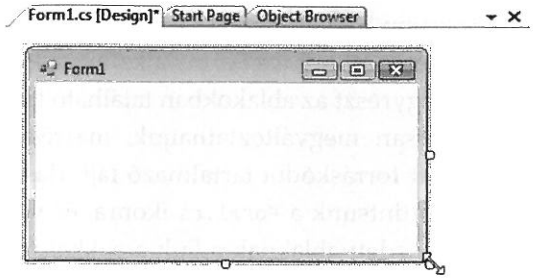
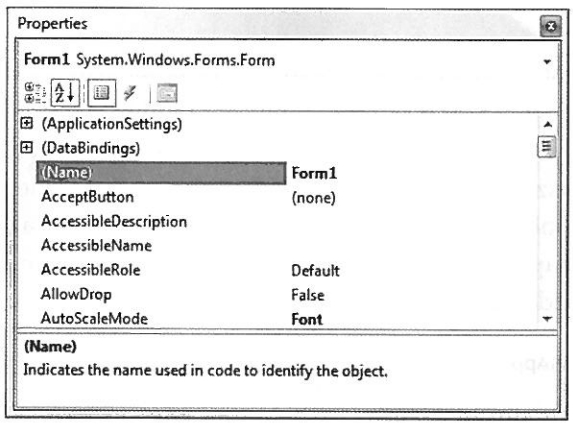


27.5. ábra: *A vizuális Forms-tervező*

Az ablak (és az űrlapon található bármelyik vezérlőelem) megjelenésének és működésének konfigurálását a Properties ablakban tehetjük meg. Ennek az ablaknak a segítségével a tulajdonságokhoz értékeket rendelhetünk, valamint egy adott vezérlőhöz eseménykezelőket hozhatunk létre (lásd részletesen később). Ha a tervezőfelület vezérlőelemek gyűjteményével rendelkezik, akkor a konfigurálni kívánt elemet a Properties ablak tetején található legördülő listamező segítségével választhatjuk ki.

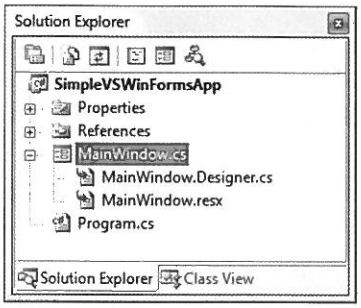
Az űrlap jelenleg nem rendelkezik tartalommal, ezért csak egy a kezdeti űrlaphoz tartozó listát látunk. Alapértelmezés szerint az űrlap a Form1 nevet kapta, ahogy ez az írásvédett (`Name`) tulajdonságban is olvasható (lásd a 27.6. ábrát).



27.6. ábra: *A Properties ablak segítségével beállíthatjuk a tulajdonságokat és az eseménykezelőket*

Megjegyzés A Properties ablakot a legördülő listamező alatti első két gomb segítségével aszerint is konfigurálhatjuk, hogy a tartalom kategóriánként vagy alfabetikus sorrendben jelenjen meg. Érdemes az elemeket ábécésorrendbe rendezni, hogy könnyen megtaláljunk egy adott tulajdonságot vagy eseményt.

A következő lényeges tervezőelem a Solution Explorer ablak. Bár minden Visual Studio projekt támogatja ezt az ablakot, a Windows Forms-alkalmazások létrehozásakor különösen hasznos, hogy egyrészt az ablakokban található fájlok és a kapcsolódó osztály nevét gyorsan megváltoztathatjuk, másrészt megnézhetjük a tervező által karbantartott forráskódot tartalmazó fájlt (lásd részletesen később). Jobb egérgombbal kattintsunk a `Form1.cs` ikonra, és válasszuk a Rename lehetőséget. Ennek a kezdeti ablaknak adjuk a sokkal inkább megfelelő `MainWindow.cs` nevet. A végeredmény a 27.7. ábrán látható.



27.7. ábra: *A Solution Explorer ablak lehetővé teszi a Form-leszármazott típusok és a kapcsolódó fájlok átnevezését*

A kezdeti űrlap

A menürendszer kialakítása előtt vizsgáljuk meg, hogy az alapértelmezés szerint mit is hozott létre a Visual Studio 2008. Először jobb egérgombbal kattintsunk a `MainWindow.cs` ikonra a Solution Explorer ablakban, majd válasszuk a View Code elemet. Az űrlap részleges típusként lett meghatározva, és ez lehetővé teszi, hogy az egyes típusokat több kódfájlban határozzuk meg (lásd az előző kötet 5. fejezetét). Az űrlap konstruktora meghívja az `InitializeComponent()` metódust, a típusunk pedig „az egy" `Form`.

```
namespace SimpleVSWinFormsApp
{
  public partial class MainWindow : Form
  {
    public MainWindow()
    {
      InitializeComponent();
    }
  }
}
```

Az `InitializeComponent()` definiálása egy önálló fájlban valósul meg, amely kiegészíti a részleges osztálydefiníciót. Az elnevezés hagyományai szerint ez a fájl mindig `.Designer.cs` végződésű, amelyet a kapcsolódó, `Formból` származtatott típust tartalmazó C#-fájl neve előz meg. A Solution Explorer ablak segítségével nyissuk meg a `MainWindow.Designer.cs` fájlt. Nézzük meg az alábbi (az egyszerűség kedvéért a megjegyzések nélküli) forráskódot:

```
partial class MainWindow
{
  private System.ComponentModel.IContainer components = null;

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

  private void InitializeComponent()
  {
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
  }
}
```

Az `IContainer` tagváltozó és a `Dispose()` metódus egy kicsit többet jelent a Visual Studio-tervezőeszközök által használt infrastruktúránál. Ám az `InitializeComponent()` is jelen van, és figyelembe is kell venni. Nemcsak az űrlap konstruktora hívja meg ezt a metódust futási időben, hanem tervezési időben a Visual Studio is ugyanezt a metódust használja annak érdekében, hogy a Forms-tervezőben megfelelően jelenítse meg a felhasználói felületet. Ennek illusztrálására módosítsuk az ablakban található `Text` tulajdonsághoz rendelt értéket a következőre: „My Main Window". A tervező aktiválásakor az űrlap címe ennek megfelelően frissül.

