarosan bemutatásra kerülő checkbox típus.

Megjegyzés Altalános szabályként elmondható, a system.windows.controls.primitives névtérben definiált típusok (beleértve a togglebutton típust) további testreszabás nélkül nem számítanak azonnal hasznosnak.

A RepeatButton típus

Az utolsó tárgyalt, buttonbase-leszármazott típus a repeatbutton típus, amelyet szintén a system.windows.controls.primitives névtérben definiálunk. Ez a típus alapértelmezés szerint szintén hasonlít a standard button típusra, azonban megvan az a képessége, hogy folyamatosan elindítja saját kattintási eseményét, amikor a végfelhasználó megnyomott állapotban tartja a vezérlőt. A kattintási esemény elindításának gyakorisága függ a delay és az interval tulajdonságokhoz hozzárendelt értékektől (amelyek mindegyike ezredmásodpercben rögzített).

A repeatbutton típus (a togglebutton típushoz hasonlóan) önmagában valóban nem túl hasznos. Azonban a felírt viselkedés hasznos lehet teste bont felhasználói felületek létrehozásakor. Vegyük figyelembe a tényt, hogy – a

feladat nagy részét (szerencsére) átháríthatjuk olyan fejlesztőeszközökre, mint a Visual Studio 2008, a Microsoft Expression Blend vagy bármely más, külső gyártótól származó termékre. Ha az eszközök elkészítik az alapvető markupt, ezt követően manuálisan véggezhetjük el a XAML-definíciók szükségességes apró simításait.

Noha az eszközök a XAML terjedelmes részt „tálcán kínálják”, érdemes megismerkednünk a XAML-szintaxis alapvető működésével, és meg kell értenünk, hogy a rendszer hogyan alakítja át a markupt szabályos „NET-szerelvényekké”. Tanulmányozzuk a XAML-t működés közben! A következő példánkban kizárólag néhány *.xaml fájli felhasználásával készítettünk WPF-alkalmazást.

A MainWindow definíciója XAML-ben

Az első Window-leszármazott osztályunkat (mainwindow) a C#-kódban olyan osztálytípusként definiáltuk, amely kibővíti a system.windows.window alaposztályt. Az osztály egyetlen button típust foglal magában, amely ha a gombra kattintunk, meghívja a regisztrált eseménykezelőt. Ugyanezt a window típust XAML nyelvben definiálva (tetelezzük fel, hogy a markupt a mainwindow.xaml nevű fájlban definiáltuk) hasonló eredményt kapunk:

```
<!-- Íme, a window definíció -->
<window x:Class="SimpleXamlApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="My Xaml App" Height="200" Width="300"
  WindowStartupLocation="CenterScreen">
  <!-- Az ablak tartalmának beállítása -->
  <button Width="133" Height="24" Name="btnExitApp"
    Click="btnExitApp_Clicked">
    Exit Application
  </button>
</!-- A gomb kattintási eseménykezelőjének megvalósítása -->
<x:Code>
<CDATA[
private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
{
    // Az aktuális alkalmazás leírójának megszerzése és az
    // alkalmazás leállítása.
    Application.Current.Shutdown();
}
]>
</x:Code>
</window>
```

```

<!-- A main() metódus hívnyzik! A StartUpuri attribútum a metódus
funktcióját tekintve azonos megjelölje -->
<Application x:class="SimpleXamlApp.MyApp"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri="MainWindow.xaml">
</Application>

```

Emlékezzünk rá, hogy a XAML segítségével bármely nem absztrakt .NET-osztály markuját elkészíthetjük, ha az osztály támogatja az alapértelmezett konstruktor. Ennek fényében az alkalmazásobjektum definícióját is minden bizomnyal meghatározhatjuk markupban. Vizsgáljuk meg a következő tar-

Alkalmazásobjektum definíciója XAML-ben

lálható (végeredményben ezt a fájlt fogjuk elkészíteni).

több WPF-alkalmazásban az „igazi kód” egy kapcsolódó C#-kódfájlbán ta-
a felhasználói felület markuját és a programozási logika kapcsolatát. A leg-
műveletet elkülöníti egy helyen, a belső kód nem választja szét egyértelműen
ajánlott a <code> elemében. Bár ez az „egyetlen fájli megközelítés” az összes
Fontos rámutatunk, a szerkesztési funkcionális implementálása nem
közvetlenül értelmezni az adatokat.

csomagoltuk, és így megakadályozzuk, hogy az XML-elemek megpróbálják
példánkban nem feltétlenül lenne szükség) a kódot a CDATA hatókörbe
lőit és egyéb metódusait. Biztonsági intézkedésként (noha erre az aktuális
teszt, hogy közvetlenül a *.xaml fájlbán készítsük el az osztály eseménykeze-
A XAML-fájli utolsó említésre méltó pontja a <code> elem, amely lehetővé
tűk a metódust, amelyhez fordulni kell, ha a kattintási esemény bekövetkezik.
külső fömerek beállításán túl kezeltük a gomb kattintási eseményét: kijelöl-
implicit módon beállítjuk az ablak content tulajdonságát. A változónév és a
button példány megjelenését és funkcióját leíró markupot, amely segítségével
Figyeljük meg, hogy az ablak definíciójának hatókörében elkészítettük a
donságainak közvetlen leképezéseit.

amelyek – mint látnuk – a system.windows.window típus hasonló nevű tulaj-
tuk a title, a height, a width és a windowstartlocation attribútumokat,
foglalkozunk részletesen). A nyitó <window> elem hatókörében meghatároz-
winfx/2006/xaml definíciója (az XML-névterekkel a fejezet későbbi részében
ribútumot a XAML-központi XML-névter, a http://schemas.microsoft.com/
gítéssel definíciója a származtatott típus nevét. Az x prefix jelzi, hogy az at-
Első is vegyük észre, hogy a <window> gyökerelem a class attribútum se-

Egyetérthetünk abban, hogy az `Application` osztályból származó `C#`-osztálytípus és a XAML leírása közötti leképezés nem annyira áttekinthető, mint a `mainwindow` XAML-definíciója esetén. A példában nyomat sem találjuk a `main()` metódusnak. Mivel a .NET végrehajtható fájlnak rendelkeznie kell programbélépési ponttal, helyesen gondoljuk, hogy ez részben a `startup` tulajdonság alapján, a fordítás során generálódik. A hozzárendelt `*.xaml` fájlségítségével határozható meg, hogy az alkalmazás indítása során melyik `mainwindow`-ból származó osztályt kell létrehozni.

A `main()` metódus a fordítás során automatikusan elkészül, és a `<code>` elem segítségével – igény szerint – a következőképpen készíthetjük el az `exit` eseménykezelőt (vegyük észre, hogy többé nem statikus metódussal van dolgunk, mivel a metódus példányszintű taggá alakul a `myapp` osztályban):

```
<Application x:Class="SimpleXamlApp.MyApp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="mainwindow.xaml" Exit="AppExit">
    <x:Code>
    <CDATA[
private void AppExit(object sender, EventArgs e)
{
    MessageBox.Show("App has exited");
}
    ]]>
    </x:Code>
</Application>
```

XAML-fájlok feldolgozása az `msbuild.exe` segítségével

Készen állunk rá, hogy a markuptól érvényes .NET-szerelvényé alakítsuk. Az átalakítás során nem alkalmazhatjuk a `C#`-fordítót, és nem használhatunk reakciófájlt. Pillanatnyilag a `C#`-fordító nem alkalmas a XAML-markup feldolgozására. Azonban az `msbuild.exe` parancssori segédprogram tudja, hogyan kell a XAML-t `C#`-kóddá alakítani, és a kódot menetekben lefordítani, ha felhívjuk a figyelmeztető `*.targets` fájlokat.

Az `msbuild.exe` eszköz lehetővé teszi, hogy összetett fordítószkripteket építsünk (micsoda megjelölés!) XML segítségével. Az XML-definíciók egyik érdekessége, hogy ugyanolyan formátummal rendelkeznek, mint a Visual Studio `*.csproj` fájlok.

Ennek köszönhetően egyetlen fájlt definiálhatunk az automatizált parancssori fordítók számára, valamint a Visual Studio 2008 projekt számára. Vizsgáljuk meg a következő, lényegre törő `SimpleXamlApp.csproj` fájlt:

```
<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <RootNamespace>SimpleXamlApp</RootNamespace>
    <AssemblyName>SimpleXamlApp</AssemblyName>
    <OutputType>WinExe</OutputType>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="WindowsBase" />
    <Reference Include="PresentationCore" />
    <Reference Include="PresentationFramework" />
  </ItemGroup>
  <ItemGroup>
    <Page Include="MainWindow.xaml" />
  </ItemGroup>
  <Import Project="$(MSBuildPath)\Microsoft.CSharp.targets" />
  <Import Project="$(MSBuildPath)\Microsoft.Windows.Common-UI.targets" />
</Project>
```

A példában a `<PropertyGroup>` elem segítségével meghatározzuk a fordítás néhány alapvető tulajdonságát, például a gyökérnévteret, az előállított szerelvény nevét és a kimenet típusát (a `csc.exe` program/target:winexe beállításának megfelelője).

Az első `<ItemGroup>` azoknak a külső szerelvényeknek a gyűjteményét határozza meg, amelyre hivatkoznunk kell. Láthatjuk, hogy ezek azok a kulcsfontosságú WPF-szerelvények, amelyeket a fejezet korábbi részében vizsgáltunk. A második `<ItemGroup>` sokkal érdekesebb. Vegyük észre, hogy az `<ApplicationDefinition Include="MyApp.xaml" />` elem `Include` attribútumát ahhoz az `*.xaml` fájlhoz rendeljük, amely az alkalmazásobjektumunkat definiálja. A `<Page Include="MainWindow.xaml" />` az alkalmazásobjektum által feldolgozott ablakokat többi `*.xaml` fájlt, amelyek az alkalmazásobjektum által feldolgozott ablakokat (és oldalakat, amelyeket a XAML-bővítményalkalmazások készítése során gyakran használunk) definiálják.

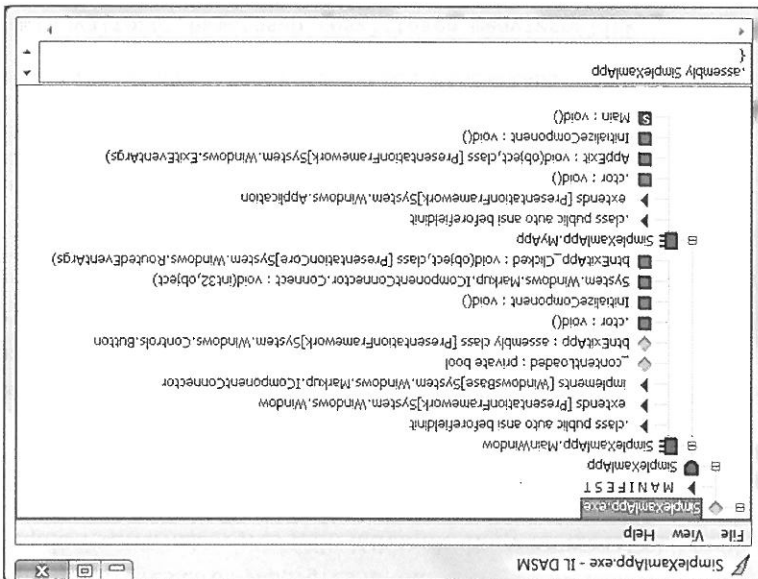
A `*.csproj` fájlt „varázssereje” az utolsó `<Import>` elemekben rejlik. Vegyük észre, hogy a fordító szkriptünk két `*.targets` fájlra hivatkozik, amelyek mindegyike további utasításokat foglal magában a fordítási folyamatra vonatkozóan. A `Microsoft.Windows.Common-UI.targets` fájl foglalta magában a XAML-definíciók megfelelő C#-kódra alakításához nélkülözhetetlen fordítási beállításokat, a `Microsoft.CSharp.targets` pedig a C#-fordítóval folytatott kommunikációhoz szükséges adatokat tartalmazza.

Megjegyzés A könyvben nem vizsgáljuk meg részletesen az msbuild.exe segédprogramot. Ha szeretnénk jobban megismerni az eszközt, a .NET Framework SDK dokumentációban keressük az „MSBuild” témakört.

Most feldolgozhatjuk a simplexam1app.csproj fájlnkat az msbuild.exe programmal:

msbuid SimpleExamApp.cspproj

A fordítás befejezését követően a szerverelvényünket a `generalt\bin\Debug\konvrtarban` találjuk. Ekkor a várt módon elindíthatjuk WPF-alkalmazásunkat. Egyetérthetünk abban, hogy néhány sornyi markup megírása az érves .NET-szerverelvények generálásának elég bízarr módja. Ha a biztonság kedvért megnyitjuk az `ildasm.exe` programmal a `stimplexam1app.exe` fájlt, láthatjuk, hogy a XAML-kódunk (valamilyen módon) átalakult végrehajtható alkalmazássá (lásd a 28.6. ábrát).



28.6. ábra: Markusz átalakítása .NET-szerelvényre? Erdékes...

Markup átalakítása .NET-szerelvénnyé

Annak érdekében, hogy megértjük, a markup hogyan alakult át .NET-szerelvénnyé, kicsit mélyebbre kell ásniuk az msbuild.exe folyamatba, és meg kell vizsgálniuk néhány, a fordító által generált fájlt, beleértve a szerelvénybe futásidőben beágyazott fontos bináris erőforrást.

XAML leképezése C#-kódra

Mint említettük, az MSBuild szkriptekben meghatározott *.targets fájlok több utasítást definiálnak, amelyek alapján a rendszer az XAML-elemeket a fordításhoz C#-kódra alakítja. Amikor az msbuild.exe feldolgozza a *.csproj fájlnkat, két *.g.cs fájlt készítet (a "g" *automatikusán generált* fájlt jelöl), amelyeket az \obj\Debug könyvtárba mentett. A *.xaml fájljaink neve alapján a kérdéses C#-fájlok a MainWindow.g.cs és a MyApp.g.cs fájlok.

Ha megnyitjuk a MainWindow.g.cs fájlt, láthatjuk, hogy az osztály kibővít a Window alaposztályt, és elvárásainknak megfelelően tartalmazza a btnnext-AppClicked() metódust. Az osztály egy System.Windows.Controls.Button típusú tagválozót is definiál. Furcsa módon *nem* találunk olyan kódot, amely a Button vagy a Window típus tulajdonságainak (Height, Width, Title stb.) beállításáról gondoskodik. A rejélyre hamarosan fény derül.

Végül vegyük észre, hogy az osztály definiál egy bool típusú privát tagválozót (-content Loaded néven), amelyről a XAML-markup közvetlenül nem tesz említést. Íme, a generált MainWindow típus definíciójának egy része:

```
public partial class MainWindow :
    System.Windows.Window,
    System.Windows.Markup.IComponentConnector
{
    internal System.Windows.Controls.Button btnnextApp;

    // Ezt a tagválozót hamarosan részletesen megvizsgáljuk.
    private bool _contentLoaded;

    private void btnnextApp_Clicked(object sender, RoutedEventArgs e)
    {
        // Az aktuális alkalmazás kezelőjének megszerzése,
        // és az alkalmazás leállítása.
        Application.Current.Shutdown();
    }
    ...
}
```


A window-ból származtatott osztály explicit módon megvalósítja a System.Windows.Markup névtérben definiált IComponentConnector interfészt. Az interfész egyetlen metódust definiál, a connect() metódust, amelyet az eredeti mainwindow.xaml fájlban meghatározott eseményre történő feliratkozást valósítja meg:

```
void System.Windows.Markup.IComponentConnector.Connect(
    int connectionId, object target)
{
    switch (connectionId)
    {
        case 1:
            this.btnExitApp = ((System.Windows.Controls.Button)(target));
            this.btnExitApp.Click += new
                System.Windows.RoutedEventHandler(
                    this.btnExitApp_Clicked);
            return;
        }
    }
    this._contentLoaded = true;
}
```

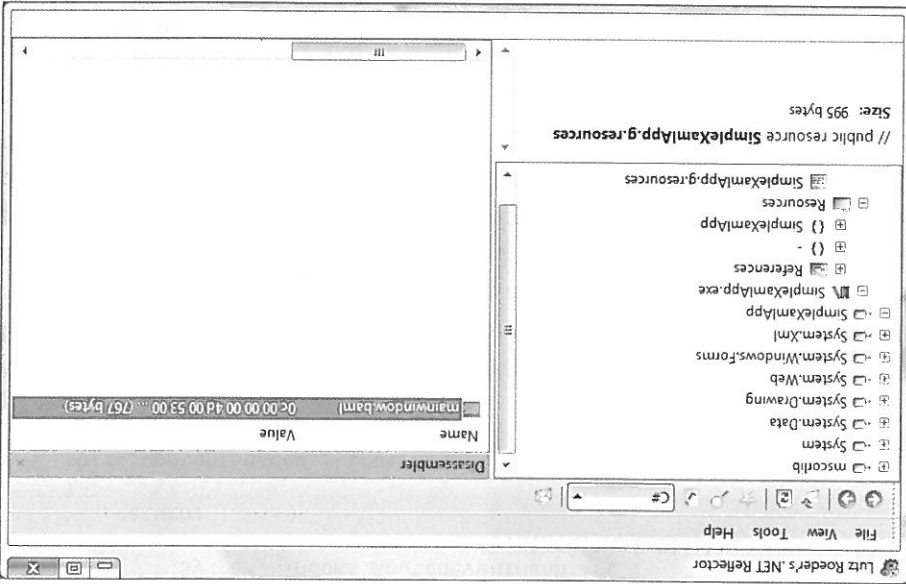
Végül a mainwindow osztály megvalósítja az InitializeComponent() metódust. Ez a metódus feloldja a szerelevényben a beágyazott erőforrás helyét az eredeti *.xaml fájl neve alapján. Ha az erőforrást a rendszer megtalálja, az Application.LoadComponent() meghívásával betölti az aktuális alkalmazásobjektumba. Utoljára pedig a (korábban említett) privát logikai tagváltozót állítjuk értékre, és így biztosítjuk, hogy a rendszer a kért erőforrást az alkalmazás ele-
te során pontosan egyszer töltsse be:

```
public void InitializeComponent()
{
    if (!_contentLoaded)
    {
        return;
    }
    _contentLoaded = true;
    System.Uri resourceLocator = new
        System.Uri("/SimpleXamlApp;component/mainwindow.xaml",
            System.UriKind.RelativeAbsolut);
    System.Windows.Application.LoadComponent(this, resourceLocator);
}
```

Ezen a ponton felmerül a kérdés: pontosan mi is az a beágyazott erőforrás?