Ha pedig a vizuális tervezőeszközöket (pl. a Properties ablak) használjuk, az IDE automatikusan frissíti az `InitializeComponent()` metódust. Ennek illusztrálására a Forms-tervező legyen az IDE aktív ablaka, és keressük meg a Properties ablakban található `Opacity` tulajdonságot. Az értéket írjuk át 0,8-ra (80%), ennek hatására az ablak enyhén átlátszóvá válik, amikor legközelebb lefordítjuk és futtatjuk a programot. Ha elvégeztük ezt a módosítást, vizsgáljuk meg ismét az `InitializeComponent()` implementációját:

```
private void InitializeComponent()
{
  this.SuspendLayout();
  //
  // MainWindow
  //
  this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
  this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
  this.ClientSize = new System.Drawing.Size(284, 264);
  this.Name = "MainWindow";
  this.Opacity = 0.8;
  this.Text = "My Main Window";
  this.ResumeLayout(false);
}
```

Amikor a Visual Studióval hozunk létre Windows Forms-alkalmazásokat, gyakorlatilag figyelmen kívül hagyhatjuk (és jellemzően ezt is kell tennünk) a `*.Designer.cs` fájlokat, mert az IDE karbantartja őket helyettünk. Ha szintaktikailag (vagy logikailag) helytelen forráskódot állítunk össze az `InitializeComponent()` metóduson belül, az a tervező összeomlásához vezethet. A Visual Studio tervezési időben gyakran átformázza ezt a metódust. Ezért ha saját forráskódot szeretnénk az `InitializeComponent()` metódushoz adni, megeshet, hogy azt az IDE törölni fogja. Ne feledjük, hogy a Windows Forms-alkalmazás minden ablaka részleges osztályokból áll össze.

A Program osztály

Azon kívül, hogy a kezdeti, `Form`-leszármazott típus implementálja , a Windows Application projekttípusai egy olyan statikus (`Program` nevű) osztályt is biztosítanak, amely meghatározza a program belépési pontját: a `Main()` metódust:

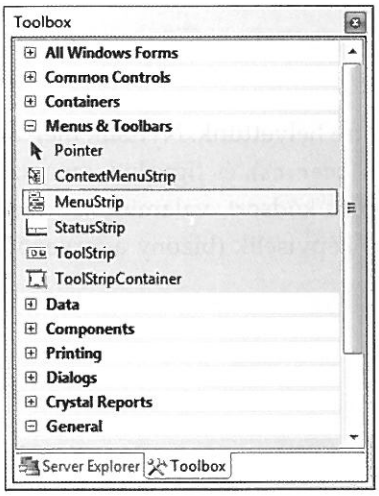
```
static class Program
{
  [STAThread]
  static void Main()
  {
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new MainWindow());
  }
}
```

A `Main()` metódus meghívja az `Application.Run()`-t, és elvégez néhány renderelési beállítást. Végül, de nem utolsósorban, a `Main()` metódus `[STAThread]` attribútummal rendelkezik. Ez arról tájékoztatja a futtatókörnyezetet, hogy

ha ez a szál klasszikus COM-objektumot hoz létre (beleértve a korábbi ActiveX felhasználói felületi vezérlőelemeket) saját élettartama során, akkor azokat egy COM által fenntartott területre, az *egyszálú tárrészbe* (Single-Threaded Apartment) helyezi. Röviden: ez biztosítja a COM-objektumok szálbiztos voltát még akkor is, ha ennek biztosítására egy adott COM-objektum szerzője nem gondoskodott a forráskódban.

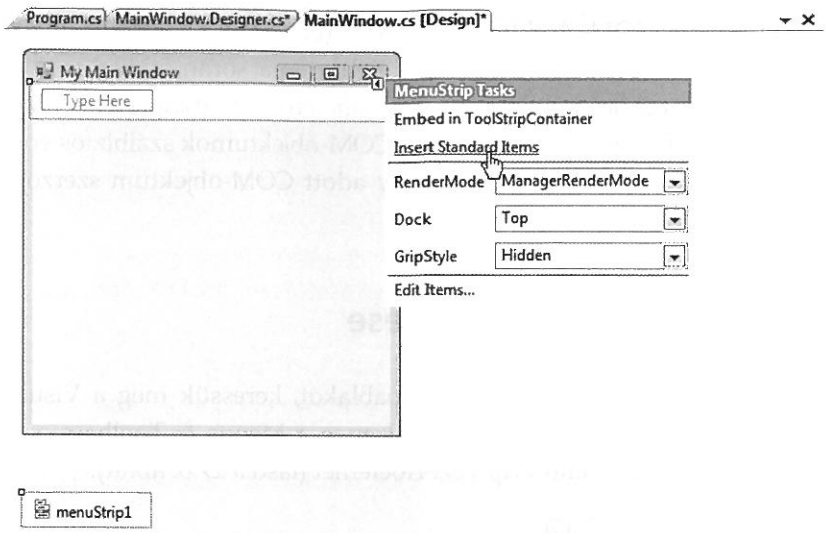
Menürendszerek vizuális építése

A továbbiakban aktiváljuk a Forms-tervezőablakot, keressük meg a Visual Studio 2008 Toolbox ablakát, majd keressük meg a Menus & Toolbars csomóponton belül található MenuStrip vezérlőelemet (lásd a 27.8. ábrát).



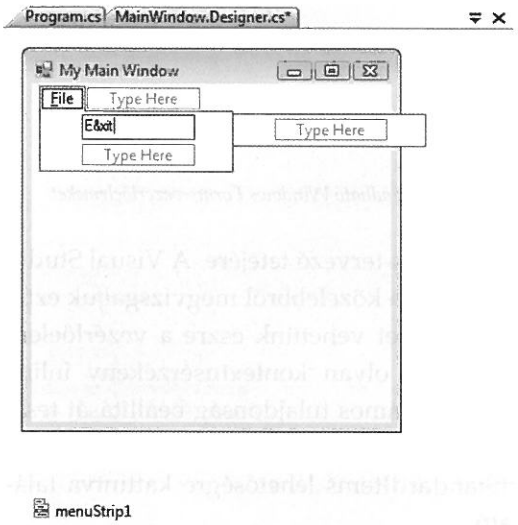
27.8. ábra: *A Toolbox ablak megjeleníti a tervezőfelülethez adható Windows Forms-vezérlőelemeket*

Húzzuk a MenuStrip vezérlőelemet a Forms-tervező tetejére. A Visual Studio a menüszerkesztő aktiválásával válaszol. Ha közelebbről megvizsgáljuk ezt a szerkesztőt, egy (nagyon) kicsi háromszöget vehetünk észre a vezérlőelem jobb felső sarkában. Az ikonra kattintva olyan kontextusérzékeny inline szerkesztőt nyitunk meg, amely azonnal számos tulajdonság beállítását teszi lehetővé (több Windows Forms-vezérlőelem hasonló inline szerkesztővel rendelkezik). Erre példát az InsertStandardItems lehetőségre kattintva találunk, ahogy ez a 27.9. ábrán is látható.



27.9. ábra: *Az inline menüszerkesztő*

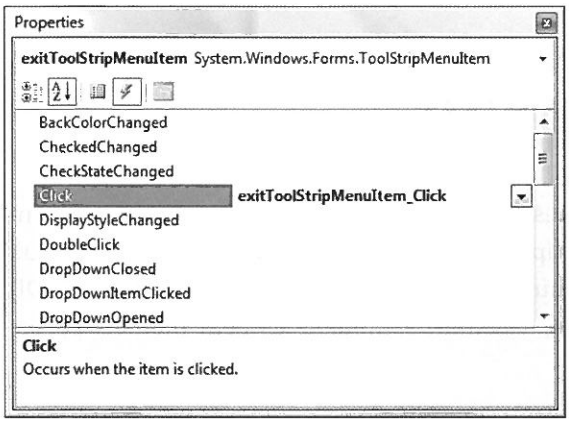
A Visual Studio egész menürendszert hozott létre helyettünk. Nyissuk meg a tervező által karbantartott fájlt (`MainWindow.Designer.cs`), és figyeljük meg az `InitializeComponent()` metódushoz adott számos kódsort, valamint azt a jó néhány tagváltozót, amelyek a menürendszert képviselik (bizony a tervezőeszközök *jó dolgok*).



27.10. ábra: *Menürendszer manuális készítése*

Végül ugorjunk vissza a tervezőhöz, és a Ctrl + Z billentyűkombinációval vonjuk vissza az előző műveletet. Így visszajutunk a kezdeti menüszerkesztőhöz, és eltávolítjuk a generált forráskódot. A menütervezővel egyszerűen csak beírjuk a fő **File** menüelemet, amelyet egy **Exit** almenü követ (lásd a 27.10. ábrát).

Az `InitializeComponent()` metódusban ugyanolyan forráskódot találunk, mint amilyet manuálisan állítottunk össze a fejezet első példájában. A mostani feladat befejezéséhez ugorjunk vissza a Forms-tervezőhöz, és kattintsunk a Properties ablakban található villámot ábrázoló gombra. Ez megjeleníti az összes olyan eseményt, amelyeket kezelhetünk a kijelölt vezérlőelemen. Jelöljük ki az `Exit` (alapértelmezésben `exitToolStripMenuItem` nevű) menüt, majd keressük meg a `Click` eseményt (lásd a 27.11. ábrát).



27.11. ábra: *Események létrehozása az integrált fejlesztői környezet segítségével*

Ekkor beírhatjuk annak a metódusnak a nevét, amelyet meg kell hívni, ha rákattintanak az elemre; vagy egyszerűbben: kattintsunk kétszer a Properties ablakban található eseménylista kívánt elemére. Ilyenkor az integrált fejlesztői környezet kiválasztja helyettünk (a *VezérlőelemNeve_EseményNeve()* mintát követő) eseménykezelő nevét. Az integrált fejlesztői környezet mindkét esetben kódcsonkot hoz létre, amelybe beleilleszthetjük az konkrét implementációt. Például:

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

```
    private void exitToolStripMenuItem_Click(object sender,
                                             EventArgs e)
    {
      Application.Exit();
    }
}
```

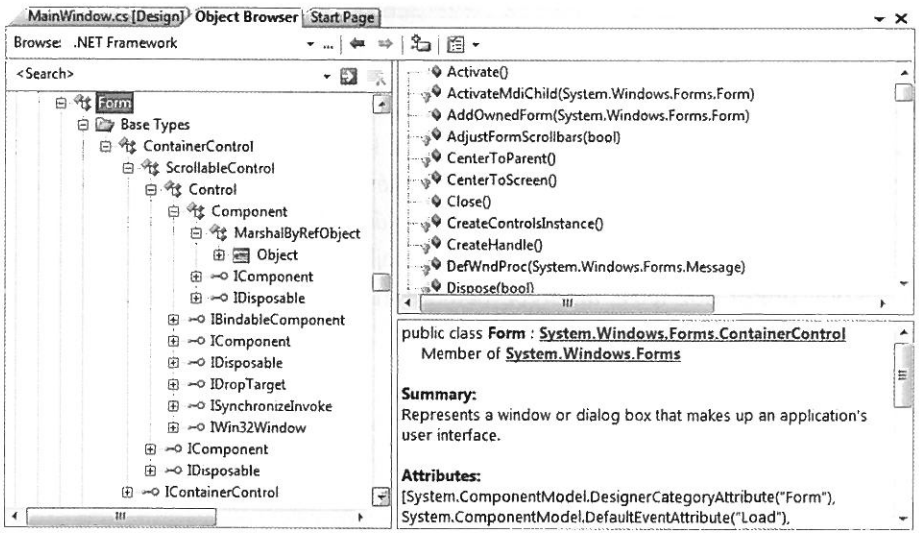
Ha megnézzük az `InitializeComponent()` metódust, látható, hogy a szükséges eseményfeliratkozás is megtörtént:

```
this.exitToolStripMenuItem.Click +=
        new System.EventHandler(this.exitToolStripMenuItem_Click);
```

Annak ellenére, hogy nyilvánvalóan sokkal több parancsikon, szerkesztő és integrált varázsló van az integrált fejlesztői környezetben, a fentiek már elegendőek a továbblépéshez.

Az űrlapok anatómiája

A következőkben még részletesebben megvizsgáljuk a `Form` típust. A Windows Forms világában a `Form` típus képviseli az alkalmazásablakokat, köztük a főablakokat, a többdokumentumos felhasználói felülettel rendelkező (MDI) alkalmazások gyermekablakait, valamint a modális és nem modális párbeszédablakokat.



27.12. ábra: *A System.Windows.Forms.Form származtatási lánca*

Ahogy azt a 27.12. ábra is mutatja, a `Form` típus nagy mennyiségű funkcionalitást örököl az ősosztályoktól, valamint az általa implementált, nagyszámú interfésztől.

A 27.2. táblázat a `Form` származtatási láncában található szülőosztályokba nyújt betekintést.

Szülőosztály	Jelentés
`System.Object`	Ahogy a .NET többi osztálya is, a *Form* „az egy" *object*.
`System.MarshalByRefObject`	Az ebből az osztályból származtatott típusokhoz távolról lehet hozzáférni a távoli típusra történő hivatkozással (és nem egy helyi másolattal).
`System.ComponentModel.Component`	Ez az osztály biztosítja az `IComponent` interfész alapértelmezett implementációját. A .NET univerzumban a komponensek olyan típusok, amelyek támogatják a tervezésidejű szerkesztést, de futási időben nem feltétlenül láthatók.
`System.Windows.Forms.Control`	Ez a típus biztosítja valamennyi Windows Forms UI-vezérlőelem általános felhasználói felületének tagjait, beleértve magát a `Form` típust is.
`System.Windows.Forms.ScrollableControl`	Ez az osztály határozza meg a vízszintes és a függőleges görgetősávra vonatkozó támogatást, valamint azokat a tagokat, amelyek lehetővé teszik a görgethető területen lévő minta kezelését.
`System.Windows.Forms.ContainerControl`	Ez az osztály biztosítja azoknak a vezérlőelemeknek a fókuszkezelését, amelyek más vezérlőelemek tárolójaként képesek működni.
`System.Windows.Forms.Form`	Ez az osztály képviseli az egyedi űrlapokat, a többdokumentumos felhasználói felülettel rendelkező (MDI) alkalmazások gyermekablakait és a párbeszédablakokat.