A BAML szerepe

Az Amikor az msbuild.exe feldolgozza a *cspobj fájlnak, *baml kiterjesztéssel létrehozott egy fájlt, amely a kezdeti mainwindow.xaml fájlt alapján kapta a nevet. Mint azt a nevből kitálalhatjuk, a BAML (Binary Application Markup Language – bináris alkalmazás jelölőnyelv) a XAML bináris megjelentése. Ezt a *baml fájlt a rendszer (a generált *.g.resources fájl segítségével) erőforrásként beágyazza a lefordított szerelvénybe. A BAML révén a WPF-sze-relvények (sokkal tömörebb formátumban) magukban foglalhatják a teljes XAML-definíciójukat. Ezt magunk is ellenőrizhetjük, ha a reflector.exe programmal a 28.7. ábrán látható módon megnyitjuk a szerelvényünket.



28.7. ábra: A beágyazott *.baml erőforrás megtekintése Lutz Roeder .NET Reflector programjával

Az Application.LoadComponent() hívása beolvassa a beágyazott BAML-erőforrást, és feltölti a definiált objektumok hierarchiáját a megfelelő állapotokkal (például az ablak Height és Width tulajdonságaival). Valójában, ha a Visual Studio alkalmazásban megnyitjuk a *.g.resources fájlt, láthatjuk az eredeti XAML-attribútumok nyomait. A 28.8. ábra példaként a StartupLocation.CenterScreen tulajdonságot emeli ki.

gaik helyes beállításait.

Az automatikusan generált kód rejtvényének utolsó darabja a MyApp.g.cs fájlban bujik meg. Láthatjuk az Application-leszármazott osztályunkat a megfelelő main() belepési pont módszerrel. A módszer megvalósítása az Application-leszármazott típuson hívja az InitializeComponent() metódust, amely cserében beállítja a startupuri tulajdonságot, és így lehetővé teszi az objektumok számára, hogy a bináris XAML-definíció alapján meghatározott tulajdonságaik helyes beállítással.

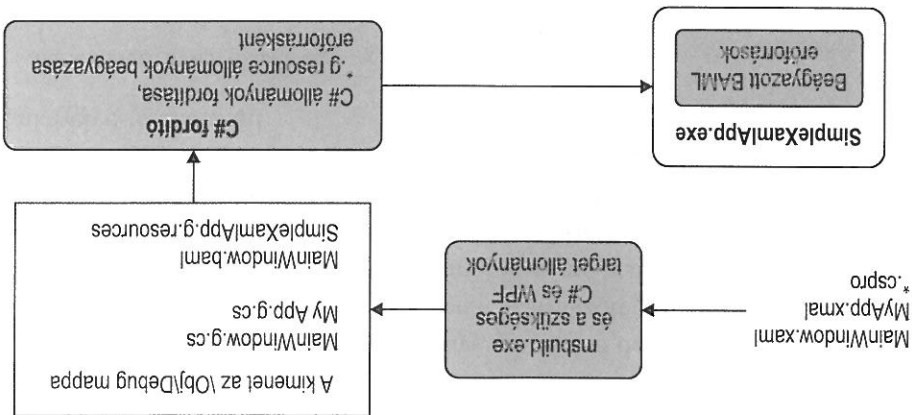
28.8. ábra: *Ime a BAMLI*

[illegible]

XAML-ből szerelvény: a folyamat összefoglalása

28. fejezet: A WPF és az XAML

Ejha! Összefoglalva, egy teljes .NET-szerelvényt hoztunk létre kizárólag három XML-dokumentum felhasználásával (a dokumentumok egyikét az msbuild.exe segédprogram használta). Mint láttuk, az msbuild.exe a *.targetId.exe definiált segédbeállítások segítségével dolgozza fel a XAML-fájlokat (és generálja az *.bam1 fájlt) a fordítási folyamat számára. Ezeket a részletek a háttérben hajtódtak végre. A 28.9. ábra bemutatja a *.xaml fájlok fordításidejű feldolgozásának átfogó képét.



28.9. ábra: A XAML-ből szerelvény fordításidejű folyamat

Fontos rámutatnunk, hogy amikor a fordító a megfelelő C#-kód és a bináris erőforrás előállításához feldolgozza az összes *.xaml fájlnkat, azokra gyakorlatilag a továbbiakban semmi szükség (és soha nem kell őket a végrehajtható fájllal együtt szállítani). A fejezet végén látjuk majd, hogy a *.xaml fájlt programozott módon végrehajtott olvasásával módunkban áll dinamikusan létrehozni window objektumot. Ebben az esetben a fizikai *.xaml fájlt nem szükséges az alkalmazással együtt szállítani.

Forráskód A SimpleXamlApp projektet a forráskódkönyvtár 28. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A kapcsolatok elkülönítése mögöttes kódfrájlokkal

Mielőtt elkezdnenk beleásni magunkat a XAML részleteibe, meg kell vizsgálnunk az alap programozási modell még egy aspektusát: a kapcsolatok elkülönítését. Emlékezzünk rá, hogy a WPF fő célja, hogy elkülönítse a felhasználói felület tartalmát a programozási logikától, és az aktuális példák erről nem gondoskodtak.

Ahelyett, hogy közvetlenül a XAML <code>elem hatókörébe ágyazzanánk az eseménykezelőinket (és az egyedi metódusokat), szerencsésébb különálló C#-fájlt definiálni a megvalósítási logika meghatározásához, és így a *.xaml fájlok kizárólag a felhasználói felület markumpját foglalják magukban. Vizsgáljuk meg a következő mögöttes MainWindow.xaml.cs kódfájlt (megjegyezés szerint a C# mögöttes kódfrájlt neve *.xaml.cs formátumú):

```
// MainWindow.xaml.cs
using System;
using System.Windows;
using System.Windows.Controls;

namespace SimpleExamApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            // Jegyezzük meg! A metódust a
            //generated MainWindow.g.cs fájl definiálja.
            InitializeComponent();
        }

        private void btnExitApp_Clicked(object sender,
            RoutedEventArgs e)
        {
            // Az aktuális alkalmazás kezelőjének megszerzése,
            // és az alkalmazás leállítása.
            Application.Current.Shutdown();
        }
    }
}
```

A példában részleges osztályt definiáltunk (az eseménykezelő logika számára), amelyt a rendszer a *.g.cs fájlban található ugyanolyan típusú részleges osztálydefiniációval fűszli össze. Mivel az `InitializeComponent()` metódust a `MainWindow.g.cs` fájlban definiáltuk, az ablakunk konstruktora a beágyazott BAML-erőforrás betöltéséhez és feldolgozásához meghívja a metódust. Így szerint mögöttes kódfejt is készíthetünk az `Application`-leszármazott típusunk számára. Mivel az események nagy része a `MyApp.g.cs` fájlban zajlik, a `MyApp.xaml.cs` fájlban található kód alig több valamivel a következőknél:

```
// MyApp.xaml.cs
using System;
using System.Windows;
using System.Windows.Controls;

namespace SimpleXamlApp
{
    public partial class MyApp : Application
    {
        private void AppExit(object sender, EventArgs e)
        {
            MessageBox.Show("App has exited");
        }
    }
}
```

Mielőtt az `msbuild.exe` segítségével lefordítanánk a fájljainkat, módosítanunk kell a *.csproj fájlunkat. A (félkövér betűvel szedett) `<Compile>` elemek segítségével biztosítanunk kell, hogy a rendszer az új C#-fájlokat is feldolgozza a fordítás folyamat során:

```
<Project DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <RootNamespace>SimpleXamlApp</RootNamespace>
    <AssemblyName>SimpleXamlApp</AssemblyName>
    <OutputType>WinExe</OutputType>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="System" />
    <Reference Include="WindowsBase" />
    <Reference Include="PresentationCore" />
    <Reference Include="PresentationFramework" />
  </ItemGroup>
  <ItemGroup>
    <ApplicationDefinition Include="MyApp.xaml" />
    <Compile Include="Mainwindow.xaml.cs" />
    <Compile Include="MyApp.xaml.cs" />
    <Page Include="MainWindow.xaml" />
  </ItemGroup>
</Project>
```

```
</ItemGroup>
<Import Project="$(MSBuildPath)\Microsoft.CSharp.targets" />
<Import Project="$(MSBuildPath)\Microsoft.WindowsFX.targets" />
</Project>
```

Ha átadjuk a fordítószkriptünket az msbuild.exe segédprogramnak, eredményként ugyanazt a végrehajtható szerelvényt kapjuk. Fejlesztési szempontból azonban a programozási logikából (C#) a leírás (XAML) letisztult részletét állítottuk elő. Mivel ez a WPF-fejlesztés ajánlott módja, örömmel vehetjük tudomásul, hogy a Visual Studio 2008 eszközzel létrehozott WPF-alkalmazások mindig az előbbieken bemutatott mögöttes kód-modellt használják.

Forráskód A CodeBehindXamlApp projektet a forráskódkönyvtár 28. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xiv. oldalát.

A XAML-szintaxis

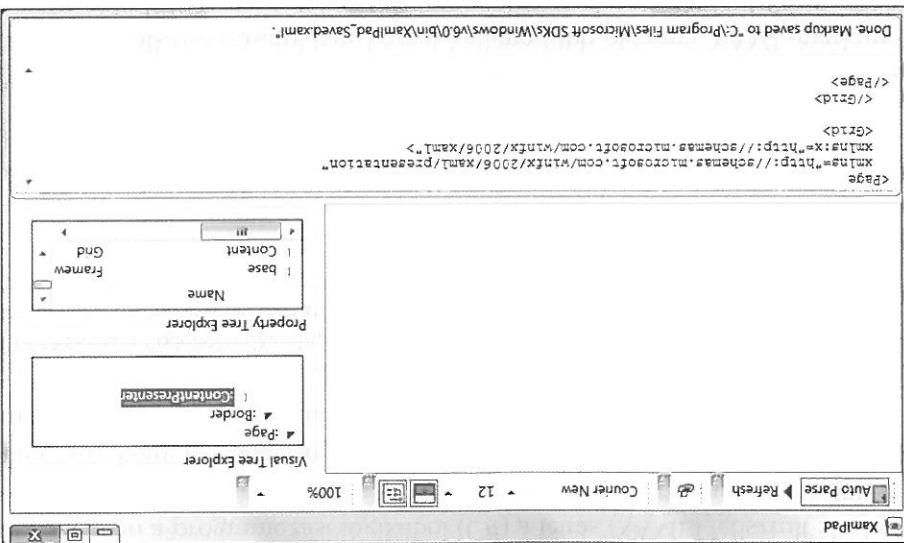
Mint azt a fejezet korábbi részében említettük, rendkívül kicsi az esélye annak, hogy WPF-alkalmazásainkban kézzel kellene több oldalnyi XAML-markupot megírjunk, hiszen ezt a feladatot a megfelelő eszközök (Visual Studio 2008, Microsoft Expression Blend stb.) elvégzik helyettünk. Azonban minél többet megtanulunk a jól formázott *.xaml fájlok szintaxisáról, annál felkészültebbek leszünk az automatikusan generált markup beállítására és módosítására, valamint melyebb ismereteket szerzünk magáról a WPF-ről. Most pedig ássunk a XAML-alapszintaxis melyére (a következő fejezetekben – ahol szükség lesz rá – további XAML-szintaxis példákkal találkozunk).

XAML-kísérletek a XamlPad segítségével

A XAML tanulmányozása során biztosan létrehozunk valamilyen tartalmat, és az eredményt gyorsan szeretnénk látni. Ehhez a Microsoft az SDK részeként a xamlpad.exe segédprogramot bocsátja rendelkezésünkre.

Megjegyzés Érdemes módon sem a .NET Framework 3.5 SDK, sem a Visual Studio 2008 nem tartalmazza a xamlpad.exe programot. A Windows SDK letöltésével juthatunk hozzá ehhez az eszközhöz, a fejezet legutolsó példájában C# segítségével elkészítjük a xamlpad.exe lecsupa-sztított verzióját.

Ha letöltöttük a Windows SDK-t, hogy hozzájussunk a xamlpad.exe programhoz, az eszközt a Start > All Programs > Microsoft Windows SDK > Tools menüelem segítségével indíthatjuk. A 28.10. ábra a xamlpad.exe kezdeti képernyőjét mutatja, amelyben aktiváltuk a Visual Tree Explorer beállítását (az eszköztár megfelelő gombjával).



28.10. ábra: A XamlPad a XAML-markup valós idejű megjelenítést biztosítja

A XamlPad segítségével az ablak alján található panelben szerkeszthetjük a XAML-markupot, és a felső részben megtekinthetjük a kimenetet. Amikor első alkalommal indítjuk az eszközt, egy üres <Page> deklarációt találunk, amelyet arra használunk, hogy egy XBAP-alkalmazás markupját tartalmazza:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <!-- Ide írhatjuk a XAML-markupot! -->
  </Grid>
</Page>
```

Noha a XamlPad nem teszi lehetővé, hogy a XamlPad nézet ablakában közvetlenül megtekinthessük egy <Window> elem markupját, a <Page> elemet <Window> elemre módosíthatjuk, és az F5 funkciógombbal egy különálló ablakban megtekinthetjük a tartalmát. Ne feledjük továbbá, hogy a <Page> és a <Window> elem markupja megegyezik!

Megjegyzés A XamlPad nem teszi lehetővé, hogy olyan markupt készítsünk, amely a kód fordítását vonja maga után. Ez magában foglalja a class attribútum definícióját (kódfrájl meghatalmazásához), a <code> elemek használatát, illetve bármely XAML-kulcsszó alkalmazását, amely a forráskód fordítását vonja maga után (mint például a FileReader vagy a ClassModifier). Bármilyen erre irányuló próbálkozás markuphibához vezet.

A markupt készítése közben a XamlPad programban feltűnik az IntelliSense hiánya. Ha azonban a Show Visual Tree gombra kattintunk, megnyílik az a felhasználati felület, amely a Visual Studio Properties ablakát utánozza, és segít elképzelni a XAML-markup struktúráját. Sajnálatos módon a Visual Tree ablak segítségével (pillanatnyilag) nem módosíthatjuk magát a XAML-t; csak a XAML-markup írásvédelet megtekintését teszi lehetővé.

Azt se feledjük, hogy a XamlPad segítségével jelenleg nem menthetjük a *.xaml fájlokat; a markupt automatikusan a XamlPad-Saved.xaml fájlba menti, és az eszközök követező betöltése során megjeleníti (azonban a markupt igény szerint bemásolhatjuk az aktuális alkalmazásba).

XAML-névterek és -kulcsszavak

Mint azt a fejezet korábbi példáiban láttuk, a WPF-központú XAML-fájl gyökereleme (például a <window>, a <page> vagy az <application> tag) rendszertínt két XML-névterre hivatkozik:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
  </Grid>
</Page>
```

Az első XML-névter, a http://schemas.microsoft.com/winfx/2006/xaml/presentation WPF-központú névttereket képez le, és az aktuális *.xaml fájl rendeltkezésére bocsátja azokat (system.windows, system.windows.controls, system.windows.data, system.windows.ink, system.windows.media, system.windows.navigation stb.). Ez az egy-többes leképezés gyakorlatilag hardkódolva van a WPF-szerelvényekben (windowssbase.dll, presentationcore.dll és presentationframework.dll) a szerelvény szintű [xmlnsdefinition] attribútum révén. Ha ezeket a WPF-szerelvényeket a reflector.exe programmal megnyitjuk, közvetlen betekintést nyerhetünk a leképezésekbe (lásd a 28.11. ábrát).

A második XML-névter, a `http://schemas.microsoft.com/winfx/2006/xaml/XAML-specifikus` kulcsszavakat, valamint a `system.windows.markup` névterben található típusok részhalmozát foglalja magában. Egy helyesen formázott XML-dokumentumnak definiálnia kell egy gyökércélelmet, amely elsődlegesen névterként egyetlen XML-névteret jelöl ki, amely tipikusan a leggyakrabban használt elemeket tartalmazó névter. Ha a gyökércélelme további másodlagos névtereket kell befoglalni (mint azt itt láttuk), a névtereket (a lehetséges névterközések feloldásának érdekében) egyedi prefix alklalmazásával kell definiálnunk. Megjegyezés szerint a prefix egyszerűen „x”-; azonban bármilyen egyedi tokent használhatunk, például a `xaml:specifictoken` is:

tanulási szerzetünk, mert ezt a részletesebb XML-névter prefixet szeretnénk al-

28. f

```
<Application XamlSpecified:Cstuf:Class="SimpleXamlApp.MyApp"
xmlns="http://schemas.microsoft.com/winfx/2006/presentation"
xmlns:XamlSpecified:Cstuf="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri="MainWindow.xaml" Exit="AppExit">
<[[CDATA[
private void AppExit(object sender, EventArgs e)
{
    MessageBox.Show("App has exited");
}
]]>
</XamlSpecified:Cstuf:Code>
</Application>
```

Mivel a XamlSpecifiedCstuf több billentyűleütést követel, maradjunk az x-nél. Mindenesetre a Class és a code kulcsszavakon túl az http://schemas.microsoft.com/winfx/2006/xaml XML-névter továbbí XAML-kulcsszavakat bo-csat rendelkezőnkre (és a System.Windows.Markup névter elemeit is). A 28.9. táblázat ezen alapvető kulcsszavakat mutatja be.

XAML-kulcsszó Jelentés

Array	.NET-tömbtípust képvisel a XAML-ben.
ClassModifier	Lehetővé teszi a Class kulcsszóval jelölt (belső vagy nyilvános) osztály láthatóságának meghatározását.
DynamicResource	Lehetővé teszi, hogy olyan WPF-erőforrásra hivatkozzunk, amelynek változásait ellenőrizni kell.
FieldModifier	Lehetővé teszi, hogy meghatározzuk egy típus tag (belső, nyilvános, privát, védett) láthatóságát a gyökér bármely megnevezett részlemre (például a <Window> elemen belül egy <Button> elem) számára. „Megnevezett elemet” a NameXAML kulcsszóval definiálunk.
Key	Lehetővé teszi, hogy kulcs értéket határozzunk meg egy XAML-elemhez – amelyet egy szótár elemben helyezünk el.
Name	Lehetővé teszi, hogy meghatározzuk adott XAML-elem generált C#-névét.
Null	Nullreferenciát jelöl.
Static	Lehetővé teszi, hogy egy típus statikus tagjára hivatkozzunk.
StaticResource	Lehetővé teszi, hogy olyan WPF-erőforrásra hivatkozzunk, amelynek változásait nem kell ellenőrizni.

XAML-kulcsszó jelentés	
Type	A C# typeof operátor XAML-megfelelője (a megadott név alapján System.Type osztályt hoz létre).
TypeArguments	Lehetővé teszi, hogy az elemet generikus típusként hozzunk létre a meghatározott típusparaméterrel (például a List<int> elemet szemben a List<bool> elemmel).