27.2. táblázat: *Alaposztályok a Form származtatási láncban*

Bár a `Form` típus teljes származtatása számos ősosztályt és interfészt foglal magába, *mégsem* kell megismernünk minden egyes szülőosztály vagy implementált interfész minden egyes tagjának a szerepét ahhoz, hogy kiváló Windows Forms-fejlesztővé válhassunk. A napi rendszerességgel használt tagok nagy részét (különösen a tulajdonságokat és az eseményeket) valójában könnyedén beállíthatjuk a Visual Studio 2008 Properties ablakában. Ami különösen fontos, hogy megismerjük a `Control` és a `Form` szülőosztályok által nyújtott funkcionalitást.

A Control osztály funkcionalitása

A `System.Windows.Forms.Control` osztály határozza meg a GUI-típusokhoz szükséges általános viselkedéseket. A `Control` alapvető tagjai lehetővé teszik a vezérlőelemek méretének és pozíciójának konfigurálását, a billentyűzet és az egér inputjának rögzítését, a tagok fókuszálásának/láthatóságának megszerzését vagy beállítását és így tovább. A 27.3. táblázat néhány érdekesebb, a kapcsolódó működés alapján csoportosított tulajdonságot határoz meg.

Tulajdonság	Jelentés

`BackColor`
`ForeColor`
`BackgroundImage`
`Font`
`Cursor`
	Ezek a tulajdonságok (színek, a szöveg betűtípusa, a megjelenítendő egérkurzor, amikor az egér a céleszköz felett áll stb.) határozzák meg a vezérlőelemek alapvető felhasználói felületét.

`Anchor`
`Dock`
`AutoSize`
	Ezek a tulajdonságok határozzák meg azt, hogy a vezérlőelemeket hogyan kell a tárolón belül pozicionálni.

`Top`
`Left`
`Bottom`
`Right`
`Bounds`
`ClientRectangle`
`Height`
`Width`
	Ezek a tulajdonságok határozzák meg a vezérlőelem aktuális dimenzióit.

Tulajdonság	Jelentés

`Enabled`
`Focused`
`Visible`
	Ezeknek a tulajdonságoknak mindegyike olyan logikai értéket ad vissza, amely az aktuális vezérlőelem állapotát határozza meg.

`ModifierKeys`	Ez a statikus tulajdonság ellenőrzi a váltóbillentyűk (Shift, Ctrl és Alt) aktuális állapotát, és ezt egy `Keys` típusban adja vissza.
`MouseButtons`	Ez a statikus tulajdonság ellenőrzi az egérgombok (a bal, jobb és középső egérgomb) aktuális állapotát, és ezeket egy `MouseButtons` típusban adja vissza.
`TabIndex` `TabStop`	Ezek a tulajdonságok a vezérlőelemek tabulátorsorrendjének konfigurálására szolgálnak.
`Opacity`	Ez a tulajdonság határozza meg a vezérlőelemek átlátszóságát (0,0: teljesen átlátszó; 1,0: teljesen átlátszatlan).
`Text`	Ez a tulajdonság jelzi a vezérlőelemhez rendelt sztringadatokat.
`Controls`	Ez a tulajdonság teszi lehetővé a hozzáférést az erősen típusos gyűjteményekhez (`ControlsCollection`), amelyek tetszőleges számú gyermek-vezérlőelemet tartalmaznak az aktuális vezérlőelemen belül.

27.3. táblázat: *A Control típus alapvető tulajdonságai*

A `Control` osztály számos olyan eseményt is meghatároz, amelyek lehetővé teszik (többek között) az egérrel, a billentyűzettel, a képpel és az áthúzással kapcsolatos tevékenységek elkapását. A 27.4. táblázat néhány érdekesebb, a kapcsolódó működés alapján csoportosított eseményt tartalmaz.

Esemény	Jelentés

`Click`
`DoubleClick`
`MouseEnter`
`MouseLeave`
`MouseDown`
`MouseUp`
`MouseMove`
`MouseHover`
`MouseWheel`
	Olyan események, amelyek lehetővé teszik az interakciót az egérrel.

Esemény	Jelentés
`KeyPress` `KeyUp KeyDown`	Olyan események, amelyek lehetővé teszik az interakciót a billentyűzettel.

`DragEnter`
`DragLeave`
`DragOver`
	Olyan események, amelyek az áthúzás tevékenységét ellenőrzik.

`Paint`	Olyan esemény, amely lehetővé teszi a GDI+ grafikus renderelő szolgáltatásaival folytatott interakciót.

27.4. táblázat: *A Control típus eseményei*

Végül a `Control` ősosztály számos olyan metódust is meghatároz, amelyek lehetővé teszik a `Control`-leszármazott típusok használatát. A `Control` típus metódusai közül jó néhány rendelkezik `On` prefixummal, amelyet aztán egy adott esemény neve követ (`OnMouseMove`, `OnKeyUp`, `OnPaint` stb.). Ezek az `On` prefixummal rendelkező virtuális metódusok a hozzájuk tartozó események alapértelmezett eseménykezelői. Ha felüldefiniáljuk ezeket a virtuális tagokat, lehetőségünk nyílik az események szükség szerinti elő- vagy utófeldolgozására, mielőtt (vagy miután) meghívjuk a szülő alapértelmezett implementációját:

```
public partial class MainWindow : Form
{
  protected override void OnMouseDown(MouseEventArgs e)
  {
    // Adjunk egyedi forráskódot a MouseDown eseményhez.

    // Ha elkészültünk, hívjuk meg a szülő implementációját.
    base.OnMouseDown(e);
  }
}
```

Ez bizonyos körülmények között hasznos lehet (különösen, ha standard vezérlőelemből származó, egyedi vezérlőelemet készítünk), az eseményeket ugyanis gyakran kell a standard C#-eseményszintaxissal kezelnünk (a Visual Studio-tervezőknek valójában ez az alapértelmezett viselkedése). Ha ily módon kezeljük az eseményeket, a szülő implementációjának befejezését követően a keretrendszer meg fogja hívni az egyedi eseménykezelőt. Nézzünk egy példát arra, hogy hogyan kezelhetjük manuálisan a `MouseDown` eseményt:

```
public partial class MainWindow : Form
{
  public MainWindow()
  {
    MouseDown += new MouseEventHandler(MainWindow_MouseDown);
  }

  private void MainWindow_MouseDown(object sender, MouseEventArgs e)
  {
    // Adjunk forráskódot a MouseDown eseményhez.
  }
}
```

Felsorolunk néhány másik, ezeken az `OnXXX()` metódusokon kívül található metódust is:

- `Hide()`: Elrejti a vezérlőelemet, és a `Visible` tulajdonságot hamis értékre állítja.
- `Show()`: Megjeleníti a vezérlőelemet, és a `Visible` tulajdonságot igaz értékre állítja.
- `Invalidate()`: Arra kényszeríti a vezérlőelemet, hogy egy `Paint` esemény küldésével újrarajzolja önmagát (a grafikus renderelés leírása ebben a fejezetben található, a „GDI+-alapú grafikus adatok renderelése" című részben).

A Form osztály funkcionalitása

A `Form` osztály jellemzően (de nem szükségszerűen) az egyedi `Form` típusok közvetlen ősosztálya. A nagyszámú – a `Control`, a `ScrollableControl` és a `ContainerControl` osztályból – örökölt tagon kívül a `Form` típusok további funkcionalitást is hozzáadnak elsősorban a főablakokhoz, az MDI-gyermekablakokhoz és a párbeszédablakokhoz. Először nézzük meg az alapvető tulajdonságokat. Ezeket a 27.5. táblázat tartalmazza.

Tulajdonság	Jelentés
`AcceptButton`	Lekérdezi vagy beállítja az űrlapon azt a gombot, amelyre a felhasználó az Enter billentyűvel kattinthat.
`ActiveMDIChild` `IsMDIChildIsMDIContainer`	Az MDI-alkalmazások kontextusán belül használjuk.

Tulajdonság	Jelentés
`CancelButton`	Lekérdezi vagy beállítja azt a gombot, amelyre a felhasználó az Esc billentyűvel kattinthat.
`ControlBox`	Lekérdezi vagy beállítja, hogy az űrlap rendelkezik-e vezérlőelem-dobozzal.
`FormBorderStyle`	Lekérdezi vagy beállítja az űrlap szegélystílusát. A `FormBorderStyle` felsorolt típussal együtt használatos.
`Menu`	Lekérdezi vagy beállítja az űrlapon rögzítendő menüt.
`MaximizeBox` `MinimizeBox`	Annak meghatározására szolgál, hogy az adott űrlap engedélyezi-e a dobozok maximalizálását és minimalizálását.
`ShowInTaskbar`	Annak meghatározására szolgál, hogy az űrlap látható-e a Windows-tálcán.
`StartPosition`	Az űrlap futásidejű kezdő pozícióját adja meg vagy veszi fel értékül a `FormStartPosition` felsorolás specifikációja alapján.
`WindowState`	Azt konfigurálja, hogy az űrlap hogyan jelenjen meg az indításkor. A `FormWindowState` felsorolt típussal együtt használatos.

27.5. táblázat: *A Form típus tulajdonságai*

A számos, `On` prefixummal ellátott, alapértelmezett eseménykezelő mellett a 27.6. táblázat néhány, a `Form` típus által meghatározott, alapvető metódus listáját tartalmazza.

Metódus	Jelentés
`Activate()`	Aktivál és az előtérbe helyez egy adott űrlapot.
`Close()`	Bezárja az űrlapot.
`CenterToScreen()`	Az űrlapot a képernyő középpontjába helyezi.
`LayoutMDI()`	A gyermekűrlapoknak a szülőűrlapokon belüli elrendezésére szolgál (a `LayoutMDI` felsorolt típus alapján).
`ShowDialog()`	Az űrlapot modális párbeszédablakként jeleníti meg.

27.6. táblázat: *A Form típus kulcsfontosságú metódusai*

Végül pedig a `Form` osztály számos olyan eseményt határoz meg, amelyek közül sok az űrlap élettartama alatt sül el. A 27.7. táblázat a legfontosabbakat tartalmazza.

Esemény	Jelentés
`Activated`	Mindig az űrlap *aktiválásakor* következik be, azaz akkor, amikor az űrlap az asztal aktuális előterében van.
`Closed`, `Closing`	Annak meghatározására szolgál, hogy az űrlap be fog záródni, vagy már bezáródott.
`Deactivate`	Mindig az űrlap *deaktiválásakor* következik be, azaz akkor, amikor az űrlap elkerül az asztal aktuális előteréből.
`Load`	Akkor következik be, amikor az űrlap már lefoglalódott a memóriába, de még nem látható a képernyőn.
`MDIChildActive`	A gyermekablak aktiválásakor következik be.

27.7. táblázat: *A Form típus néhány eseménye*

A Form típus életciklusa

Ha már programoztunk felhasználói felületeket GUI-elemkönyvtárakkal (pl. Java Swing, Mac OS X Cocoa vagy nyers Win32 API), akkor tudjuk, hogy az „ablaktípusok" számos olyan eseménnyel rendelkeznek, amelyek az élettartamuk alatt elsülnek. Ugyanez érvényes a Windows Form-alkalmazásra is. Az űrlapok élettartama konstruktoruk meghívásakor kezdődik, még azelőtt, hogy átadnánk őket az `Application.Run()` metódusnak.

Miután az objektumot a felügyelt heapre helyeztük, a keretrendszer elsüti a `Load` eseményt. A `Load` eseménykezelőjén belül lehetőségünk van az űrlap megjelenésének és működésének konfigurálására, az űrlapon megjelenő gyermekvezérlőelemek (pl. `ListBox`, `TreeView` stb.) készítésére vagy éppen az űrlap működése közben használt erőforrások lefoglalására (adatbázis-kapcsolatok, proxyk távoli objektumokhoz stb.).

A `Load` esemény után a következő kiváltódó esemény az `Activated`. Az `Activated` akkor sül el, amikor az űrlap, mint aktív ablak, az asztal előterébe kerül. Az `Activated` esemény logikai párja (természetesen) a `Deactivate`, amely akkor következik be, amikor az űrlap, mint aktív ablak, elkerül az előtérből. Az `Activated` és a `Deactivate` esemény egy adott `Form` típus élettartama során számtalan esetben kiváltódhat, ha aktív alkalmazások között navigálunk.

Ha felhasználóként a kérdéses űrlap bezárása mellett döntünk, két eseményt váltunk ki, ezek a `Closing` és a `Closed`. Először a `Closing` esemény következik be, amely kiváló alkalmat kínál arra, hogy a program feltegye a felhasználónak az olyannyira utált (de hasznos) *„Biztos*, hogy be akarja zárni az alkalmazást?" kérdést. A megerősítő lépés igencsak fontos annak biztosítására, hogy a felhasználónak maradjon lehetősége elmenteni az alkalmazásközpontú adatokat a program befejezése előtt.

A `Closing` esemény a `System.ComponentModel` névtérben definiált `CancelEventHandler` metódusreferenciát használja. Ha a `CancelEventArgs.Cancel` tulajdonság igaz, megakadályozhatjuk az ablak megsemmisítését, és arra kényszeríthetjük, hogy normál üzemmódba térjen vissza. Ha a `CancelEventArgs.Cancel` tulajdonság hamis, akkor bekövetkezik a `Closed` esemény, a Windows Forms-alkalmazás kilép, ez kidobja az alkalmazástartományt, és befejezi a folyamatot.