28.9. táblázat: XAML-kulcsszavak

Több említtett kulcsszót látunk majd működés közben. Egy egyszerű példán keresztül vizsgáljuk meg a következő XAML <window> definíciót, amely a ClassModifier és a FileModifier, valamint a Name és a ClassSuffix-eket tartalmazza (emlékezzünk rá, hogy a xamlpad.exe nem teszi lehetővé, hogy olyan XAML-kulcsszót használjunk, amelynek alkalmazása a kód fordítását vonja maga után, mint például a code, a FileModifier vagy a ClassModifier):

```
<!-- Ez az osztály most belső osztály lesz.
Kódfájl használata esetén a részleges osztályt is belső
osztályként kell definiálni! -->
<window x:class="MyWPFApp.MainWindow" x:ClassModifier="internal"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml">
<!-- Ez a gomb a *.g.cs fájlban nyilvános lesz -->
<Button x:Name="myButton" x:FileModifier="public">
    OK
</Button>
</window>
```

Alapértelmezés szerint a C#/XAML típusdefiníciók nyilvánosak, míg a tagok alapértelmezés szerint belsők. A XAML-definíciónk alapján azonban az eredményként kapott, automatikusan generált fájl belső osztálytípust tartalmaz egy nyilvános Button típussal:

```
internal partial class MainWindow : System.Windows.Window,
    System.Windows.Markup.IComponentConnector
{
    public System.Windows.Controls.Button myButton;
    ...
}
```

XAML-elemek és -attribútumok

Miután létrehoztuk a gyökérellemünket, és bevezettük a szükséges XML-névtereket, egy valószínűleg WPF-alkalmazásban a gyermek a paneltípusok egyike lesz, amely a felhasználói interfészt leíró további felhasználói felület-elemeket foglal magában. A következő fejezet részletesen foglalkozik ezekkel a paneltípusokkal, tehát most tétellezzük fel, hogy a <window> típusunk egyetlen button elemet foglal magában.

Mint azt a fejezet során láttuk, a XAML-elemek a meghatározott .NET-névterben osztályra vagy struktúra típusokra, a nyitó elem címke attribútumai pedig tulajdonságokra vagy a típus eseményeire képeződnek le (nem hivatalosan). Ezért, ha az alábbihoz hasonló kódot készítünk:

```
<window x:Class="MyWPFApp.MainWindow" x:ClassModifier="internal"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
<!-- Ez feltételezi, hogy van egy myButton_Click esemény
a kódfejünkben! -->
<Button x:Name="myButton" x:FieldModifier="public"
Height="50" Width="100" Click="myButton_Click">
    OK
</Button>
</window>
```

akkor gyakorlatilag a következő kóddal kiegészített button készíthetjük el:

```
Button myButton = new Button();
myButton.Height = 50;
myButton.Width = 100;
myButton.Content = "OK";
myButton.Click += new RoutedEventArgs(myButton_Click);
```

A fejezetben eddig látott példák után ez a leképezés egyértelmű; vizsgáljuk meg azonban a gomb tartalmának hozzárendelését. Emlékezzünk rá, hogy több WPF-vezérlőelem a contentcontrol ösztályból származik. Ennek köszönhetően tetszőleges számú belső elemet (például button, scrollbar) tartalmazhatnak. A content tulajdonságot úgy állítottuk be implicit módon, hogy az "OK" szöveget elhelyezzük a nyitó és a záró elemekben. Ha szükséges, explicit módon a következőképpen állíthatjuk be a content tulajdonságot:

Ezen a ponton a content tulajdonság implicit módon elvégzett beállítása puszta személyes preferencia kérdésének tűnhet. A történet izgalmasbba válik, ha elgondolkodunk azon, hogy a XAML segítségével hogyan alakítanánk a gombon tartalmát objektumra egyetlen egyszerű string helyett (grafikus renderelés, scrollbar, vagy textbox stb.). Mint a fejezet korábbi részében már említettük, a megoldás a tulajdonság-objektum szintaxis alkalmazása.

```
<Button x:Name="myButton"
        Height="50" Width="100" Content="OK">
</Button>
```

A XAML tulajdonság-objektum szintaxisa

A *tulajdonság-objektum szintaxis* lehetővé teszi, hogy összetett objektumokat rendeljünk egy tulajdonsághoz. Íme a „görgetőssáv a gombban” eset XAML-leírása, amely tulajdonság-objektum szintaxis segítségével állítja be a content tulajdonságot:

```
<Button x:Name="myButton" Height="100" Width="100">
  <Button.Content>
    <Scrollbar Height="50" Width="20"/>
  </Button.Content>
</Button>
```

Vegyünk észre, hogy ebben az esetben a <Button.Content> beágyazott elemben definiáltuk a scrollbar típust. A tulajdonság-objektum szintaxis mindig lebontható <TípusNév.TulajdonságNév> minítárra; jelen esetben nyilván a <Button> a típus, a content pedig a tulajdonság. A 28.12. ábra a xamlpad.exe programban látható kimenetet mutatja.

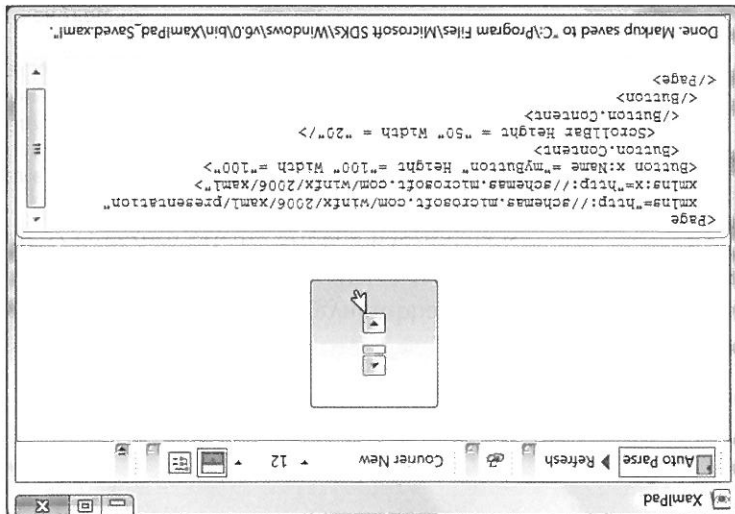
Emelkezzünk rá, hogy a contentControl-ból származtatott osztály gyermekelemével a rendszer automatikusan beállítja a content tulajdonságot, tehát a következő definíció is helyes:

```
<Button x:Name="myButton" Height="100" Width="100">
  <Scrollbar Height="50" Width="20"/>
</Button>
```

A tulajdonság-objektum szintaxis nem korlátozódik a content tulajdonság beállítására. Ezt a XAML-szintaxist bármikor alkalmazhatjuk, ha típus tulajdonság-objektumot kell beállítanunk. Vizsgáljuk meg például a gomb típus background tulajdonságát.

A tulajdonságot a WPF API-ban található bármely Brush típuson beállíthatjuk. Ha egyszerű ecsetípust szeretnénk használni, ehhez csak a következő markap szükséges, mivel a Brush-leszámazott típust (például `Background`) vagy tulajdonságokhoz rendelt sztringérték automatikusan ecset típusa alakul:

```
<!-- Itt a "Green" Brushes.Green típusra képeződik le -->
<Button x:Name="myButton" Height="100" Width="100"
        Background="Green">
    <Button.Content>
        <ScrollBar Height="50" Width="20"/>
    </Button.Content>
</Button>
```



28.12. ábra: A tulajdonság-leszámazás szintaxis lehetővé teszi, hogy összetett tulajdonságokat rendeljünk a tulajdonságokhoz

Há kifinomultabb esetben (például `LinearGradientBrush` esetére) van szükségünk, a név/érték szintaxis nem lesz elegendő. Figyelembe véve, hogy a `LinearGradientBrush` teljes osztálytípus, a tulajdonság-leszámazás szintaxis segítségével kell a típusnak a kezdeti értékeket átadnunk:

```
<Button x:Name="myButton" Height="100" Width="100">
    <Button.Content>
        <ScrollBar Height="50" Width="20"/>
    </Button.Content>
    <Button.Background>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
            <GradientStop Color="Blue" Offset="0"/>
            <GradientStop Color="Yellow" Offset="0.25"/>
        </LinearGradientBrush>
    </Button.Background>
```

```

<Button x:Name="myButton" Height="100" Width="100"
        Background="Pink">
</Button.Content>
<ScrollBar Height="50" Width="20"/>
</Button.Content>
</Button>

```

Ebben az esetben nem nyertünk semmit, inkább feleslegesen bonyolítottuk a folyamatot, miközben csak a következőket kellett volna betennünk:

```

<Button x:Name="myButton" Height="100" Width="100">
</Button.Content>
<ScrollBar Height="50" Width="20"/>
</Button.Content>
<Button.Background>
Pink
</Button.Background>
</Button>

```

Noha a tulajdonság- és szintaxis- elemekkel egyeztetett az egyszerű sztring értékek használata is: (például a `LinearGradientBrush` objektumot) rendelünk tulajdonság értékekhez, megengedett az egyszerű sztring értékek használata is:

28.13. ábra: Szokvány gomb típus



Pillanatnyilag ne foglalkozzunk a `LinearGradientBrush` típus konfigurálásával (a 30. fejezet a WPF grafikus rendszert vizsgálja). Egyes- más módon csak figyeljünk meg, hogy a tulajdonság- és szintaxis- elemekkel hata- rozunk meg a `Button` típus `Content` és `Background` tulajdonságait. A 28.13. áb- rán ezen különleges gomb rendszere látható.

```

<GradientStop Color="green" Offset="0.75" />
<GradientStop Color="red" Offset="0.50" />
</LinearGradientBrush>
</Button.Background>
</Button>

```


A XAML csatolt tulajdonságai

A tulajdonságkezelő szintaxison kívül a XAML definiál egy olyan szintaxist, amellyel *csatolt tulajdonságokat* határozhatunk meg. A csatolt tulajdonságok sokoldalúak, és egyik céljuk, hogy lehetővé tegyék a különböző gyermek-elemek számára egy olyan tulajdonság egyedi értékeinek meghatározását, amelyet valójában a szülőelemben definiáltunk. A csatolt tulajdonság-szintaxis alkalmazásának legelterjedtebb módja a felhasználófelület-elemek elhelyezése a WPF-paneltípusok (Grid, DockPanel stb.) valamelyikén. A következő fejezetben részletesen megvizsgáljuk ezeket a paneleket. Egyelőre tekintsük meg a csatolt tulajdonság-szintaxis következő példáját:

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <DockPanel LastChildIsTrue="true">
    <!-- Elemek dokkolása a panelhez csatolt tulajdonságok
        használatával -->
    <label DockPanel.Dock="Top" Name="lblInstruction">
      FontSize="15">Enter car information</label>
    <label DockPanel.Dock="Left" Name="lblMake">Make</label>
    <label DockPanel.Dock="Right" Name="lblColor">Color</label>
    <label DockPanel.Dock="Bottom" Name="lblPetName">Pet Name</label>
    <Button Name="btnOK">OK</Button>
  </DockPanel>
</Page>
```

A példában DockPanel típust definiáltunk, amely négy, a tárolóban dokkolt Label típust foglal magában. Vegyük észre, hogy a csatolt tulajdonság szintaxisa <SzülőTípus.SzülőTulajdonság> (például DockPanel.Dock). A Button típus nem határoz meg dokkolási területet; elfoglalja a DockPanel fennmaradó területét, mivel a nyitó <DockPanel> definícióban a LastChildIsTrue tulajdonság van beállítva. A csatolt tulajdonságokkal kapcsolatosan néhány dologgal tisztában kell lennünk. Nem univerzális szintaxisal állunk szemben, amelyet bármely szülő bármely elemén alkalmazhatunk. A következő XAML-definíció például nem elemezhető anélkül, hogy hibát ne kapnánk:

```
<!-- A Height tulajdonság beállítására Button objektumon csatolt
      tulajdonsággal -->
<Button Name="myButton" Width="100">
  <Button.Content>
    <ScrollBar Button.Height="100" Height="50" Width="20"/>
  </Button.Content>
```

```
<Button.Background>
    pink
</Button.Background>
</Button>
```

A csatolt tulajdonság valójában a WPF-specifikus fogalom, a *függetlenségű tulajdonság* speciális formáját képviseli. Röviden, a függetlenségű tulajdonságok lehetővé teszik, hogy egy mező értékeit több bemenet alapján számítsuk ki. A függetlenségű tulajdonságoknak, és emellett fogva a csatolt tulajdonságoknak is „regisztrálniuk” kell, hogy mely tulajdonságokat mely objektumok állíthatják be (ezt az előzőleg bemutatott `scrollbar/button` példában nem tettük meg).

A WPF a háttérben több technológia – például az adatkötés, a stílusok és témák és az animációs szolgáltatások – használatára esetén alkalmazza a függetlenségű tulajdonság-mechanizmust. A függetlenségű tulajdonság megvalósításával önirányító ellenőrzést, alapértelmezett értékeket és visszahívási mechanizmust biztosíthatunk, és lehetővé teszi, hogy futásidejű információk alapján állítsuk be a tulajdonságok értékét.

A függetlenségű tulajdonságokkal kapcsolatos furcsaság, hogy a függetlenségű tulajdonság értékeknek *beállítása* pontosan úgy néz ki, mint egy „normal”. NET-tulajdonság beállítása. Ezért az esetek többségében nem vesszük észre, hogy beállítottuk egy függetlenségű tulajdonság értékét.

Azonban a függetlenségű tulajdonságok megvalósítása a szintalak mögött való-
jában teljesen eltérő. A legtöbb WPF-alkalmazásunkban *nem* kell egyedi függő-
ségi tulajdonságokat készítenünk. Ez csak abban az esetben válhat gyakori fel-
adattá, ha egyedi WPF-vezérlőelemeket kell készítenünk. Ez a XAML megjele-
nésenek köszönhetően szintén nem számít mindennapos tevékenységnek.
A 29. fejezetben részletesen megvizsgáljuk a függetlenségű tulajdonságokat.

XAML-típusátalakítók

Gyakorlati szempontból, amikor értéket rendelünk az attribútumokhoz (pé-
ldául `background = "pink"`), vagy a nyitó és a záró elemek hatókörében impli-
cit módon beállítjuk a tartalmat (például `<button>ok</button>`), feltehetően nem
jűk, hogy az értékek sztringadatok. Ha azonban ezt jobban átgondoljuk, ráö-
hetünk, hogy ez nem minden esetben igaz. Gondoljunk például a `button fi-`
pus `background` tulajdonságának definíciójára (amelyet a `control` összesítálytól
örököltünk):

```
// A System.Windows.Controls.Control.Background tulajdonság.  
public Brush Background  
{  
    ...  
}
```

Mint látnuk, a tulajdonság a `system string` helyett a `brush` típust csomagolja be! Felmerül a kérdés, hogy (ebben az esetben) a „pink” értéket mi alakítja `solidcolorbrush` objektummá, amely a rózsaszínek megfelelő RGB értékekkel rendelkezik? Íme, egy másik példa, vizsgáljuk meg egy meghatározott méreteekkel rendelkező `Ellipsis` XAML-definióját:

```
<Ellipsis Fill = "Purple" Width = "100.5" Height = "87.4">  
</Ellipsis>
```

Ha megvizsgálunk az `Ellipsis` típus `width` és `height` tulajdonságait, láthatunk, hogy `prototípus`uk szerint `string`-ek helyett `double` típusú adatokkal dolgoznak.

A színeket megőrt a XAML-elemek különböző *típusátalakítók* segítségével alakítják a stringadatokat a megfelelő mögöttes objektummá. Ha például a „Pink” értéket olyan tulajdonsághoz rendeljük, amely `prototípus` szerint `ecset` típusokkal dolgozik, a rendszer a `colorconverter` és a `brushconverter` típusokat alkalmazza. Több átalakító létezik: a `sizeconverter` (az előző `Ellipsis` típus `width` és `height` tulajdonságainak beállítására), a `rectconverter`, a `vectorconverter` és így tovább.

A nevéől függetlenül az összes típusátalakító a `system.componentmodel.Typeconverter` összeszályból származik. Ez a típus több virtuális metódust definiál, például a `canconvertto()`, a `convertto()`, a `canconvertfrom()` és a `convertfrom()` metódusokat, amelyeket származtatott típusban felüldefiniálhatnak, és így gondoskodhatnak a mögöttes fordítástól.

Tulajdonképpen nem kell tudnunk, hogy *melyik* típusátalakító képes le a XAML-stringadatainkat a megfelelő mögöttes objektumra. Legalábbis elég megértőnünk, hogy az átalakítók a rendszer a háttérben a XAML-definíciók egy szerkesztésére használja.

Megjegyzés Noha maga a XAML viszonylag új technológia, a típusátalakítók koncepciója a .NET platform megjelenése óta létezik. A színeket megőrt a `Windows Forms` és a `GDI+` különböző átalakítókát használ. Például a `GDI+ System.Drawing` névtér a `FontConverter` típusátalakítót definiálja, amely a „Wingdings” stringet a `Wingdings` betűkészletet alkalmazó `Font` objektumra képezi le.

A XAML markupszabványok

A típusátalakítók érdekes szerkezetek abból a szempontból, hogy nincs fizikai bizonyíték a XAML szintén velük folytatott együttműködésnek. Ehelyett a típusátalakítók a rendszer a színtalálék mögött a *.xaml fájlt feldolgozás során alkalmazza. Ezzel szemben a XAML támogatja a *markupszabványok* is. A típusátalakítókhoz hasonlóan a markupszabványok lehetővé teszik, hogy egy egyszerű címkézési értéket futásidőben objektummá alakítsunk. A különbség azonban az, hogy a markupszabványok nagyon speciális XAML-szintaxisal rendelkeznek.

Mivel a típusátalakítók és a markupszabványok ugyanazt a célt szolgálják, felmerülhet a kérdés, hogy miért készíthetünk típusdefiniciókat két különböző módszerrel. A markupszabványok a típusátalakítókhoz hasonlóan működést biztosítanak, és lehetővé teszik, hogy a XAML nyelvet leegyszerűsítsük, új funkcionális bővítéssel.

A markupszabványok segítségével egy másik típuson a statikus tulajdonság visszaterjesztéséhez rendelhetjük egy tulajdonság értéket, címkézéssel adatmódot deklarálhatunk, vagy típusinformációkat gyűjthetünk. Valójában, a XAML-kulcsszavak egy részét használja (például az `Array`, a `Stack`, a `Typed` és a `Type`) ábrába bejuttatott markupszabványok bővítése. A típusátalakítókhoz hasonlóan a markupszabványok a rendszer belsőleg olyan osztályként ábrázolja, amely a markupszabványon osztható színtalálék (megkezdés szerint a markupszabványon osztható színtalálék nevében az `extension` előtagot használják).

Vizsgáljuk meg működés közben a markupszabványok Tételét fel, hogy a `Label` típusok készíthetők beállítjuk a `content` tulajdonságot, és a `System.Environment` statikus tagjaival így jelentünk meg információkat arról a számtalálékról, amelyen az alkalmazásunkat futtatjuk. Íme a teljes markupszabvány:

```
<Page
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:corlib="clr-namespace:System;assembly=mscorlib">
  <StackPanel>
    <Label Content="{x:Static corlib:Environment.MachineName}" />
    <Label Content="{x:Static corlib:Environment.OSVersion}" />
    <Label Content="{x:Static corlib:Environment.ProcessorCount}" />
  </StackPanel>
</Page>
```

Vegyünk észre, hogy a <Page> definíció új XML-névter deklarációval rendelkezik, amelyet a corlib névtérprefixummal látunk el (a prefixum neve bármely más XML-névter prefixumához hasonlóan tetszőleges). Az XML-névterhez rendelt érték azonban egyedi, mivel egy clr-namespaces nevű regisztrált tokent (amely lehetővé teszi, hogy a típus definíciót tartalmazó .NET-névterre mutassunk), valamint egy assembly nevű másik tokent (amely a névteret magában foglaló szerelvény „barátságos” nevét képviseli) használunk.

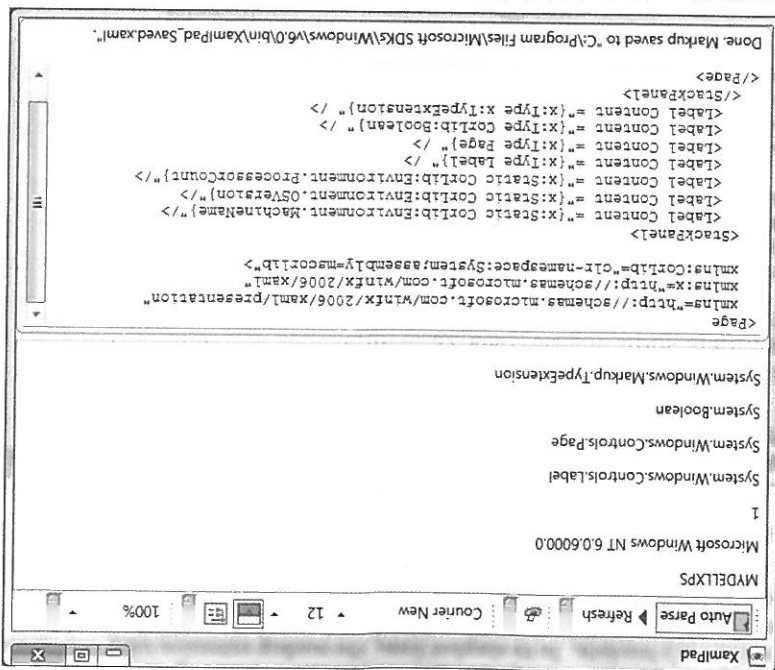
Az alkalmazott XML-névterekben figyeljük meg, hogy az egyes label típusok a static jelölő kiterjesztés segítségével az environment típus statikus tagját indítják. Mint láttuk, a markpübövíítőket mindig kapcsos zárójel fogja közre. Legegyeszerűbb formájában a jelölő kiterjesztés két értéket vesz fel: a jelölő kiterjesztés nevét (ebben az esetben static) az az érték követi, amelyet hozzárendelünk (például corlib.system.environment.OverVersion).

```
<!-- A "static" jelölő kiterjesztés segítségével a content
tulajdonságot a statikus tulajdonság értékre állítjuk -->
<label content="{x:static corlib:environment.OverVersion}" />
```

Íme, egy másik példa. Tétellezzük fel, hogy a különböző típusok teljesen meghatározott nevét szeretnénk megszerezni, hogy aztán az értékeket label típus sok content tulajdonságaihoz rendeljük. Ebben az esetben a type jelölő kiterjesztést alkalmazhatjuk:

```
<Page
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:corlib="clr-namespaces:system;assembly=mscorlib">
  <stackpanel>
    <label content="{x:static corlib:environment.MachineName}" />
    <label content="{x:static corlib:environment.OSVersion}" />
    <label content="{x:static corlib:environment.ProcessorCount}" />
    <label content="{x:type label}" />
    <label content="{x:type Page}" />
    <label content="{x:type Corlib:Boolean}" />
    <label content="{x:type x:TypeExtension}" />
  </stackpanel>
</Page>
```

A példában a WPF label típusának, buton típusának, valamint Boolean adat típusának az mscorlib.dll szerelvényben teljesen meghatározott nevét kapjuk, ráadásként pedig a type jelölő kiterjesztés teljesen meghatározott nevét. Ha az oldalt megtekintjük a xamlpad.exe segédprogramban, a 28.14. ábrához hasonló képet kapunk.



28.14. ábra: Markuppótvétők alkalmazása a tulajdonságokon statikus tagok értékeinek beállítására és típusinformációk megszerzésére

Az erőforrások és adatkötések előzetes áttekintése

A XAML-szintaxis bevezetésének összefoglalásaként az utolsó példában nemcsak az array jelölő kiterjesztés (amelyet az arrayextensio osztálytípus képvisel) használatát vizsgáljuk meg, de tanulmányozunk néhány egyszerű deklarativ adatkötést is, és áttekinthetjük a WPF-erőforrásokat. Az array jelölő kiterjesztés lehetővé teszi, hogy adattömböt rendeljünk egy meghatározott tulajdonsághoz. Amikor a XAML segítségével definiálunk ilyen tömböt, a type jelölő kiterjesztéssel határozzuk meg, hogy milyen típusú tömböt hozunk létre (sztring tömböt, bittérkép tömböt stb.). Vizsgáljuk meg a következő <Page> definíciót:

```
<Page>
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:corlib="clr-namespace:System;assembly=mscorlib"
  <StackPanel>
    <Label Content="{x:Array Type = Corlib:String}" />
  </StackPanel>
</Page>
```

Há a `xamlpad.exe` segítségével megtekintjük a rendelt markupt, a megtekintési panelben láthatjuk kitérva a `system.string[]` értéket. A kapcsolósárjelszintaxis alkalmazásával nem tudjuk adatokkal feltölteni a tömböt. Ehhez részlemek segítségével létre kell hoznunk a saját tömbünket, amelynek típusa megegyezik az adott tömb típusával. Vizsgáljuk meg a következő XAML-definíció részletét:

```
<x:Array Type="CorLib:String">
  <CorLib:String>Sun K'l Moon</CorLib:String>
  <CorLib:String>Red House Painters</CorLib:String>
  <CorLib:String>Besnard Lakes</CorLib:String>
</x:Array>
```

A példában sztringtömböt hoztunk létre. Az `<x:Array>` típus hatókörében három szöveges értéket adunk hozzá, majd lezárjuk a definíciót. Mivel ez szabályos XAML-markup, felmerülhet a kérdés, hogy hová helyezhetjük a tömbdeklarációnkat? Ha közvetlenül a `<Page>` elem hatókörébe helyezzük, akkor implicit módon beállítjuk a `<Page>` elem `content` tulajdonságát, és (ismét) láthatjuk a `system.string[]` értéket a `xamlpad.exe` megtekintési panelében (azonban nem ezt akartuk):

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:CorLib="clr-namespace:system;assembly=mscorlib">
  <!-- Hmm, éppen most állítottuk be a content tulajdonságot! -->
  <x:Array Type="CorLib:String">
    <CorLib:String>Sun K'l Moon</CorLib:String>
    <CorLib:String>Red House Painters</CorLib:String>
    <CorLib:String>Besnard Lakes</CorLib:String>
  </x:Array>
</Page>
```

Igazából az a célunk, hogy nevet adjunk a tömbnek, majd a markup valamely más pontján hivatkozzunk rá (például egy `ListBox` feltöltéséhez). Pontosán ezt tehetjük, ha a tömbünket *erőforrás elem*ben definiáljuk. Tiszta kell lenniünk azzal, hogy a WPF „erőforrások” nem mindig arra képesek lehetnek, mint amit rendszertől elvárunk (sztringtáblákra, ikonokra, bittérképekre stb.) A WPF-erőforrások segítségével bármelyik egyedi markupidarabot ábrázolhatunk, például a sztringtömbünket is (a WPF-erőforrásokkal kapcsolatban bővebb információkat a 30. fejezetben találunk).

Vizsgáljuk meg az utolsó <Page> definíciót, amely a key jelölő kiterjesztés segítségével a <stackpanel> elemhez adja a „goodmusic” sztringtömb-erőforrást. Ezt követően a staticresource jelölő kiterjesztés segítségével beállítjuk a listBox típus ItemsSource tulajdonságát a tömbünkre (vegyük észre, hogy ezúttal ugyanarra a kulcsra hivatkozunk):

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:corlib="clr-namespace:System;assembly=mscorlib">
  <StackPanel>
    <StackPanel.Resources>
      <x:Array Type="corlib:String" x:key = "GoodMusic">
        <corlib:String>Sun k'l Moon</corlib:String>
        <corlib:String>Red House Painters</corlib:String>
        <corlib:String>Besnard Lakes</corlib:String>
      </x:Array>
    </StackPanel.Resources>
    <Label Content="Really good music" />
    <ListBox Width="200"
      ItemsSource="{StaticResource GoodMusic}" />
  </StackPanel>
</Page>
```

Mint látjuk, a <stackpanel> hatókörében beágyazott <stackpanel.Resources> elemet helyezzük el. Ez az elem biztosít helyet a sztringtömbünk számára. A staticresource markuptövitők olyan erőforrásokat képviselnek, amelyek a kezdeti kötet követően nem változnak (innen a „státikus” erőforrás fogalma). Ha olyan erőforrással dolgozunk, amely az első kötés után módosul (például egy meghatározott rendszerszín), a másik jelölő kiterjesztést, a dynamicresource kiterjesztést használhatjuk. A 28.15. ábra a xamlpad.exe programban látható képet mutatja.

Megjegyzés A Key és a StaticResource markuptövitőket azért nem minősítettük az x prefixummal (az itt vizsgált többi jelölő kiterjesztéssel ellentétben), mert azokat a http://schemas.microsoft.com/winfx/2006/xaml/presentation gyökeir XML-névterében definiáltuk (mivel WPF-központúak).

Összefoglalva, eddig megvizsgáltunk több példát, amelyek bemutatták a XAML-szintaxis alapvető szempontjait. Nem vitás, hogy a XAML rendkívül érdekes technológia, amely lehetővé teszi, hogy deklaratív módon leírjuk .NET-objektumok hierarchiáját. Ez rendkívül hasznos a grafikus felhasználói

felületek konfigurálása során, de ne feledjük, hogy a XAML *bármely* szerelvény *bármely* típusát leírhatja, ha a típus az alapértelmezett konstruktort magában foglaló nem absztrakt típus.

Megjegyzés A XAML-t is feldolgozhatjuk futásidőben, erre a fejezet későbbi részében látunk példát.



28.15. ábra: Markuppbtörök, statikus erőforrások és egyszerű adatkötés

A WPF és a XAML-alapszintaxis bevezető áttekintésének végére értünk. A következő fejezet ezen információkra épül, és bemutatja a WPF-elrendezéskezelőket, valamint az általuk tartalmazott vezérlőelemet. Azonban mielőtt tovább haladnánk, térjünk ki a Visual Studio 2008 WPF-projekt sablonokra, és vizsgáljuk meg a Microsoft Expression Blend szerepét.

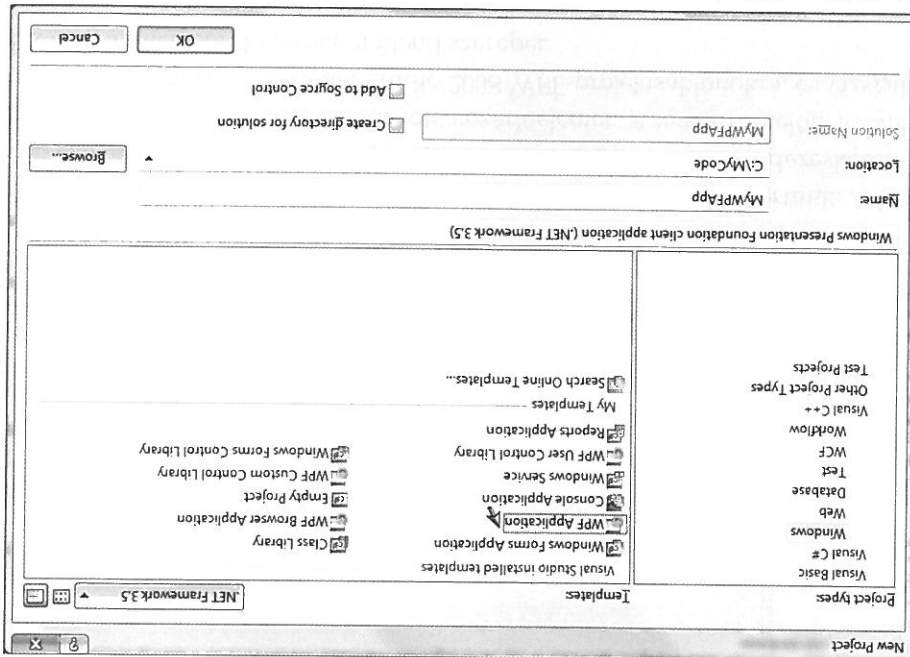
WPF-alkalmazások készítése a Visual Studio 2008 segítségével

A fejezet során egy szerű szövegszerkesztővel, a parancssoros fordítóval és a xamlpad.exe segítségével készítettük példáinkat. Ennek az volt az oka, hogy a WPF-alkalmazások alapszintaxisát vizsgáltuk, és nem akartuk, hogy egy gra-

Élőben tervező vonzó tulajdonságai eltereljük a figyelmünket. Most, hogy látjuk, hogyan lehet nyers WPF-alkalmazásokat készíteni, vizsgáljuk meg, a Visual Studio 2008 hogyan egyszerűsítheti a WPF-alkalmazások fejlesztését.

WPF-projektsablonok

A Visual Studio 2008 New Project párbeszédablaka WPF-központú projektmunkaterületet definiál, amelyek mindegyikét a Visual C# gyűjtemény window csomópontja foglal magában. Mint azt a 28.16. ábrán láthatjuk, a WPF Application, a WPF User Control Library, a WPF Custom Control Library és a WPF Browser Application (például XBAP) sablonok közül választhatunk.

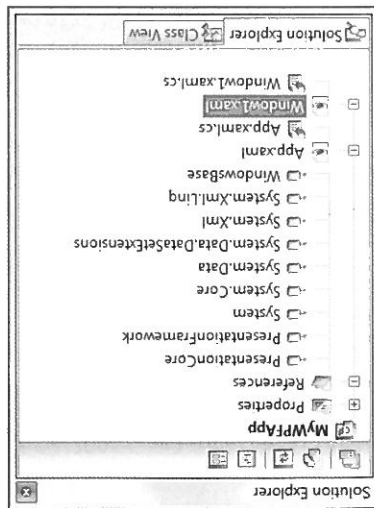


28.16. ábra: A Visual Studio 2008 WPF projektsablonjai

Amikor WPF asztali alkalmazást készítünk, a WPF Application projekt típust kell választanunk. Ekkor beállítódna az egyes WPF-szerelvények (a PresentationCore.dll, a PresentationFoundation.dll és a WindowsBase.dll) hivatkozásai, és a kódfájlok, illetve a XAML-definíciók alapján rendelkezésünkre bocsátja a studio a window és az Application típusokból származtatott típusokat (lásd a 28.17. ábrát).

A kezdeti ablak nevének módosítása

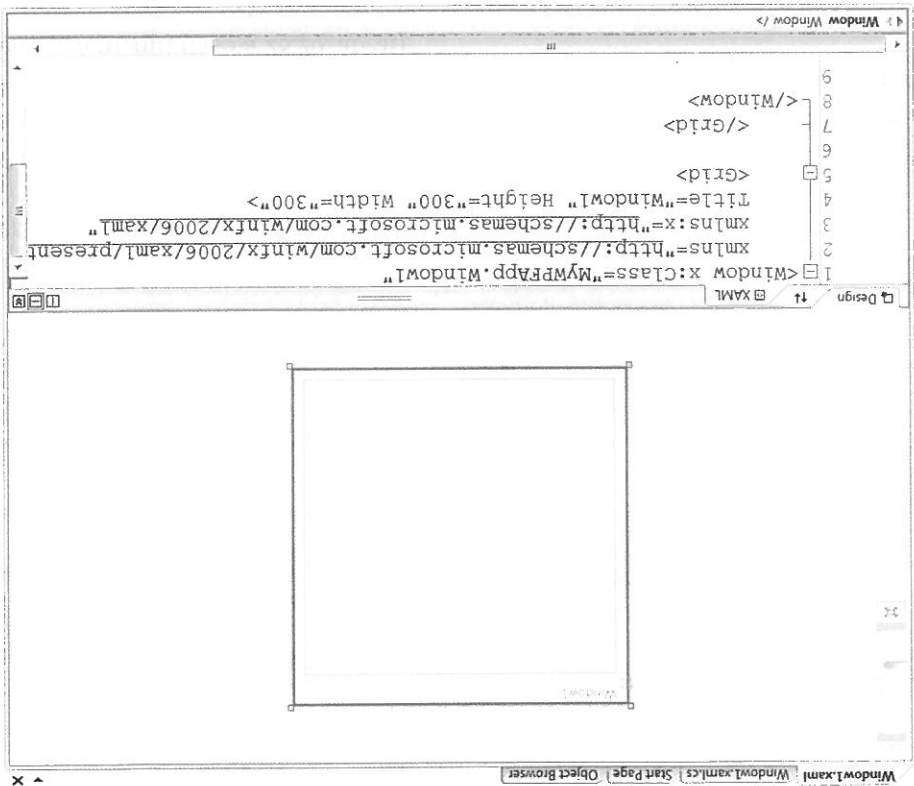
28.17. ábra: A WPF Application projekt típus kezdeti fájjai



Termékszíntű projekteket egészen biztosan szeretnénk más nevet adni a Window-leszármazott típusunknak (és a fájlnak, amely a típust definiálja), hogy a Window1 alapértelmezett név helyett hozzáillőbb nevet kapjon. A WPF-alkalmazás számára szükséges összes mozgó részt figyelembe véve, ez jóval bonyolultabb, mint az első pillantásra tűnhet. Íme a folyamat részletes ismertetése.

Ha először a jobb gombbal a kezdeti Window1.xaml fájlt nevére kattintunk, majd a Rename menüelemet választjuk, örömmel láthatjuk, hogy a rendszer a kapcsolódó Window1.xaml.cs fájlt is átnevezte a választásunknak megfelelően (például MainWindow.xaml-re). Azonban a *.xaml.cs fájlban az osztálytípus neve továbbra is Window1 marad. Ha a jobb gombbal a *.xaml.cs fájlban az osztály nevére kattintunk, és kiválasztjuk a Refactor > Rename lehetőséget, akkor alkalmasabb nevet (MainWindow) választhatunk a típus számára. Ha ezt követően megpróbáljuk futtatni a programunkat, az eredmény futásidejű kivétel lesz.

Ennek egyik oka az, hogy a nyitó <Window> elem class attribútuma még mindig az eredeti Window1 osztállyévre hivatkozik, és ezt az új osztállyévének megfelelően módosítanunk kell:



28.18. ábra: A WPF-tervező

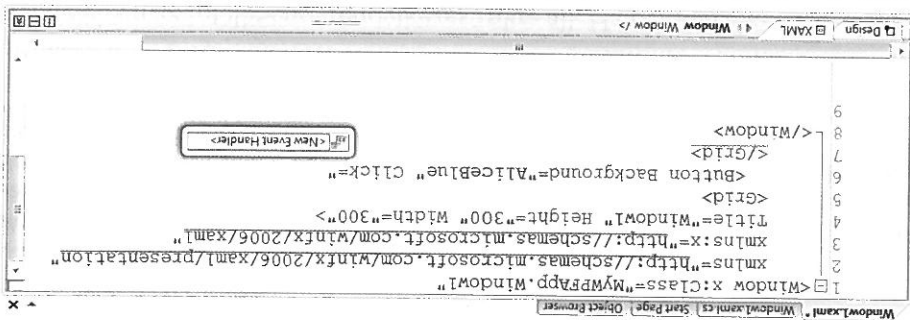
Megjegyzés A vizuális tervező megjelenítési paneljeit az osztott ablak beágyazott gombjai-val – például a Swap Panes gomb (amelyet a felfelé/lefelé nyílak jeleznek), a Horizontál és a Vertikal osztógombok stb. – átrendezhetjük. Szánjunk rá pár percet, és alakítsuk ki a szá-munkra legkényelmesebb konfigurációt.

Amikor a XAML-panelben XAML-markupot készítünk, várakozásainknak megfelelően az IntelliSense rendelkezésünkre áll. Ha például a kezdeti <Grid> típus hatókörén belül button deklarációt kezdünk írni, a képernyőn megjelenik azon tulajdonságok és események listája, amelyeket a típus támogat. Továbbá, amikor kiválasztunk egy tulajdonság tagot, a 28.19. ábrán látható módon a lehetséges értékek listája jelenik meg.

A Windows Formstól eltérően a WPF-alkalmazásokban az események kezelése *nem* úgy történik, hogy a Properties ablakban a villám ikonnal rendelkező gombra kattintunk (valójában ez a gomb a WPF-alkalmazások fejlesztése során nem áll rendelkezésünkre).

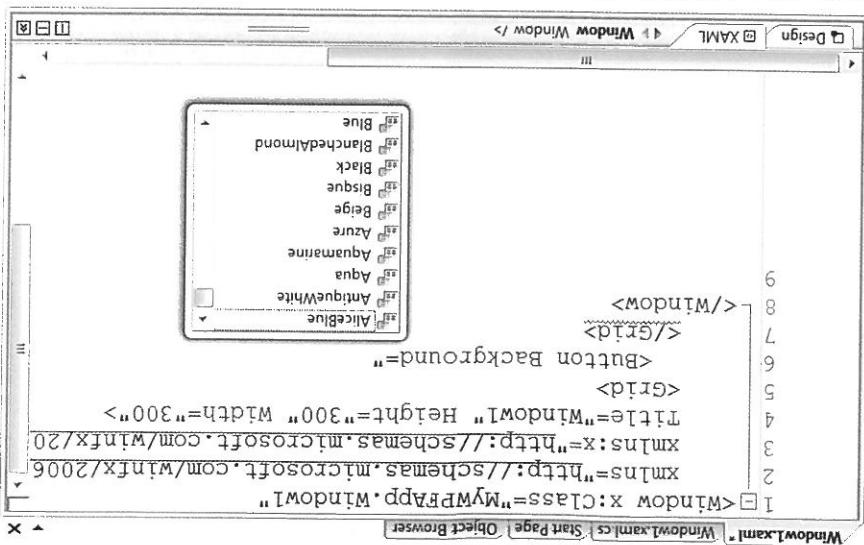
Ha kézzel írjuk be egy esemény nevet (idézőjelek között – ahogyan a XAML megköveteli), meghatározhatunk egy tetszőleges metódusnevet. Ha inkább azt szeretnénk, hogy az IDE alapértelmezett nevet generáljon (amely *Vezélő-Neve_EseményNev* formátumú), kattintsunk kétszer a <New EventHandler> elöugró menüelemre! Mindkét esetben az IDE a forráskódhoz adja a megfelelő eseménykezelőt:

28.20. ábra: Események kezelése a vizuális tervező segítségével



Amikor a WPF-vezérlőeseményeit kezeljük, a megszokott C#-szintaxis segítségével kézzel megírhatjuk a teljes forráskódot; ha azonban a XAML-szerkesztőbe beírjuk egy esemény nevet, azzal aktiváljuk a New Event Handler elöugró menüt (lásd a 28.20. ábrát).

28.19. ábra: A XAML IntelliSense