Az űrlap élettartama során bekövetkező események sorrendjének gyors vizsgálatához tételezzük fel, hogy vagy egy új, `FormLifeTime` nevű Windows Forms-projektünk, és a kezdeti űrlapot (a Solution Explorerrel) átneveztük a következőre: `MainWindow.cs`. Az űrlap konstruktorán belül regisztráljunk a `Load`, az `Activated`, a `Deactivate`, a `Closing` és a `Closed` eseményekre. (A `+=` karakterek beírása és a Tab billentyű kétszeri megnyomása után az integrált fejlesztői környezet automatikusan létrehozza a megfelelő metódusreferenciát és eseménykezelőt, lásd az előző kötet 11. fejezetében):

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

  // Az élettartam során bekövetkező események kezelése.
  Closing += new CancelEventHandler(MainWindow_Closing);
  Load += new EventHandler(MainWindow_Load);
  Closed += new EventHandler(MainWindow_Closed);
  Activated += new EventHandler(MainWindow_Activated);
  Deactivate += new EventHandler(MainWindow_Deactivate);
}
```

Megjegyzés Azért kezeljük manuálisan ezeket az eseményeket, mert a Properties ablak (valami különleges okból kifolyólag) nem listázza a `Closing` és a `Closed` eseményt. A `Load`, `Activated` és `Deactivate` eseményeket azonban tervezésidejű eszköz segítségével is kezelhetjük.

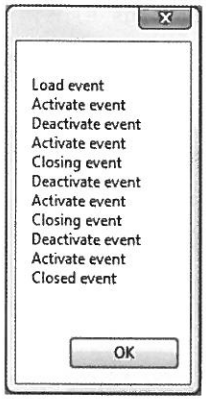
A `Load`, a `Closed`, az `Activated` és a `Deactivate` eseménykezelőkön belül olyan egyszerű üzenet segítségével módosítjuk az új, `Form`-szintű (`lifeTimeInfo` nevű) sztringtagváltozó értékét, amely az éppen elkapott esemény nevét jeleníti meg. A `Closed` eseménykezelőn belül egy üzenetdobozban jelenítjük meg ennek a sztringnek az értékét:

```
private void MainWindow_Load(object sender, System.EventArgs e)
{
  lifeTimeInfo += "Load event\n";
}

private void MainWindow_Activated(object sender, System.EventArgs e)
{
  lifeTimeInfo += "Activate event\n";
}

private void MainWindow_Deactivate(object sender,
                                   System.EventArgs e)
{
  lifeTimeInfo += "Deactivate event\n";
}

private void MainWindow_Closed(object sender, System.EventArgs e)
{
  lifeTimeInfo += "Closed event\n";
  MessageBox.Show(lifeTimeInfo);
}
```

A `Closing` eseménykezelőben a bemeneti `CancelEventArgs` használatával megkérdezzük a felhasználót, hogy valóban be kívánja-e zárni az alkalmazást. Az alábbi forráskódban látható, hogy a `MessageBox.Show()` metódus olyan `DialogResult` típust ad vissza, amely tartalmazza annak az értékét, hogy a felhasználó melyik gombot választotta. Olyan üzenetdobozt hoztunk létre, amely egy Yes és egy No gombot tartalmaz; tehát az érdekel bennünket, hogy a `Show()` metódus a `DialogResult.No` értéket adja-e vissza, vagy sem.

```
private void MainWindow_Closing(object sender, CancelEventArgs e)
{
  lifeTimeInfo += "Closing event\n";
  // Egy Yes és No gombot tartalmazó üzenetdoboz megjelenítése.
  DialogResult dr = MessageBox.Show("Do you REALLY want to close
        this app?", "Closing event!", MessageBoxButtons.YesNo);

  // Melyik gombra kattintottak?
  if (dr == DialogResult.No)
    e.Cancel = true;
  else
    e.Cancel = false;
}
```

Futtassuk az alkalmazást, és (az `Activated` és a `Deactivate` esemény kiváltásához) váltogassunk az alkalmazások között, hogy az űrlap előtérbe, majd pedig háttérbe kerüljön. Ha végül bezárjuk az alkalmazást, a 27.13. ábrán láthatóhoz hasonló üzenetdoboz jelenik meg.



27.13. ábra: *A Form-leszármazott típusok élete és tartama*

Forráskód A FormLifeTime projektet a forráskódkönyvtár 27. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Reagálás az egér eseményeire

A `Control` ősosztály olyan eseményeket határoz meg, amelyek lehetővé teszik, hogy többféle módon reagáljunk az egértevékenységekre (pl. egér gombjának lenyomása vagy felengedése, kattintás, egér mozgatása stb.). Ennek kipróbálására hozzunk létre egy új Windows Application-projektet MouseEventsApp névvel (a Solution Explorerrel), nevezzük át a kezdeti űrlapot `MainWindow.cs`-re, és kezeljük a `MouseMove` eseményt a Properties ablakban. Ez létrehozza az alábbi eseménykezelőt:

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

  // Létrehozva a Properties ablak segítségével.
  private void MainWindow_MouseMove(object sender, MouseEventArgs e)
  {
  }
}
```

A `MouseMove` esemény a `System.Windows.Forms.MouseEventHandler` metódusreferenciára épül. Ez a metódusreferencia csak olyan metódusokat hívhat meg, amelyekben az első paraméter egy `System.Object`, míg a második `MouseEventArgs` típusú. Ez utóbbi több olyan tagot tartalmaz, amelyek részletes információval szolgálnak az egér által kiváltott esemény állapotáról:

```
public class MouseEventArgs : EventArgs
{
  private readonly MouseButtons button;
  private readonly int clicks;
  private readonly int delta;
  private readonly int x;
  private readonly int y;

  public MouseEventArgs(MouseButtons button, int clicks, int x,
                        int y, int delta);

  public MouseButtons Button { get; }
  public int Clicks { get; }
  public int Delta { get; }
  public Point Location { get; }
  public int X { get; }
  public int Y { get; }
}
```

A 27.8. táblázat a nyilvános tulajdonságok jelentését ismerteti.

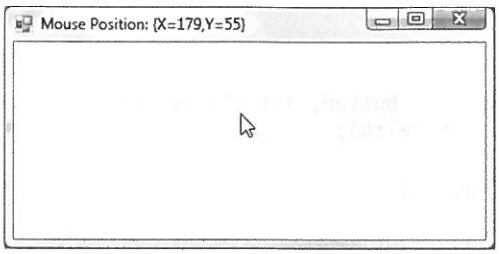
Tulajdonság	Jelentés
`Button`	Ez a tulajdonság lekérdezni, hogy melyik egérgombot nyomták meg, és a `MouseButtons` felsorolt típus megfelelő értékét adja vissza.
`Clicks`	Ez a tulajdonság lekérdezi, hogy hányszor nyomták le és engedték fel az egérgombot.
`Delta`	Ez a tulajdonság az egérgörgő görgetésének előjeles számláló-értékét kérdezi le.
`Location`	Ez a tulajdonság visszaad egy `Point` típust, amely az aktuális X és Y értékeket tartalmazza.
`X`	Ez a tulajdonság lekérdezi az egérkattintások x koordinátáját.
`Y`	Ez a tulajdonság lekérdezi az egérkattintások y koordinátáját.