```
private void Button_Click(object sender, RoutedEventArgs e)
{
}
```

Megjegyzés Emlékezzünk rá, hogy ha vezérlőelem típusú tagváltozót szeretnénk definiálni az IDE-vel, értéket kell rendelnünk a Name tulajdonsághoz. Ha névtelen vezérlőelemek eseményeit kezeljük, az eseménykezelő neve egyszerűen *VezérlőNeve_EseményNeve[Sortszám]* (például *Button_Click*, *Button_Click_1*, *Button_Click_2* stb.).

Miután megvizsgáltuk a Visual Studio 2008 alapvető eszközeit, amelyekkel WPF-alkalmazásokat kezelhetünk, készítsünk példaprogramot az IDE segítségével, amely bemutatja a XAML futásidőben megvalósított elemzésének folyamatait.

XAML feldolgozása futásidőben: a SimpleXamlPad.exe

A WPF API támogatja a XAML-leírások programozott módon végrehajtott betöltését, elemzését és mentését. Ez a lehetőség sokszor rendkívül hasznosnak bizonyulhat. Tételizzuk fel például, hogy öt különböző XAML-fájlnak van, amelyek egy window típus megjelölését és funkcionálisát írják le. Feltevé, hogy minden vezérlőelem (és a szükséges eseménykezelők) neve minden fájlban megegyezik, az ablakon (az alkalmazásnak átadott indítási argumentum alapján) dinamikusan alkalmazhatunk „grafikus arculatokat” (skineket). A XAML-lel futásidőben folytatott kommunikáció a *XamlReader* és a *XamlWriter* típusokra épül, amelyek mindegyikét a *System.Windows.Markup* névtér definiálja. Annak illusztrálására, hogy programozott módon hogyan nyerhetünk ki egy window objektumot külső *.xaml fájlból, létrehozunk egy WPF-alkalmazás projektet (*SimpleXamlPad* néven), amely a fejezet korábbi részében vizsgált *xamlpad.exe* alapműködését utánozza.

Az alkalmazásunk egészen biztosan nem lesz olyan sokoldalú, mint a *xamlpad.exe*, de lehetővé teszi, hogy betrhassunk XAML-definíciókat, megtekinthessük az eredményeket és a XAML-kódot külső fájlba menthessük. Mitután a Visual Studio 2008 segítségével létrehoztuk a *SimpleXamlPad* projektet, a kiindulási ablakot nevezzük át *MainWindow*-ra (a korábban leírt módon), és a következőképpen módosítsuk a XAML-definíciót:

Megjegyzés A következő fejezet részletesen ismerteti a vezérlőelemeket és a paneleket, most ne nyugalankodjunk a vezérlőelem-deklarációk részletei miatt.

```

<window x:Class="SimpleXamlPad.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Simple Xaml Viewer" Height="338" Width="1041"
Loaded="Window_Loaded" Closed="Window_Closed"
WindowStartupLocation="CenterScreen"
>
<DockPanel LastChildIsTrue="True" >
<!-- Ez a gomb elindít egy ablakot a definiált XAML-fél -->
<Button DockPanel.Dock="Top" Name = "btnViewXaml" Width="100"
Height="40" Content = "View Xaml"
Click="btnViewXaml_Click" />
<!-- Ez a terület, ahová gépelhetünk -->
<TextBox AcceptsReturn="True" Name = "txtXamlData"
FontSize = "14" Background="Black" Foreground="Yellow"
BorderBrush = "Blue" VerticalScrollBarVisibility="Auto"
AcceptsTab="True">
</TextBox>
</DockPanel>
</window>

```

Elsőző is vagyunk észre, hogy a kezdeti <Grid> típust <DockPanel> típusra cseréltük, amely egy Button (btnViewXaml névvel) és egy TextBox (txtXamlData néven) típust foglal magában, és a kód kezeli a Button típus kattintási eseményét. Azt is figyeljük meg, hogy a Window Loaded és Closed eseményeit a nyitó <Window> elem kezeli. Ha a tervező segítségével kezeltük eseményeinket, a következő forráskódot találjuk a MainWindow.xaml.cs fájlunkban:

```

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void btnViewXaml_Click(object sender, RoutedEventArgs e)
    {
    }

    private void Window_Closed(object sender, EventArgs e)
    {
    }

    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
    }
}

```


Mielőtt folytatnánk, feltétlenül importáljuk a `MainWindow.xaml.cs` fájlunkba a következő névtéreket:

```
using System.IO;
using System.Windows.Markup;
```

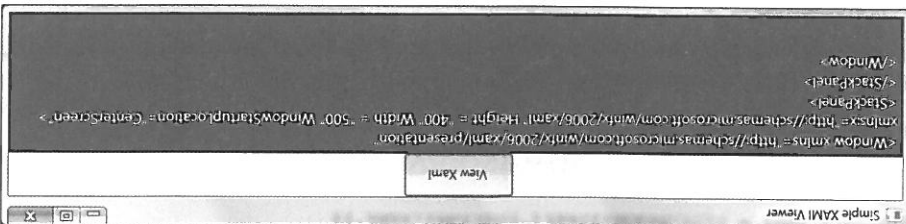
A Loaded esemény megvalósítása

A főablak `loaded` eseményének feladata annak meghatározása, hogy az alkalmazást tartalmazó könyvtárban pillanatnyilag megtalálható-e a `YourXaml.xaml` fájl. Ha ez a fájl létezik, beolvassuk az adatokat, és a rendszer azokat a főablakban egy `TextBox`-ba helyezi. Ha nem létezik, a `TextBox`-ot egy üres ablak alapélelmézzett XML-leírásával töltjük meg (ez a leírás pontosan ugyanaz a makup, mint egy kezdeti ablakdefiniáció, egyetlen különbség, hogy a `Window.Content` tulajdonságnak `[implicit]` beállításához a `<grid>` helyett a `<stackpanel>` elemet alkalmazunk).

Megjegyzés Az XML-névtérrel képviselő, éppen készülő sztringet nem könnyű feladat beírni a beágyazott idézőjelekhez szükséges vezérlőkarakterek miatt (példánk körületekintően!).

```
{
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Az alkalmazás főablakának betöltése során
    // helyezzünk el egy szerű XML-szöveget a szöveges blokkban!
    if (File.Exists(System.Environment.CurrentDirectory +
        "\\YourXaml.xaml"))
    {
        txXamlData.Text = File.ReadAllText("YourXaml.xaml");
    }
    txXamlData.Text =
        "<Window xmlns='http://schemas.microsoft.com/"
        + "/winfx/2006/xaml/presentation'"
        + " xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'"
        + " Height='400' Width='500'"
        + " WindowStartupLocation='CenterScreen'"
        + ">StackPanel>\n"
        + "</StackPanel>\n"
        + "</Window>";
}
}
```

A megközelítés segítségével a `SimpleXamlPad.exe` alkalmazás be tudja tölteni az előző munkamenet során létrehozott XAML-t, vagy szükség szerint megadja a markup alapértelmezett blokkját. Ezt követően futtathatjuk a programunkat, és a 28.21. ábrán látható képernyő jelenik meg a TextBoxban.



28.21. ábra: A `SimpleXamlPad.exe` első futtatása

A Button kattintási eseményének megvalósítása

Amikor a Button-ra kattintunk, először a `YourXaml.xaml` fájlba menjünk a `TextBox` aktuális adatait. Ekkor a `File.Open()` metódussal beolvassuk a rögzített adatokat, hogy megszerezzünk egy `Stream`-leszármazott típust. Erre azért van szükség, mert a `XamlReader.Load()` megköveteli, hogy az elemezni kívánt XAML-t a `Stream` (nem pedig egyszerű `System.String`) típusból származtatott típus képviselje.

Ha betöltöttük a létrehozni kívánt `<Window>` XAML-markupot, a memória-beli XAML alapján hozzuk létre a `System.Windows.Window` egy példányát, és modalis párbeszédablakként jelenítjük meg:

```
private void btnViewXaml_Click(object sender, RoutedEventArgs e)
{
    // A szöveges blokk adatainak kitérása helyi *.xaml fájlba.
    File.WriteAllText("YourXaml.xaml", txtXamlData.Text);

    // Ez az ablak dinamikusan XAML-kódolást kap.
    Window myWindow = null;

    try
    {
        // A helyi *.xaml fájl megnyitása.
        using (Stream sr = File.Open("YourXaml.xaml", FileMode.Open))
        {
            // A XAML összekapcsolása a Window objektummal.
            myWindow = (Window)XamlReader.Load(sr);
            myWindow.ShowDialog();
        }
    }
}
```

```
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

Vegyünk észre, hogy a logika jelentős részét try/catch blokkba csomagoljuk. Így módon, ha a yourXaml.xaml fájl helytelen markuptól tartalmaz, az eredmény-ként kapott üzenetdobozban láthatjuk a hibákat, amelyeket okoztunk.

A Closed esemény megvalósítása

Végül, a window típus Closed eseménye biztosítja, hogy a TextBox legfrissebb adatát rögzítse a yourXaml.xaml fájl:

```
private void window_Closed(object sender, EventArgs e)
{
    // A szöveges blokk adatainak kitírása helyi *.xaml fájlba.
    File.WriteAllText("yourXaml.xaml", txtXamlData.Text);
}
```

Az alkalmazás tesztelése

Most pedig indítsuk el a programunkat, és írjunk XAML-markuptól a szöveges területre. Ne feledjük, hogy ez a program (a xamlpad.exe programhoz hasonlóan) nem teszi lehetővé, hogy bármilyen kódgenerálással kapcsolatos XAML-attribútumot (például a class attribútumot vagy bármilyen eseménykezelőt) meghatározzunk. Tesztelés céljából írjuk be a következő XAML-markuptól a <stackPanel> hatókörébe:

```
<stackPanel>
<Rectangle Fill = "Green" Height = "40" Width = "200" />
<Button Content = "OK" Height = "40" Width = "100" />
<Label Content="{x:type Label}" />
</stackPanel>
```

Há a gombra kattintunk, feltűnik egy ablak, amely a XAML-definíciókat jeleníti meg (de lehet, hogy egy üzenetdobozban elemzési hibát kapunk – na-gyon figyelmesen gépeljünk!). A 28.22. ábra a lehetséges kimenetet mutatja. Nagyszerű! Egészen biztosan eszünkbe jut több lehetséges javítás az alkalmazással kapcsolatban, de ehhez tudnunk kell, hogyan dolgozzunk a WPF-vezérlőelemekkel és azokkal a panelekkel, amelyekben ezek vannak. Mielőtt a következő fejezetben megvizsgálalnánk a WPF-vezérlőelemmodellt, a fejezetet a Microsoft Expression Blend alkalmazás gyors áttekintésével zárjuk.

Miközben az új technológiák, mint például a XAML és a WPF, tanulmányozása sok fejlesztő számára izgalmas, kevesen lelkesebben a gondolatért, hogy ablakok, 3D képek, animációk és más összetevők leírásához több ezer sornyi markupt készítsunk. Az összetett entitások sokoldalú XAML-leírásának elkészítése még a Visual Studio 2008 segítségével is fáradszó és hibára hajlamosító folyamat. A Visual Studio 2008 jól felszerelt alkalmazás, amellyel forráskódot készíthetünk, és *finomhangolhatjuk* a XAML-leírások automatizálását szolgáló eszközök által generált XAML-definíciókat.

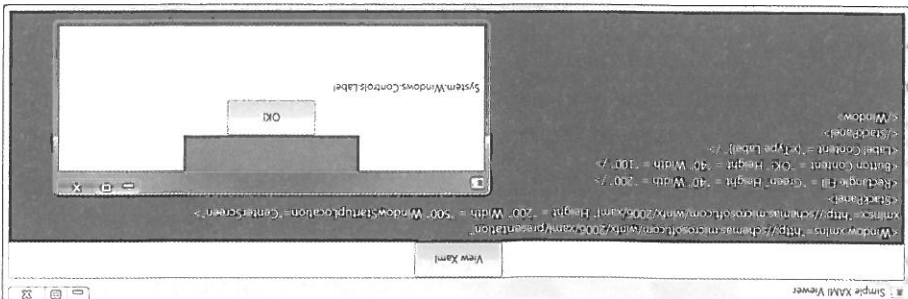
Emlékezzünk rá, hogy a WPF egyik legnagyobb előnye a kapcsolatok elkülönítése. Azonban a WPF nem pusztán a fájlok (például C#-kódfájlok és XAML-fájlok) szintjén alkalmazza a kapcsolatokat elkülönítést. Valójában a WPF-alkalmazás tízszerte tartja a kapcsolatokat elkülönítést azon eszközök szintjén, amelyek segítségével az alkalmazásainkat készítjük. Ez fontos, mert a profi WPF-alkalmazások rendszerint megkövetelik, hogy lehetőségük legyen grafikus-művész szolgáltatásait végük igénybe az alkalmazás megjelenésének és funkcionálitásának megfelelő kialakításához. A nem technikai beállítottságú emberek nem szívesen használják a Visual Studio 2008 alkalmazást XAML írásához.

Ezen problémák megoldására a Microsoft új termékcsaládot készített, amely az Expression-családba tartozik. Az Expression család tagjainak részletes ismeretét a <http://www.microsoft.com/expression/webhelyen/talajluk> címen, az Expression Blend a sokoldalú WPF front end alkalmazások fejlesztésére rendezkedett be.

A Microsoft Expression Blend szerepe

Forráskód A SimpleXamlPad projektet a forráskódkönyvtár 28. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xiv. oldalát.

28.22. ábra: A SimpleXamlPad.exe működés közben



Az Expression Blend előnyei

Az Expression Blend első fontos előnye, hogy a mód, amellyel a grafikusművész elkészíti a felhasználói felületet (nem egyezik meg, de) hasonló más multimédia-alkalmazásokhoz, például az Adobe Photoshop vagy a Macromedia Director programokhoz. Például az Expression Blend támogat az animációkhoz történetketekerek készítésére alkalmas eszközöket, szinkronizáló segédprogramokat, elrendezési és grafikus átalakító eszközöket és így tovább. Az Expression Blend továbbá biztosít olyan, a forráskód világához közel álló tulajdonságokat, mint az adatkötések létrehozásának támogatása vagy a triggerek. Ettől függetlenül egy grafikusművész rendkívül sokoldalú felhasználói felületet készíthet anélkül, hogy akár egyetlen sornyi XAML-markupot vagy C#-kódot látna. A 28.23. ábrán az Expression Blend működés közben látható.



28.23. ábra: Az Expression Blend a háttérben a felhasználó számára láthatatlannul generálja a XAML-markupot

Az Expression Blend másik fontos előnye, hogy ugyanazokat a projekt-munkaterületeket használja, mint a Visual Studio 2008! Ha a grafikusművész kiterjesztési felületet, a C#-fejlesztő megnyithatja ugyanazt a projektet, és belenyúlhat a forráskódba, hozzáadhat eseménykezelőket, finomhangolhatja a XAML-t és így tovább. A grafikusművész az Expression Blend eszközzel hasonlóképpen megnyithat létező Visual Studio 2008 WPF-projek-

teket, és kicsinostíthat egy jellegtelen front endet. Vagyis röviden, a WPF rendkívül rugalmas próbálkozás a kapcsolódó kódfrágók és a fejlesztőeszközök együttműködésének támogatására.

Bar az igaz, hogy Expression Blendhez hasonló eszköz használatainak ismerete többé-kevésbé kötelező, ha a WPF segítségével élvonalbeli és sokoldalú alkalmazásokat készítünk, a könyv jelen kiadása nem foglalkozik ennek részleteivel. Ezen könyv célja az, hogy megvizsgálja a WPF mögöttes programozási modelljét, de nem merül el műtűvészetelméleti részletekben. Ha mélyebben foglalkoztat bennünket a témakör, a Microsoft Expression család tagjainak próbaverzióját leölthetjük a támogatási webhelyről. Erdemes letölteni az Expression Blend próbaverzióját is, és megapaszalni, hogy az eszköz mire képes.

Összefoglalás

A Windows Presentation Foundation (WPF) egy olyan interfész-eszközrendszert, amely a .NET 3.0 verziójának megjelenése óta áll rendelkezésünkre. A WPF fő célja, hogy a korábban egymáshoz nem kapcsolódó asztali technológiákat (2D grafika, 3D grafika, ablak- és vezérlőelem-fejlesztés stb.) összeolvassza és egységesítse egyetlen programozási modellben. A WPF-programok rendszerint a XAML-t (Extensible Application Markup Language – bővíthető alkalmazás jelölőnyelv) alkalmazzzák, amely lehetővé teszi, hogy markap segítségével deklaráljuk WPF-elemünk megjelenését és funkcionálitását.

Mint azt a fejezetben láttuk, a XAML deklaratív szintaxisa segítségével leírhatjuk .NET-objektumaink hierarchiáját. A fejezetben a XAML tanulmányozásán sa során megismerkedtünk a szintaxis újabb elemeivel, például a tulajdonság-elem szintaxisával, a csatolt tulajdonságokkal, valamint a típusátalakítók és a XAML-markupbővítmények szerepével. A fejezet végén megvizsgáltuk, hogyan tudunk a.XamlReader és a.XamlWriter típusok segítségével programozott módon együttműködni a XAML-definíciókkal, bepillantást nyertünk a Visual Studio 2008 WPF-spectifikus funkcióiba, és röviden áttekintettük a Microsoft Expression Blend szerepét.

Programozás WPF-vezérlőelemekkel

Az előző fejezetben megismerkedtünk a WPF programozási modell alapjaival, beleértve az ablak- és alkalmazásstípusok vizsgálatát, valamint a XAML több jellegzetességét. Itt a WPF-vezérlőelemek csoportját mutatjuk be részletesen, és bővíthetjük jelenlegi ismereteinket. A fejezetet a beépített WPF-vezérlőelemek vizsgálatával kezdjük, ezt követi majd két jelentős, a vezérlőelemekhez kapcsolódó WPF-téma: a függőségi tulajdonságok és a továbbított események.

Ha a központi programozási modellen már túljutottunk, a fejezet további részében több érdekes példával illusztráljuk, hogyan használhatjuk a WPF-vezérlőelemeket alkalmazásainkban. Például megismerjük, hogyan rendezzük a vezérlőelemeket a különböző WPF-tárolókban (canvas, grid, stack-panel, wrappanel stb.), és hogyan hozzuk létre főablakot menürendszerrel, állapotsávvál és eszköztárral. A fejezet végén megvizsgáljuk a *vezérlőutasítást* sok használatát (amelyekkel beépített viselkedéseket adhatunk a különböző felhasználói felületi-elemekhez és bemeneti parancsokhoz), valamint megismerkedünk a WPF *adatkötési* modellel.

Megjegyzés Sok vezérlőelem XAML-definíciója „laza (hátértékd nélküli) XAML-fájlként” szerepel a leírt forráskódokban. A megjelölt kimenet megtekintéséhez az adott *.xaml fájlból átmásolhatjuk a markupt a 28. fejezetben létrehozott SimpleXamlPad.exe alkalmazásunkba. Vagy a <window> és </window> elemeket átirhatjuk <Page> és </Page> elemekre, majd a fájlt a kétszer rákattintva megtekinthetjük az Internet Explorerben.

A WPF vezérlőelem-könyvtár vizsgálata

Hacsak nem vagyunk teljesen járatlanok a grafikus felhasználói felület készítésében, a WPF-vezérlőelemek beépített készlete nem tartalmaz sok meglepő dolgot számunkra, függetlenül attól, hogy korábban melyik GUI-eszközrendszert használtuk (MFC, Java AWT/Swing, Windows Forms, VB 6.0, Mac OS X [Cocoa], GTK+/GTK# stb.). A 29.1. táblázatban láthatók a kapcsolódó működésük szerint csoportosított alapvető WPF-vezérlőelemek.

WPF-vezérlőelem	Példák tagokra	Jelentés
-----------------	----------------	----------

Alapvető felhasználói bemeneti vezérlőelemek	Button, RadioButton, ComboBox, CheckBox, Expander, ListBox, Slider, ToggleButton, TreeView, ContextMenu, ScrollBar, Slider, TabControl, TextBox, RepeatButton, RichTextBox, Label	Mint elvárható, a WPF a vezérlőelemek egész csapadójával rendelkezik a felhasználói interfész legbonyolultabb elemeinek felépítéséhez.
--	---	--

Ablakkereteket díszítő vezérlőelemek	Menu, Toolbar, StatusBar, ToolTip, ProgressBar	A felhasználói felületek ezen elemei a Window objektum keretét díszítik bemeneti eszközökkel (például Menu típusal) és felhasználói információk elemekkel (állapotsáv, eszközleírás stb.).
--------------------------------------	--	--

Media-vezérlőelemek	Image, MediaElement, SoundPlayerAction	Ezek támogatják az audio- / videólejátszást és képmegjelenítést.
Elrendezést szabályozó vezérlőelemek	Border, Canvas, DockPanel, Grid, Gridview, GroupBox, Panel, StackPanel, Viewbox, WrapPanel	A WPF több olyan vezérlőelemet biztosít, amelyek segítségével a többi vezérlőelemet csoportosíthatjuk és rendszerezhetjük a megfélelőbb elrendezés érdekében.

29.1. táblázat: Alapvető WPF-vezérlőelemek

A 29.1. táblázatban bemutatott GUI-típusokon túl a WPF további vezérlőelemeket biztosít speciális dokumentumfeldolgozáshoz (documentviewer, flowdocument-reader stb.), valamint a Tablet PC fejlesztésekhez hasznos) Ink API-t támogató típusokat és különböző rögzített párbeszédablakokat (passwordbox, printdialog, fildialog, openfildialog és savefildialog) is rendelkezésünkre bocsájt.

Megjegyzés A fildialog, az openfildialog és a savefildialog típusokat a PresentationFramework.dll szerezlelvény Microsoft.Windows névtérre definiálja.

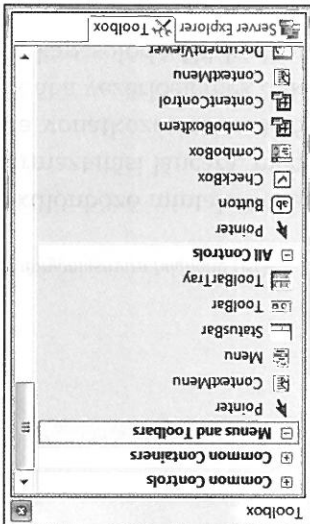
Ha Windows Forms tapasztalatokkal kezdünk a WPF tanulmányozásába, fel-tűnhet, hogy a beépített vezérlőelemek jelenlegi kínálata valamilyen szegénye-sebb, mint a Windows Forms esetében (például a WPF nem rendelkezik „lép-tetőgomb” – spin button – vezérlőelemekkel). A jó hír az, hogy sok ilyen hi-ányzó vezérlőelem gyorsan kifejezhető a XAML segítségével, sőt, a projektek közötti újrafelhasználás érdekében akár még felhasználói vagy egyedi vezér-lőelemként is modellezhető.

Megjegyzés A könyv jelen kiadása nem tér ki az egyedi WPF felhasználói vezérlőelemekre és WPF vezérlőelem-könyvtárakra. Ha szeretnénk többet megtudni a témakörrel, további infor-mációért lapozzuk fel a .NET Framework 3.5 SDK dokumentációját.

A WPF-vezérlőelemek és a Visual Studio 2008

Ha Visual Studio 2008 segítségével hozunk létre új WPF-alkalmazás projektet (lásd az előző fejezet), a vezérlőelemek többségét a 29.1. ábrán látható mó-don (kategóriánként csoportosítva) megtaláljuk a Toolbox menüben.

A Windows Forms projektekhez hasonlóan ezeket a vezérlőelemeket át-húzzhatjuk a vizuális tervezőbe, és a Properties ablakban konfigurálhatjuk őket. Továbbá emlékezzünk vissza a 28. fejezetből, hogy ha az eseményeket a XAML-szerkesztő segítségével kezeljük, az integrált fejlesztői környezet au-tomatikusan generálja a megfelelő eseménykezelőt a kódfájljainkban.

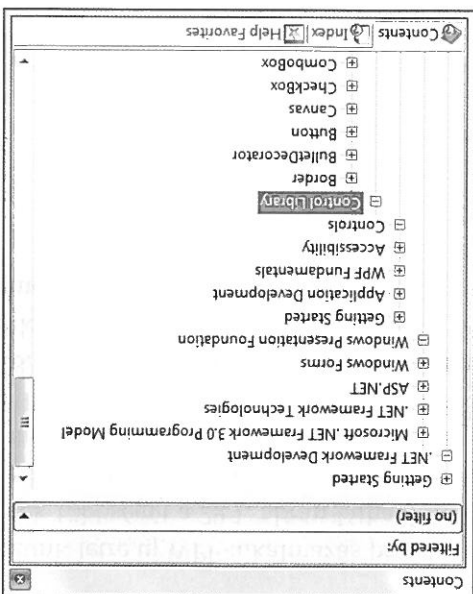


29.1. ábra: A Visual Studio 2008 Toolbox mutatja a beépített WPF-vezérlőelemeket

A részletek a dokumentációban található

A várakozásainkkal ellentétben ennek a fejezetnek *nem* az a célja, hogy végigvezessen minden egyes WPF-vezérlőelem minden egyes tagján. Inkább az alapvető vezérlőelemekről nyújtunk átfogó képet, különös hangsúlyt fektetve azokra az alapul szolgáló programozási modellekre (függetlenség, tulajdonságok, továbbított események, paramcsok stb.) és kulcsfontosságú szolgáltatásokra, amelyek a legtöbb WPF-vezérlőelem esetén azonosak.

Az egyes vezérlőelemek konkrét működésére vonatkozó ismereteinket a .NET Framework 3.5 SDK dokumentációból melyíthetjük el – konkrétan a sugórendszer „Control Library” szakaszból, amely a .NET Framework Development > Windows Presentation Foundation > Controls útvonalon érhető el (lásd a 29.2. ábrát).



29.2. ábra: Az egyes WPF-vezérlőelemek részletes leírása csak egy gombnyomásra található (F1)

Itt megtaláljuk az összes vezérlőelem teljes leírását, különböző mintakódokat (XAML és C# nyelven), valamint a vezérlőelem származtatási láncára, megvalósított interfészeire és alkalmazott attribútumaira vonatkozó információkat. A kis helyreigazítást követően gyorsan tekinthetjük át a vezérlőelemek leírását és konfigurálását a XAML-ben, valamint a kapcsolódó C#-kódban való használatát.

Vezérlőelemek deklarálása a XAML-ben

Hosszú évek során a fejlesztőket arra tanították, hogy a vezérlőelemeket viszonylag rögzített és kiszámítható elemekként kezeljék. Például a `Label` szövegnek mindig szöveges tartalma van, és szinte soha nincs látható keretük (bár lehetne). A gombok szöveges tartalommal rendelkező szűrke téglalapok, és időnként akár beágyazott képekkel is rendelkezhetnek. Amikor egy `Page`-t azt igényelte, hogy a „standard” eszközöket (például `Button` típust) egyedi `div`-t együk (például kör alakú képként megjelenő `Button` vezérlőelem), a fejlesztők gyakran arra kényszerültek, hogy az egyedi vezérlőelemet külön kódolással állítsák elő.

A WPF alapjaiban változtatja meg a vezérlőelemekről alkotott nézetünket. Nemcsak arra van lehetőségünk, hogy a jelölőkkel kifejezzük a vezérlőelem megjelenését és használatát, de nagyon sok WPF-vezérlőelem (különösen a `ContentControl`-t) úgy készült, hogy tetszőleges *templom* rendelkezzhessen. Emelkezünk vissza a 28. fejezetből arra, hogy a `ContentControl` tulajdonságot beállíthatjuk `mind explicit` (az elemek nyitó címkéjében, attribútumként), `mind pedig implicit módon`, ha a gyökér `gyermek`elemeként megadjuk a beágyazott tartalmat.

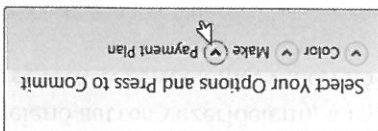
Tételezzük fel, hogy létrehozunk egy `ControlReview` nevű új `Visual Studio` 2008 WPF alkalmazás projektet.

Ahelyett, hogy felteleleznénk, hogy „minden `Button` vezérlőelem egy szűrke téglalap, amely szöveggel és esetleg képpel rendelkezik”, a XAML segítségével a következő implicit tartalmat adhatjuk meg a `Button` típushoz (feltelelve, hogy a következőket a kezdeti `<window>` elem `<grid>` elemén belül deklaráltuk):

```
<!-- Egyedi gomb beépített választási lehetőségekkel! -->
<Button Name="btnPurchaseOptions" Height="100" Width="300">
<StackPanel>
<Label Name="lblInstructions" Foreground="DarkGreen"
Content="Select Your Options and Press to Commit"/>
<StackPanel Orientation="Horizontal">
<Expander Name="colorExpander" Header="Color">
<!-- Tételezzük fel, hogy itt szerepelnek az elemek... -->
</Expander>
<Expander Name="makeExpander" Header="Make">
<!-- Tételezzük fel, hogy itt szerepelnek az elemek... -->
</Expander>
<Expander Name="paymentExpander" Header="Payment Plan">
<!-- Tételezzük fel, hogy itt szerepelnek az elemek... -->
</Expander>
```

```
</StackPanel>
</StackPanel>
</Button>
```

Vegyük észre, hogy ez a `<Button>` típus három `<expander>` típust tartalmaz (szinten a fejezet későbbi részében tárgyaljuk) egy halmozásban rendeztünk el. Mielőtt nagyon belemélyednénk az egyes vezérlők működésébe, nézzük meg a 29.3. ábrát, amelyen a megjelenített kimenet látható.



29.3. ábra: Egyedi Button vezérlőelem XAML-deklarációja

Egyszerű összehasonlításként nézzük meg, hogy ugyanazt a vezérlőelemet hogyan építettük volna fel Windows Forms használatával. Ebben az API-ban ilyen vezérlőelemet csakis úgy hozhattunk volna létre, ha építettünk egy `Button` típusból származtatott egyedi típust, amely manuálisan kezeli a grafikus tartalmak megjelenítését, módosítja a belső vezérlőelemek gyűjteményét, felülír bizonyos eseménykezelőket, stb.

Az asztali alkalmazások leíró nyelveinek megszűlésével az egyedi WPF-vezérlőelemek létrehozásának egyetlen nyomós oka az, ha egyedi viselkedéseket (események, virtuális metódusok felülírása, további interfésztípusok támogatása stb.) támogató vezérlőre van szükségünk, vagy mindenképpen egyedi tervezési idejű konfigurációs segédprogramokat kell támogatnunk. Ha csupán egyedi rendterelést létrehozásával kell törödnünk, a XAML is megteszi.

Együttműködés a vezérlőelemekkel a forráskódfájlokban

Emlékezzünk vissza az előző fejezetből, hogy a WPF típus tulajdonságait attritűtumokkal állíthatjuk be az elem nyitó címkéjén belül (vagy a tulajdonság-elem szintaxis alkalmazásával). Az esetek többségében a XAML-elemek attritűtumai közvetlenül a vezérlőelem osztályreprezentációjának tulajdonságaira és eseményeire képezhetők le a `system.windows.controls` névtérben. Eppen ezért mindig lehetőségünk van a vezérlőelemeket csak teljesen a jelölőben, vagy teljesen a kódban definiálni, esetleg a kettő kombinációját alkalmazni.

Megjegyzés A vezérlőelemhez a kapcsolódó kódfrájlbán kizárólag akkor van közvetlen hozzáférésünk, ha a XAML-definíció nyitó elemében a Name attribútum segítségével deklaráltuk.