27.8. táblázat: *A MouseEventArgs típus tulajdonságai*

Implementáljuk a `MouseMove` eseménykezelőt az egér aktuális X és Y pozíciójának megjelenítésére az űrlap címében a `Location` tulajdonság használatával:

```
private void MainWindow_MouseMove(object sender, MouseEventArgs e)
{
  Text = string.Format("Mouse Position: {0}", e.Location);
}
```

Amikor futtatjuk az alkalmazást, és mozgatjuk az egeret az ablakban, az egér pozíciója megjelenik a `MainWindow` címében (lásd a 27.14. ábrát).



27.14. ábra: *Az egérmozgás kezelése*

Az egérgombkattintás meghatározása

A következő kérdés annak meghatározása, hogy a `MouseUp`, a `MouseDown`, a `MouseClick` vagy a `MouseDoubleClick` esemény bekövetkeztekor melyik egérgombbal kattintott a felhasználó. Ha pontosan meg szeretnénk határozni, melyik (azaz a jobb, a bal vagy a középső) gomb volt használatban, meg kell vizsgálnunk a `MouseEventArgs` osztály `Button` tulajdonságát. A `Button` tulajdonság értékét a `MouseButtons` felsorolt típus határozza meg:

```
public enum MouseButtons
{
  Left,
  Middle,
  None,
  Right,
  XButton1,
  XButton2
}
```

Ennek bemutatásához a Properties ablak segítségével kezeljük a `MainWindow` típusban található `MouseUp` eseményt. Az alábbi `MouseUp` eseménykezelő megmutatja, hogy melyik egérgombbal történt kattintás az üzenetdobozban:

```
private void MainWindow_MouseUp (object sender, MouseEventArgs e)
{
  // Melyik egérgombbal kattintottak?
  if(e.Button == MouseButtons.Left)
    MessageBox.Show("Left click!");
  if(e.Button == MouseButtons.Right)
    MessageBox.Show("Right click!");
  if (e.Button == MouseButtons.Middle)
    MessageBox.Show("Middle click!");
}
```

Forráskód A MouseEventApp projektet a forráskódkönyvtár 27. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Reagálás a billentyűzet eseményeire

A Windows-alkalmazások jellemzően számos olyan beviteli vezérlőelemet (pl. a `TextBox`) határoznak meg, ahol a felhasználó az információt a billentyűzet segítségével adhatja meg. Ha a billentyűzet inputját ily módon rögzítjük, nincs szükség a billentyűzetesemények konkrét kezelésére, hiszen a szöveges adatot különböző tulajdonságok segítségével (pl. a `TextBox` típus `Text` tulajdonsága) egyszerűen kiolvashatjuk a vezérlőelemből.

Ha azonban ennél különlegesebb okokból (pl. a billentyűk leütésének szűrésére egy adott vezérlőelemen vagy a billentyűk megnyomásának kezelése magán az űrlapon) ellenőriznünk kell a billentyűzet inputját, az alaposztálykönyvtárak rendelkezésre bocsátják a `KeyUp` és a `KeyDown` eseményt. Ezek az események a `KeyEventHandler` metódusreferenciát használják, ez pedig bármely olyan metódusra mutathat, amelynek első paramétere egy objektum, a második paramétere pedig a `KeyEventArgs`. A típus definíciója az alábbi:

```
public class KeyEventArgs : EventArgs
{
  private bool handled;
  private readonly Keys keyData;
  private bool suppressKeyPress;

  public KeyEventArgs(Keys keyData);

  public virtual bool Alt { get; }
  public bool Control { get; }
  public bool Handled { get; set; }
  public Keys KeyCode { get; }
```

```
    public Keys KeyData { get; }
    public int KeyValue { get; }
    public Keys Modifiers { get; }
    public virtual bool Shift { get; }
    public bool SuppressKeyPress { get; set; }
}
```

A 27.9. táblázat a `KeyEventArgs` által támogatott leglényegesebb tulajdonságokat tartalmazza.

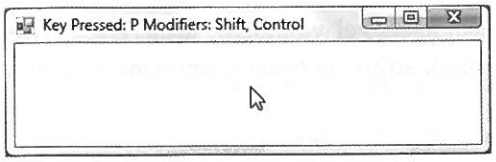
Tulajdonság	Jelentés
`Alt`	Lekérdezi, hogy megnyomták-e az Alt billentyűt.
`Control`	Lekérdezi, hogy megnyomták-e a Ctrl billentyűt.
`Handled`	Lekérdezi vagy beállítja, hogy az esemény kezelése teljes mértékben megtörtént-e a kezelőben.
`KeyCode`	A `KeyDown` és a `KeyUp` esemény billentyűzetkódját kérdezi le.
`Modifiers`	Jelzi, hogy mely váltóbillentyűket (Ctrl, Shift, és/vagy Alt) nyomták meg.
`Shift`	Lekérdezni, hogy megnyomták-e a Shift billentyűt.

27.9. táblázat: *A KeyEventArgs típus tulajdonságai*

Ennek bemutatására tételezzük fel, hogy van egy új, a `KeyUp` eseményt az alábbi módon kezelő, `KeyboardEventApp` nevű Windows Applicaton-projektünk.

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

  private void MainWindow_KeyUp(object sender, KeyEventArgs e)
  {
    Text = string.Format("Key Pressed: {0} Modifiers: {1}",
                  e.KeyCode.ToString(), e.Modifiers.ToString());
  }
}
```

Fordítsuk le és futtassuk a programot. Ekkor már nemcsak azt tudjuk meghatározni, hogy melyik egérgombbal kattintottak, hanem azt is, hogy melyik billentyűt nyomták meg. A 27.15. ábra például a P, a Ctrl és a Shift billentyűk egyidejű megnyomásának eredményét ábrázolja.



27.15. ábra: *A billentyűesemények kezelése*

Forráskód A KeyboardEventApp projektet a forráskódkönyvtár 27. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Párbeszédablakok tervezése

A grafikus felhasználói felülettel ellátott programokban általában a párbeszédablakok szolgálnak az alkalmazásban használt felhasználói adatok bevitelére. A többi, esetleg korábban már használt GUI API-tól eltérően itt nincs „`Dialog`" ősosztály. A Windows Forms alatti párbeszédablakok a `Form` osztályból származtatott egyszerű típusok.

Emellett jó néhány párbeszédablakot fix méretűnek szánnak; ezért általában a `FormBorderStyle` tulajdonságot `FormBorderStyle.FixedDialogra` állítjuk. A párbeszédablakoknál a `MinimizeBox` és a `MaximizeBox` tulajdonság jellemzően hamis, így a párbeszédablakok rögzített konstansként vannak konfigurálva. Végül, ha a `ShowInTaskbar` tulajdonságot hamis értékre állítjuk, akkor megakadályozzuk, hogy az űrlap láthatóvá váljon a Windows-tálcán.

A párbeszédablakok készítésének és kezelésének illusztrálására hozzunk létre egy új Windows Application-projektet CarOrderApp névvel. Nevezzük át a kezdeti `Form1.cs` fájlt a Solution Explorerrel, és adjuk neki a `MainWindow.cs` nevet; majd az űrlaptervezővel hozzuk létre egy egyszerű menüt, amelynek elemei: File > Exit, illetve Tool > Order Automobile. Ha ezzel végeztünk, a Properties ablakban kezeljük az Exit és az Order Automobile almenük `Click` eseményét. A 27.16. ábra a főablak kezdeti megjelenését mutatja.

Az alkalmazás egyszerű bezárásához implementáljuk a File > Exit menü kezelőjét:

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
  Application.Exit();
}
```