Mivel az előző XAML-felolvasó olyan típusokat tartalmaz, amelyekhez hozzárendeltük a Name attribútumot, a típusokhoz közvetlen hozzáférésünk van a kódfrájlbán, és szabadon kezelhetünk deklarált eseményeket is. Például a Label vezérlőelem FontSize tulajdonságának értékét a következőképpen módosíthatjuk:

```
public partial class MainWindow : System.Windows.Window
{
    public MainWindow()
    {
        InitializeComponent();
        // A Label elem FontSize tulajdonságának módosítása.
        lblInstructions.FontSize = 14;
    }
}
```

Ez azért lehetséges, mert a XAML-definíciójukban Name attribútummal rendelkező vezérlőelemek tagválattozóként szerepelnek az automatikusan generált *.g.cs fájlban (lásd a 28. fejezetet):

```
public partial class MainWindow : System.Windows.Window,
    System.Windows.Markup.IComponentConnector
{
    //XAML-markup alapján definiált tagválattozók.
    internal System.Windows.Controls.Button btnPurchaseOptions;
    internal System.Windows.Controls.Label lblInstructions;
    internal System.Windows.Controls.Expander colorExpander;
    internal System.Windows.Controls.Expander makeExpander;
    internal System.Windows.Controls.Expander paymentExpander;
    ...
}
```

Amikor egy adott vezérlőelem eseményeit kívánjuk kezelni, a XAML-definícióban a következőképpen rendelhetünk metódusnevet az adott eseményhez:

```
<Button Name="btnPurchaseOptions"
        Click="btnPurchaseOptions_Click"
        Height="100" Width="300">
...
</Button>
```

A kapcsolódó kódfájl tartalmazza a metódus definícióját, amelynek formátuma az alapul szolgáló metódusreferencián alapul (ismét emlékezzünk vissza arra, hogy a Visual Studio 2008 IDE automatikusan frissíti a kódfájljunkat):

```
private void btnPurchaseOptions_Click(object sender,
    RoutedEventArgs e)
{
    MessageBox.Show("Button has been clicked");
}
```

Fontos megjegyezni, hogy az eseményeket nyugodtan kezelhettük teljes mértékben a forráskódban. Ha például az előző kattintási esemény XAML-definícióját töröljük, a Window elem konstruktorát a következőképpen módosíthatnánk:

```
public MainWindow()
{
    InitializeComponent();

    // A label elem fontsize tulajdonságának módosítása.
    lblInstructions.FontSize = 14;

    // A gomb kattintási eseményének kezelése.
    btnPurchaseOptions.Click +=
        new RoutedEventArgsHandler(btnPurchaseOptions_Click);
}
```

Most, hogy az alapvető vezérlőelem-model még frissen el emlékeztetünkben, a következő feladatunk, hogy megvizsgáljuk a WPF vezérlőelem-model két fontos (de kissé bonyolult) jellemzőjét: a függőségi tulajdonságokat és a továbbított eseményeket. Bár ezeknek a fogalmaknak a részletei a napi WPF programozási feladataink során általában rejtve maradnak, minél inkább tisztában vagyunk ezekkel az alacsony szintű részletekkel, a későbbiekben annál felkészültebben vehetjük bele magunkat a bonyolultabb WPF programozási feladatokba.

Forráskód A ControlReview kódfájlokat a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A függőségi tulajdonságok szerepe

Mint sejtető, a Windows Presentation Foundation API-k a .NET-típusrend-szer minden egyes tagját (osztályok, struktúrák, interfészek, metódusre-n-csák, felsorolások) és minden egyes lehetséges típustagját (tulajdonságok, me-tódusok, események, konstans adatok/írásvédett mezők stb.) használják a megvalósításaikban. A WPF azonban bevezetett egy új programozási mecha-nizmust, amelynek neve *függőségi tulajdonság*.

Megjegyzés A függőségi tulajdonságok WPF-specifikus programozási szerkezetek. A mai na-pig egyetlen .NET programozási nyelvnek sincs saját szintaxisa a tulajdonságok ezen sajátos tí-pusának definiálására. Viszont a „propdp” C#-kódreszlet új függőségi tulajdonság vázát fogja generalni (a kódreszleteket lásd a 2. fejezetben).

A „normális” .NET-tulajdonságokhoz (amelyet a WPF-szakirodalom gyakran *CLR-tulajdonságoknak* nevez) hasonlóan a függőségi tulajdonságokat beállít-hatjuk deklaratív módon XAML segítségével, vagy programozott módon a kódfejletben. Ráadásul léteznek függőségi tulajdonságok (a CLR-tulajdonsá-gokhoz hasonlóan) adatmezők beágyazására, és konfigurálhatókat őket írás-védetként, csak írhatóként vagy írható-olvashatóként, és így tovább.

A helyzetet még érdekesebbé teszi, hogy a legtöbb esetben szerencsére fo-galmunk sincs arról, hogy CLR-tulajdonság helyett tulajdonképpen egy füg-gőségi tulajdonságot állítottunk be! Például a WPF-vezérlőelemek a Height és a Width tagjaikat a FrameworkElement elemtől, míg a Content tagot a Con-tentElement elemtől öröklik – ezek mind függőségi tulajdonságok:

```
<!-- Éppen most állítottunk be három függőségi tulajdonságot! -->
<Button Name = "btnMyButton" Height = "50" Width = "100"
        Content = "OK"/>
```

Ezeket a hasonlóságokat látva meglepőnek találhatjuk, hogy a WPF miért ve-zetett be új fogalmat egy általánosan használt elvre. A válasz a függőségi tu-lajdonság mögött megbúvó megvalósításban rejlik. Implementálásuk után a függőségi tulajdonságok számos hatékony szolgáltatást biztosítanak, ame-lyeket a különböző WPF-technológiák, többek között az adatkötés, az animá-ciók szolgáltatások, a témák és stílusok stb. használnak. Díóhéjban a függőségi tulajdonságok – a CLR-tulajdonságoknál megismert egyszerű adatbeágyazá-son túl – a következő előnyöket nyújtják:

- A függőségi tulajdonságok öröközhetnek értékeiket a szülőelem XAML-definíciójából.
- A függőségi elemek támogatják annak lehetőségét, hogy az értékeket külső típusokkal állítsuk be (emlékezzünk vissza a 28. fejezetből arra, hogy a csatolt tulajdonságok éppen ezt teszik, mivel a csatolt tulajdonságok a függőségi tulajdonságokra épülnek).
- A függőségi tulajdonságok lehetővé teszik a WPF számára az érték kiszámítását több külső érték alapján.
- A függőségi tulajdonságok teremtik meg a visszahívási értesítések és triggerek (elegendő gyakorlati animációk, stílusok és témák létrehozása során) számára szükséges infrastruktúrát.
- A függőségi tulajdonságok lehetővé teszik az adataik statikus tárolását (ezzel csökkenthető a memóriafelhasználás).

A függőségi tulajdonságok egyik legjelentősebb elterjedése, hogy lehetővé teszik a WPF számára az érték kiszámítását több tulajdonságbeáramlás-érték alapján. Az egyéb kérdéses tulajdonságok közé tartozhatnak az operációs rendszer tulajdonságai (beleértve a rendszerszintű felhasználói beállításokat), az adatkötesen és az animáción/forगतकोंव्य-लोगिकान, az erőforrásokon és a stílusokon alapuló értékek, vagy más XAML-elemekkel való szülő/gyermek kapcsolat alapján ismert értékek.

Egy másik jelentős elterjedés az, hogy a függőségi tulajdonságokat beállíthatjuk a tulajdonság értékátvitelének megfigyelésére, és ez alapján külső műveletek végrehajtásának kikényszerítésére. Például egy függőségi tulajdonság értékének változása kiválthatja azt, hogy a WPF megváltoztassa a Window objektumban a vezérlőelemek elrendezését, újrakössön külső adatforrásokhoz, vagy végighaladjon egy egyedi animáció lépésén.

Létező függőségi tulajdonság vizsgálata

A teljes igazsághoz hozzátartozik, hogy elég kicsi annak az esélye, hogy manuálisan kelljen függőségi tulajdonságot létrehozunk WPF-projektjeink során. Valójában kizárólag akkor kell ezt megtennünk, ha olyan egyedi WPF-vezérlőelemet hozunk létre, amelyhez aloszállyként már létező vezérlőelemet adunk, és így akarjuk annak viselkedését módosítani. Ebben az esetben, ha olyan tulajdonságot hozunk létre, amelynek WPF adatkötió motorral, témamotorral

vagy animációs motorral kell dolgoznia, vagy, ha a tulajdonságnak üzenetszó-rással közölnie kell, hogy megváltozott, a függőségi tulajdonság a megfelelő eszköz. Minden más esetben a normal CLR-tulajdonság is megteszi.

Bár ez teljesen igaz, azért nem árt tisztában lenni a függőségi tulajdonságok alapvető felépítésével, mivel ezáltal a WPF egyes „misztikus” tulajdonságai érthetőbbé válnak, és jobban megismerjük az alapul szolgáló WPF programozási modellt. A függőségi tulajdonság belső felépítésének bemutatására vizsgáljuk meg a következő C#-kódot, amely a `FrameworkElement` osztálytípus `Height` tulajdonságának megvalósítását mutatja be:

```
public class FrameworkElement : UIElement, IInputElement, IHaveResources
{
    ...
    // Figyeljük meg, hogy ez DependencyProperty
    // típusú statikus mező.
    public static readonly DependencyProperty HeightProperty;

    // A statikus DependencyProperty mezőt a statikus konstruktorban
    // hozzuk létre és „regisztráljuk”.
    static FrameworkElement()
    {
        HeightProperty = DependencyProperty.Register(
            "Height",
            typeof(double),
            typeof(FrameworkElement),
            new FrameworkPropertyMetadataOptions.AffectsMeasure,
            new PropertyChangedCallback(
                FrameworkElement.OnTransformity)),
            new ValueCallback(
                FrameworkElement.IsWidthHeightValueId));
    }

    // Vegyük észre, hogy a Height tulajdonságnak még vannak get/set
    // blokkjai. Viszont a megvalósítás az örökölt
    // GetValue()/SetValue() metódusokat használja.
    public double Height
    {
        get { return (double) base.GetValue(HeightProperty); }
        set { base.SetValue(HeightProperty, value); }
    }
}
```

Mint láthatjuk, a függőségi tulajdonságok jóval több kiegészítő logikát követelnek a tipikus CLR-tulajdonságoknál! Lássuk részletesen, hogy mi is történik: először is a függőségi tulajdonságokat a `System.Windows.DependencyProperty` osztálytípussal képviseljük, és szinte mindig publikus, statikus, írásvédelemmel

zöként deklaráljuk. Ne feledjük, hogy a függőségi tulajdonságok egyik előnye, hogy nem kötődnek közvetlenül az objektum példányához (ami a memóriatakarékoságot segíti), ezért használunk statikus adatokat.

A függőségi tulajdonságok regisztrálása

Mivel a függőségi tulajdonságokat statikusan deklaráljuk, a kezdőértékeiket a típus statikus konstruktorában kapják meg. Azonban az egyszerű numerikus mezőktől eltérően a `DependencyProperty` objektum indirekt módon jön létre a statikus `DependencyProperty.Register` metódus visszatérési értékének rögzítésével. Ezt a metódust többszörösen túlterheljük; ebben a példában a `Register` metódust a következőképpen hívjuk meg:

```
HeightProperty = DependencyProperty.Register(
    "Height",
    typeof(double),
    typeof(FrameworkElement),
    new FrameworkPropertyMetadata(1.0 / (double) 0.0,
        FrameworkPropertyMetadataOptions.AffectsMeasure,
        new PropertyChangedCallback(
            FrameworkElement.OnTransform),
        new ValueCallback(
            FrameworkElement.IsWidthHeightValid));
```

A `Register` első argumentuma az osztály azon „normál” CLR-tulajdonságának neve, amely a `DependencyProperty` mezőt használja (jelen esetben a `Height`), míg a második argumentum az alap adattípusra vonatkozó típusinformáció, amelyhez kötöttük (itt `double` típusú).

A harmadik argumentum annak az osztálynak a típusinformációját határozza meg, amelyhez a tulajdonság tartozik (esetünkben a `FrameworkElement`). Bár ez feleslegesnek tűnhet (hiszen a `HeightProperty` mezőt már definiáltuk a `FrameworkElement` osztályban), ez a WPF nagyon hasznos sajátossága, hiszen lehetővé teszi, hogy egy típus tulajdonságokat „csatoljon” egy másik típushoz (még akkor is, ha az osztálydefiníció lezárt).

Megjegyzés Emlékezzünk vissza, hogy a C# 2008 bővítő metódusai (lásd a 13. fejezetet) lehetővé teszik új tagok hozzáadását a lezárt típusokhoz. A bővítő metódusok jelentik a legkövetlenebb módszert a típusok bővítésére olyan új funkcionálisokkal, amelyeknek nem kell WPF-központú szolgalátásokban részt venniük (pl. animáció).

A `register()` módszernek átadott utolsó argumentumok azok, amelyek ténylegesen megadják a függőségi tulajdonságok sajátos izét. Itt tudjuk megadni a `FrameworkPropertyMetadata` objektumot, amely leírja annak részleteit, hogy a WPF hogyan kezelje a tulajdonságot visszahívási értesítések esetén (ha a tulajdonságnak értesítenie kell másokat az értéke változásakor), az értéket hogy hitelesítsük, valamint azokat a különféle lehetőségeket is itt adhatjuk meg (amelyeket a `FrameworkPropertyMetadataOptions` felsorolás képvisel), amelyek azt szabályozzák, hogy a szövegben forgo tulajdonság mire van hatással (dolgozik-e adatkötéssel, örököllhető-e, stb.).

Wrapper tulajdonság definiálása a DependencyProperty mezőhöz

Amint a `DependencyProperty` objektum konfigurálásának részleteiről gondoskodtunk egy statikus konstruktorban, az utolsó feladatunk, hogy a mezőt becsomagoljuk egy tipikus CLR-tulajdonságba (esetünkben a `Height`-ba). Vegyük azért észre, hogy a „set” blokkok nem csupán visszaadnak vagy beállítanak egy osztályszintű `double` tagváltozót, hanem ezt indirekt módon teszik a `System.Windows.DependencyObject` alaposztály `GetValue()` és `SetValue()` módszereinak alkalmazásával:

```
public double Height
{
    get { return (double) base.GetValue(HeightProperty); }
    set { base.SetValue(HeightProperty, value); }
```

Megjegyzés Az igazat megvallva, nem szükséges burkolt tulajdonságot létrehozni a `DependencyProperty` mezőhöz, ha a mező publikus, mivel statikusan is elérhetjük az örökölt `GetValue()/SetValue()` publikus metódusok hívása során. A gyakorlatban viszont a legtöbb függőségi tulajdonság rendelkezik barátságos burkolóval, mivel ez nagyon XAML-barát.

Most, hogy már láttuk, hogy a függőségi tulajdonságok hogyan épülnek fel a felszín alatt, tartsuk észben, hogy teljességgel lehetséges normál CLR-tulajdonságok használatát, amelyek ugyanazokat a szolgáltatásokat támogatják, mint a WPF függőségi tulajdonságok (értesítések, statikus memóriafoglalás stb.). Viszont ehhez nem kevés háttérkódolás szükséges, amelyet manuálisan kell létrehozunk, és sok helyen megismételünk. A beépített `DependencyProperty` típus (és további infrastruktúradarabok) alkalmazásával ugyanezeknek a szolgáltatásoknak a készen elérhető megvalósítását kapjuk.

Mivel a függőségi tulajdonságok különbözőző WPF-központú típusok felhasználásával állnak elő, egyértelműen lehetőségünk van saját függőségi tulajdonságaink létrehozására – bár erre nem lesz szükség a könyvben található példákhoz. A következő kód összefoglalja a függőségi tulajdonságok deklarációjának központi elemeit (itt a tulajdonságot a statikus írásvédett `dependencyProperty` típus deklarálása során regisztráljuk):

```
public class MyOwnerClass : DependencyObject
{
    // A DependencyProperty használata a MyProperty háttértáráként.
    // Ez lehetővé teszi az animációt, a stílus alkalmazását,
    // a kötetstb...
    public static readonly DependencyProperty MyPropertyProperty =
        DependencyProperty.Register("MyProperty", typeof(int),
            typeof(OwnerClass), new UIPropertyMetadata(0));

    // XML-barát burkoló a statikus írásvédett mezőhöz.
    // Ez azért szükséges, mert nem tudunk metódusokat
    // (GetValue/SetValue) hívni XML-ben.
    public int MyProperty
    {
        // A GetValue/SetValue a DependencyObject
        // átlapoztályból származik.
        get { return (int)GetValue(MyPropertyProperty); }
        set { SetValue(MyPropertyProperty, value); }
    }
}
```

Ha további részleteket szeretnénk erről a WPF programozási szerkezetről megtudni, olvassuk el a .NET Framework 3.5 SDK dokumentáció „Custom Dependency Properties” témakörét.

Továbbított események

Nem a tulajdonságok az egyetlen .NET programozási szerkezetek, amelyeknek egy kis frissítésre volt szükségük, hogy megfelelően működjenek a WPF API-val. A szabványos CLR-eseménymodell is finomodott egy kicsit, hogy biztosítsa az események feldolgozhatóságát oly módon, hogy az megfelelő legyen egy objektumra XML-leírása számára. Tételizzuk fel, hogy létrehozunk a WPFControlEvents nevű új WPF-alkalmazás projektet. Most frissítsük a kezdő ablak kezdeti XML-leírását a következő <Button> típus hozzáadásával a kezdeti <Grid> típushoz:

```

<Button Name="btnClickMe" Height="75" Width = "250"
Click = "btnClickMe_Clicked">
<StackPanel Orientation="Horizontal">
<Label Height="50" FontSize="20">Fancy Button!</Label>
<Canvas Height="50" Width="100">
<Ellipse Name = "outerEllipse" Fill = "green" Height = "25"
Width = "50" Cursor="Hand" Canvas.Left="25"
Canvas.Top="12"/>
<Ellipse Name = "innerEllipse" Fill = "yellow" Height = "15"
Width = "36" Canvas.Top="17" Canvas.Left="32"/>
</Canvas>
</StackPanel>
</Button>

```

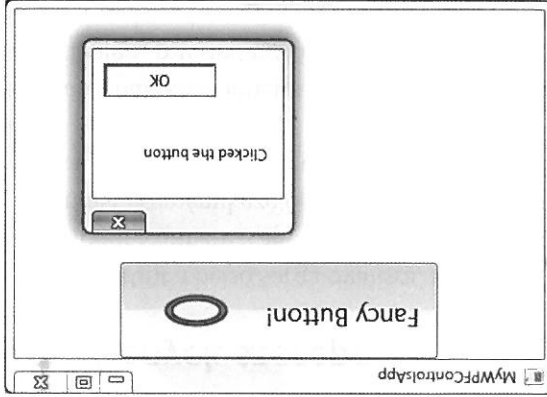
Figyeljük meg, hogy a <Button> esemény nyitó definíciójában úgy kezeltük a Click eseményt, hogy megadtuk a módus nevét, mely akkor fut le, ha az esemény bekövetkezik. A kattintási esemény a RoutedEventArgs objektust veszi fel először, amely olyan eseménykezelőt vár, amely objektumot vesz fel először paraméterként, és System.Windows.RoutedEventArgs típust másodikként:

```

public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
{
// Csinaljunk valamit, ha a gombra kattint.
MessageBox.Show("Clicked the button");
}

```

A 29.4. ábra mutatja az elvárt kimenetet, amikor rákattintunk a jelenlegi vezérlőelemre (megjelentési okok miatt módosítottam a kezdeti <Grid> típust <StackPanel> típusra, ami megmagyarázza, hogy a gomb miért az ablak tetejének közepén van, ahelyett, hogy az ablak közepén helyezkedne el).



29.4. ábra: Események kezelése egy összetett Button típus esetén

Most vizsgáljuk meg a Button típusunk jelenlegi összetételét. Több beágyazott elemet tartalmaz, a felhasználói felületének megjelenítéséhez (Canvas, Ellipse, Label stb.). Képzeljük el, hogy a WPF-eseménykezelés milyen unalmas lenne, ha arra kényszerülnénk, hogy kezeljünk egy kattintási eseményt mindezen részlem esetében. Végül is a végfelhasználó bárhova kattinthat a gomb hatásainak hatókörén belül (a Label objektumra, az ovális zöld területre, a gomb felszínre stb.). Nemcsak munkaigényes lenne, ha a Button típus minden aspektusához külön eseménykezelőt hoznánk létre, hanem olyan utálatos kódot kapnánk, amelyet nehéz lenne karbantartani a jövőben.

A Windows Forms eseménymodellben az ehhez hasonló egyedi vezérlőelem megkövetelné tőlünk, hogy a gomb minden eleme számára létrehozzuk a kattintási eseményt. Szerencsére a WPF *továbbított eseményei* automatikusan gondját viselik ennek. A továbbított események automatikusan továbbítanak egy eseményt felfelé (vagy lefelé) az objektumfán, megfelelő kezelőt keresve. Konkrétában, a továbbított események három „bejárási stratégiát” használnak. Ha egy esemény a kezdőponttól felfelé mozog az objektumfán, akkor az eseményt *buborékeseménynek* nevezzük. Ezzel ellentétben, ha egy esemény a kezdőponttól lefelé halad a kapcsolódó részlemek felé, akkor *lefutó esemény*. Végül, ha egy eseményt csak az eredeti elem váltja ki és kezeli (amit normális CLR-eseményként jellemezhetünk), akkor a neve *közvetlen esemény*.

Megjegyzés A függőségi tulajdonságokhoz hasonlóan a továbbított események a WPF-specifikus segédtypusok használatával megvalósított WPF-specifikus szerkezetek, így nincs olyan specifikus C#-szintaxis, amelyet a továbbított események kezeléséhez el kellene sajátítanunk.

A továbbított buborékesemények szerepe

Ha a jelenlegi példában a felhasználó rákattint a belső sárga oválisra, a kattintási esemény „kibuggyan” a következő hatókör szintjére (Canvas), majd a StackPanel, végül a Button típushoz, ahol a kattintási eseménykezelője található. Ehhez hasonlóan, ha a felhasználó rákattint a Label objektumra, az esemény „kibuggyan” a StackPanel, majd a Button típushoz.

A továbbított buborék eseménymintának köszönhetően nem kell foglalkoznunk azzal, hogy egy összetett vezérlőelem minden tagjához regisztráljunk bizonyos kattintási eseménykezelőket. Azonban, ha egyedi kattintási logikát szeretnénk végrehajtani több elemen ugyanazon az objektumfán belül, ezt megtehetjük. Példaként tételizzuk fel, hogy az outere Ellipse vezérlőelem kattintását

egyedi módon szerelnénk kezelni. Először kezeljük a `mousedown` eseményt ezen részem esetében (a grafikusán renderelt típusok, mint az `Ellipse` nem támogatják a „kattintás” eseményt, azonban figyelhetük az egér gombjának tevékenységet a `mousedown`, a `mouseup` stb. események révén):

```
<button Name="btnClickMe" Height="75" Width = "250"
Click = "btnClickMe_Click">
<stackpanel Orientation="Horizontal">
<label Height="50" FontSize="20">Fancy Button</label>
<Canvas Height="50" Width="100">
<ellipse Name = "outerEllipse" Fill="green"
Height="25" MouseDown="OuterEllipse_MouseDown"
Width="50" Cursor="Hand" Canvas.Left="25"
Canvas.Top="12"/>
<Ellipse Name = "innerEllipse" Fill="yellow" Height = "15"
Width="36" Canvas.Top="17" Canvas.Left="32"/>
</Canvas>
</StackPanel>
</Button>
```

Ezután implementálunk egy megfelelő eseménykezelőt, amely szemléltetés-képpen egyszerűen megváltoztatja a főablak `Title` tulajdonságát:

```
public void OuterEllipse_MouseDown(object sender, RoutedEventArgs e)
{
// Az ablak címének megváltoztatása.
this.Title = "You clicked the outer ellipse!";
}
```

Ezzel most már különböző műveletek történnék attól függően, hogy a végfelhasználó hova kattintott (ami lényegében a külső ellipszis és a gomb hatókörén belül minden más hely).

Megjegyzés A továbbított buborékesemények mindig a kezdőponttól a következő tartalmazó hatókör felé mozognak. Így, ha ebben a példában az `innerEllipse` objektumra kattintanánk, akkor az esemény a Canvas felé tartana, *nem* pedig az `outerEllipse` objektum felé, ugyanis mindkét `Ellipse` típus a Canvas hatókörében van.

A buborékesemények folytatása és leállítása

Jelenleg, ha a felhasználó az `outerEllipse` objektumra kattint, kiválthatja az `Ellipse` típushoz regisztrált `MouseDown` eseménykezelőt, amikor a buboréklogika megáll (ezért nem láthatjuk, hogy a rendszer végrehajthja a `Button` típus kattintási eseménykezelőjét). Legtöbbször ez a kívánt hatás; azonban, ha arra

```
<Ellipse Name = "outerEllipse" Fill = "Green" Height = "25"
MouseDown = "outerEllipse_MouseDown"
PreviewMouseDown = "outerEllipse_PreviewMouseDown"
Width = "50" Cursor = "Hand" Canvas.Left = "25"
Canvas.Top = "12"/>
```

esemény kezelésével az `outerEllipse` objektum esetében:

és buborékesemények kölcsönös kapcsolatát szemléltessük a `PreviewMouseDown` szintaxist a kódfejlécsben, majd implementáljuk a kezelőt a kódfejlécsben. A `left` névét XAML-ben (vagy, ha kell, használjuk a megfelelő `C#` eseménykezelő-esemény kezelésének folyamata, egyszerűen rendeljük hozzá az eseménykezelő-Egy `left` esemény kezelése pont ugyanúgy néz ki, mint bármilyen más

esemény következik be.

buborékeseményt a rendszer kiváltána, először a `PreviewMouseDown` `left` párja, amely a buborék megfelelő *előt* indul el. Például mielőtt a `MouseDown` tárlékban lévő összes buborékeseménynek van kapcsolódó `left` esemény-lehető belsőbb hatókörök fele haladnak. Nagyjából a WPF alapoztaly-könyv-előtaggal kezdődnek, pl. `PreviewMouseDown` a kezdőelemtől az objektumtárlékban vagy *left* jellegűek lehetnek. A `left` események (amelyek mind a `Preview` A továbbított események tulajdonképpen *buborék* (ahogy az előbb leírtuk)

A továbbított left események szerepe

akár külön elemként (pl. egy `Ellipse` a `Button` típuson belül) szerepeljen.

tartalomcsoport akár egyetlen logikai elemként (pl. egy `Button` elemként), viden, a továbbított buborékesemények lehetővé teszik, hogy egy összetett box megjelenítése követ a `Button` típus kátfutási eseménykezelőjén belül. Rö-Ebben az esetben láthatjuk, hogy az ablak címe megváltozik, amit a `message-`

```
}
e.Handled = false;
// A továbbítás ("bugbuborékozás") folytatódik!
this.Title = "You clicked the outer ellipse!";
// Az ablak címe megváltoztatása.
```

```
{
public void outerEllipse_MouseDown(object sender, RoutedEventArgs e)
```

nak értéket `false`-ra:

szerepénk utasítani a WPF-et, hogy folytassa az esemény mozgatótást fölfelé az objektumtárlék, beállíthatjuk a `RoutedEventArgs` típus `Handled` tulajdonságá-

Ezután tervezzük át az aktuális C#-osztálydefiniációt az eseménykezelők módosításával (minden típusnál), hogy információt fűzzenek egy sztringhez, és végül megjelenítsék az értéket. Ez lehetővé teszi, hogy megvizsgáljuk a háttérben kiváltott események folyamatait:

```
public partial class MainWindow : System.Windows.Window
{
    // Ezt használjuk arra, hogy tároljuk az eseménykezelőkhöz
    // tevékenység adatait.
    string mouseActivity = string.Empty;

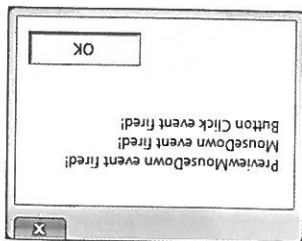
    public MainWindow()
    {
        InitializeComponent();
    }

    public void btnClickMe_Clicked(object sender, RoutedEventArgs e)
    {
        // A végző sztring megjelenítése.
        mouseActivity += "Button Click event fired!\n";
        MessageBox.Show(mouseActivity);
        // A sztring törlése a következő teszthez.
        mouseActivity = string.Empty;
    }

    public void outEventListener_MouseDown(object sender,
        RoutedEventArgs e)
    {
        // Adat hozzáadása a sztringhez.
        mouseActivity += "MouseDown event fired!\n";
        // A "bugyborékolás" folytatódik!
        e.Handled = false;
    }

    public void outEventListener_PrevMouseEvent(object sender,
        RoutedEventArgs e)
    {
        // Adat hozzáadása a sztringhez.
        mouseActivity = "PreviousMouseDown event fired!\n";
        // A "bugyborékolás" folytatódik!
        e.Handled = false;
    }
}
```

Ha futtatjuk a programot, és nem kattintunk sehova a külső ellipszis határain belül, akkor egyszerűen a „Button Click event fired!” üzenet jelenik meg az üzenetdobozban. Azonban, ha a külső ellipszisképen belülre kattintunk, akkor a 29.5. ábrán látható üzenetdoboz jelenik meg.



29.5. ábra: Először a lefűt, majd a buborékesemény

Elgondolkozhatunk, hogy vajon miért hajlamosak párban járni a WPF-események (egy lefűt és egy buborékesemény). A válasz az, hogy az események előnnezetével lehetőségünk van bármilyen speciális logika (adatellenőrzés, buborékesemény leállítás stb.) végrehajtására, mielőtt a buborékre szel elindulna. Az esetek nagy többségében *nem lesz szükségünk* a Preview prefixumos lefűt események kezelésére, egyedül csak a (nem Preview prefixumos) buborék

eseményekkel kell törödnünk.

A függőségi tulajdonság kézi megírásának feladatához hasonlóan, a lefűt események kezelése általában csak akkor szükséges, ha leszarmaztatunk egy már meglévő WPF-vezérlőelemet. Ehhez kapcsolódik, hogy ha egyedül WPF-vezérlőelemet készítenünk, akkor létrehozhatunk egyedül továbbított eseményeket (akár buborék-, akár lefűt eseményeket) egy olyan technika használatával, amely hasonlít az egyedül függőségi tulajdonság készítéséhez. További részletekért forduljunk a „How to: Create a Custom Routed Event” témakörhöz a .NET Framework 3.5 SDK dokumentációjában.

Forráskód A WPFControlEvents kódját a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A Button típusok használata

Most, hogy áttekintettük a függőségi tulajdonságok és a továbbított események részleteit, abban a helyzetben vagyunk, hogy jobban megérthetjük magukat a WPF-vezérlőelemeket, kezdve a gombtípusokkal. Összönösen tisztában lehetünk a gombtípusok szerepével. Ezek a felhasználói felület olyan elemei, amelyeket megnyomhatunk az egér vagy a billentyűzet révén (az Ender vagy a szökőz billentyűvel), amikor ezek aktívak. A WPF-ben a buttonbase szülőosztályként szolgál a három alapvető le származott típus (button, repeatbutton és togglebutton) számára.

A Buttonbase típus

Mint minden szülőosztály, a buttonbase egy polymorf interfészt biztosít a le származott típusok számára (az alaposztályából, a contentcontrol osztályból örökölt tagok mellett). Például a buttonbase definiálja a kattintási eseményt is. Emellett ez a szülőosztály határozza meg az ispressed tulajdonságot, amellyel végrehajthatunk egy műveletet, ha a le származott típust meg nyomták, de még nem engedték fel. Ezenkívül a 29.2. táblázat bemutatja a buttonbase absztrakt ösztály néhány további érdekes tagját.

A Buttonbase tagja		Jelentés
Clickmode	A Clickmode felsorolás értékeinek alapján ezzel a tulajdonsággal határozhatjuk meg, hogy mikor tűz eljen a kattintási esemény.	
Command	Ahogy a fejezet későbbi részében szó lesz róla, a felhasználói felület több elemének van társított „parancsa”, amely a command tulajdonság beállításával rendelhető hozzá.	
CommandParameter	Ezzel a tulajdonsággal paramétereket adhatunk a command tulajdonság által megadott elemnek.	
CommandTarget	Ez a tulajdonság lehetővé teszi, hogy meghatározzuk a command tulajdonság által megadott utasítás fogadóját.	

29.2. táblázat: A Buttonbase típus tagjai

Az utasításközpontú tagok mellett (amelyeket a fejezet végén vizsgálunk meg) a leggyakoribb tag talán a `ClickMode`, amellyel megadhatjuk egy gombra kattintás három különböző módját. Ehhez a tulajdonsághoz a kapcsolódó `System.Windows.Controls.ClickMode` felsorolásból bármilyen értéket hozzárendelhetünk:

```
public enum ClickMode
{
    Release,
    Press,
    Hover
}
```

Tételezzük fel például, hogy a következő XAML-kóddal írunk le egy gombot, melynek a `ClickMode` tulajdonságát `ClickMode.Hover` értékre állítottuk:

```
<Button Name = "btnHoverClick" ClickMode = "Hover"
        Click = "btnHoverClick_Click"/>
```

Ezáltal a kattintási esemény azonnal elindul, amint az egérkurzor bekerül a gomb területére. Noha lehet, hogy nem ez a leggyakoribb mód használatára egy tipikus nyomógomb esetében, a "lebegés" (`Hover`) mód hasznos lehet egyedi stílusok, sablonok vagy animációk készítésekor.

A Button típus

Az első lezármazott típus, a `Button`, két azonnali érdeklődésre számot tartó tulajdonságot biztosít számunkra: az `IsCancel` és az `IsDefault` tulajdonságok, amelyek nagyon hasznosak, ha `OK` és `Mégse` gombokat tartalmazó párbeszédablakokat hozunk létre. Amikor az `IsCancel` értéke `true`, a gomb kattintási eseménye első, ha a felhasználó megnyomja az `Esc` billentyűt. Ha az `IsDefault` értéke `true`, a gomb kattintási eseménye első, ha a felhasználó megnyomja az `Enter` billentyűt. Tekintsük meg a két `Button` típus következő XAML-leírását:

```
<!-- Tételezzük fel, hogy ezeket egy window típus <StackPanel>
      osztályán belül definiáljuk -->
<Button Name = "btnOK" IsDefault = "true" Click = "btnOK_Click"
        Content = "OK"/>
<Button Name = "btnCancel" IsCancel = "true"
        Click = "btnCancel_Click" Content = "Cancel"/>
```

Ha implementálunk a deklarált eseménykezelők mindegyikét a kapcsolódó kódajában, akkor futtathatunk az alkalmazást, és ellenőrizhetünk, hogy az Enter vagy az Esc billentyű megnyomásakor a megfelelő eseménykezelőt hívja-e a rendszer. Ez lenne a helyzet akkor is, ha az ablak egy másik felhasználóiféltől-eleme (pl. egy szövegbeviteli mező) lenne aktiv.

A ToggleButton típus

Alapértelmezés szerint a `ToggleButton` típus (amelyet a `System.Windows.Controls.Primitives` névtérben definiálunk) rendelkezik egy olyan felhasználói felülettel, amely hasonló a `Button` típushoz; azonban egyedi képessége, hogy megőrizi lenyomva tartott állapotát, ha a felhasználó rákattint. Ennek érdekében a `ToggleButton` biztosítja az `IsChecked` tulajdonságot, amely változat a `true` és a `false` értékek között, amikor a végfelhasználó rákattint a felhasználói felület elemére. Továbbá a `ToggleButton` két olyan eseményről gondoskodik (`Checked` és `Unchecked`), amelyekkel kezelhető az állapotváltozás elfogása. Így, egy olyan egyszerű kapcsoló XAML-leírása, amely két egyedi eseménykezelőn kezel az eseményeket:

```
<!-- Egy Igen/Nem kapcsológomb -->
<ToggleButton Name="toggleoffbutton"
  Checked="toggleoffbutton_Checked"
  Unchecked="toggleoffbutton_Unchecked">
  Offi
</ToggleButton>
```

Az eseménykezelők egyszerűen odaillő szöveges üzenettel módosítják a `Content` tulajdonságot:

```
protected void toggleoffbutton_Checked(object sender,
  RoutedEventArgs e)
{
    toggleoffbutton.Content = "On!";
}

protected void toggleoffbutton_Unchecked(object sender,
  RoutedEventArgs e)
{
    toggleoffbutton.Content = "Off!";
}
```

Ha szeretnénk tömöríteni a mögöttes kódjainkat, hogy egyetlen kezelőt használjon minden egyes eseményhez, módosíthatjuk a XAML-definíciót, hogy mind a checked, mind az unchecked esemény ugyanarra a kezelőre mutasson (vagyis a toggleonoffbuttonpressed eseménykezelőre), majd használjuk az isChecked tulajdonságot, hogy változtassuk az üzenetet:

```
protected void toggleonoffbuttonpressed(object sender,
    RoutedEventArgs e)
{
    if (toggleonoffbutton.IsChecked == false)
        toggleonoffbutton.Content = "Off!";
    else
        toggleonoffbutton.Content = "On!";
}
```

Végül ne feledjük, hogy a ToggleButton támogatja a háromállapotú működést (az IsThreeState tulajdonság és az Indeterminate esemény révén), és lehetővé teszi annak ellenőrzését, hogy a vezérlő bekapcsol, nem bekapcsolt, vagy egyik sem. Noha furcsának tűnhet, hogy egy gomb ilyen módon figyelje önmagát, azonban értelmet nyer a dolog azon típusok esetében, amelyek a ToggleButton típusból származnak, mint például a haramosan bemutatásra kerülő checkbox típus.

Megjegyzés Általános szabályként elmondható, a System.Windows.Controls.Primitives névtérben definiált típusok (beleértve a ToggleButton típust) további testreszabás nélkül nem számlálnak azonnal hasznosnak.

A RepeatButton típus

Az utolsó tárgyal, ButtonBase-leszármazott típus a RepeatButton típus, amelyet szintén a System.Windows.Controls.Primitives névtérben definiálunk. Ez a típus alapértelmezés szerint szintén hasonlít a standard Button típusra, azonban megvan az a képessége, hogy folyamatosan elindítja saját kattintási eseményét, amikor a végfelhasználó megnyomott állapotban tartja a vezérlőt. A kattintási esemény elindításának gyakorisága függ a Delay és az Interval tulajdonságokhoz hozzárendelt értékektől (amelyek mindegyike ezredmásodpercben rögzített).

A RepeatButton típus (a ToggleButton típushoz hasonlóan) önmagában valóban nem túl hasznos. Azonban a felírt viselkedés hasznos lehet teste szabott felhasználói felületek létrehozásakor. Vegyük figyelembe a tényt, hogy – a