A tabulátorsorrend konfigurálása

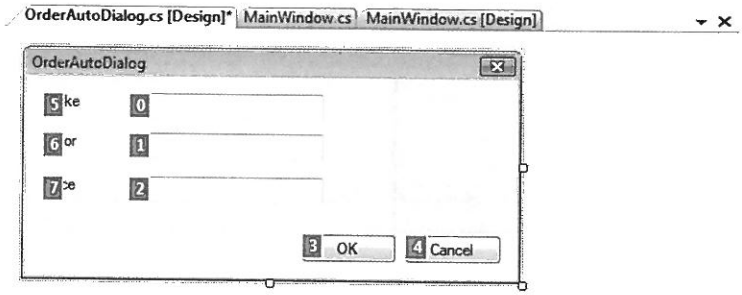
A következőkben formalizáljuk a tabulátorsorrend szerepét. Ha egy űrlap több GUI-elemet tartalmaz, a felhasználók arra számítanak, hogy a vezérlők között a Tab billentyű megnyomásával válthatnak. A vezérlőelemek tabulátorsorrendjének konfigurálásához azonban meg kell ismernünk két kulcsfontosságú tulajdonságot, a `TabStopot` és a `TabIndexet`.

A `TabStop` tulajdonság az igaz vagy a hamis értéket veszi fel attól függően, hogy a GUI-elemet elérhetővé kívánjuk-e tenni a Tab billentyűvel, vagy sem. Ha a `TabStop` tulajdonság értéke az adott vezérlőn igaz, a `TabOrder` tulajdonság az aktivációs sorrendet a tabulálás sorrendjének megfelelően (amely 0 alapú) adja meg. Nézzük meg az alábbi példát:

```
// Tabulálási tulajdonságok konfigurálása.
txtMake.TabIndex = 0;
txtMake.TabStop = true;
```

A tabulátorsorrend-varázsló

Míg a `TabStop` és a `TabIndex` tulajdonságot a Properties ablakban manuálisan is beállíthatjuk, a Visual Studio 2008 IDE egy tabulátorsorrend-varázslót is kínál, amelyet a View > Tab Order kiválasztásával nyithatunk meg (ez a menüelem csak akkor érhető el, ha az űrlaptervező az aktív). Az aktiválást követően a tervezésidejű űrlap megjeleníti az egyes vezérlők aktuális `TabIndex` értékét. Ezeknek az értékeknek a módosításához kattintsunk az elemekre a kívánt sorrendben (lásd a 27.19. ábrát).



27.19. ábra: *A tabulátorsorrend-varázsló*

A tabulátorsorrend-varázslóból való kilépéshez egyszerűen nyomjuk meg az Esc billentyűt.

Az űrlap alapértelmezett beviteli gombjának a beállítása

Számos felhasználói beviteli űrlap (különösen a párbeszédablakok) rendelkezik olyan különleges gombbal, amely automatikusan válaszol, ha a felhasználó megnyomja az Enter billentyűt. Ha azt szeretnénk, hogy ha a felhasználó az aktuális űrlapon megnyomja az Enter billentyűt, akkor lefusson a `btnOK` kattintási esemény kezelője, akkor egyszerűen állítsuk be az űrlap `AcceptButton` tulajdonságát az alábbiak szerint (ugyanez a beállítás a Properties ablakban is elvégezhető):

```
// Az Enter billentyű megnyomása olyan, mintha
// a felhasználó a btnOK gombra kattintott volna.
this.AcceptButton = btnOK;
```

Megjegyzés Néhány űrlap esetében, ha a felhasználó megnyomja az Esc billentyűt, az a Cancel gombra történő kattintást szimulálja. Ez úgy érhető el, hogy a Form `CancelButton` tulajdonságot a Cancel gombra történő kattintást kiváltó `Button` objektumhoz rendeljük.

Párbeszédablakok megjelenítése

Ha egy párbeszédablakot szeretnénk megjeleníteni, először azt kell eldönteni, hogy a párbeszédablak indítása modális vagy ne modális legyen. A modális párbeszédablakokat a felhasználónak még az előtt be kell zárnia, mielőtt visszatérne a párbeszédablakot először indító ablakhoz (pl. a legtöbb About doboz modális). Egy modális párbeszédablak megjelenítéséhez egyszerűen hívjuk meg a `ShowDialog()` metódust a párbeszédablak-objektumból. A nem modális ablakok a `Show()` hívásával jeleníthetők meg. Ez a metódus lehetővé teszi a felhasználó számára a párbeszédablak és a főablak közötti váltogatást (ilyen pl. a Find/Replace párbeszédablak).

Példánkban módosítsuk a `MainWindow` típus Tools> Order Automobile menükezelőjét annak érdekében, hogy az `OrderAutoDialog` objektumot modálisan jelenítse meg. Vizsgáljuk meg a következő kezdeti forráskódot:

```
private void orderAutomobileToolStripMenuItem_Click(object sender,
                                                    EventArgs e)
{
  // Hozzuk létre a párbeszédobjektumot.
  OrderAutoDialog dlg = new OrderAutoDialog();
```

```
    // Jelenítsük meg modális párbeszédablakként, és találjuk ki,
    // melyik gombra kattintottak; ehhez használjuk a DialogResult
    // visszaadott értékét.
    if (dlg.ShowDialog() == DialogResult.OK)
    {
      // Az OK gombra kattintottak, tehát tennünk kell valamit...
    }
  }
```

Megjegyzés A `ShowDialog()` és a `Show()` metódusokat tetszés szerint meghívhatjuk akár a párbeszédablak tulajdonosát képviselő objektum megadásával is (ezt a párbeszédablakot betöltő űrlap ábrázolja). A párbeszédablak tulajdonosának meghatározása létrehozza az űrlaptípusok z-sorrendjét, és (a nem modális párbeszédablakok esetén) biztosítja, hogy a főablak megsemmisítésekor minden, „a tulajdonába tartozó" ablak is eltűnjön.

Ügyeljünk arra, hogy ha létrehozzuk egy `Form`-leszármazott típus egy példányát (jelen esetben az `OrderAutoDialog` típust), a párbeszédablak *nem* látható a képernyőn, egyszerűen csak lefoglalódik a memóriában. Az űrlap valójában csak a `Show()` vagy a `ShowDialog()` metódus hívásakor lesz látható. A `ShowDialog()` visszaadja a gombhoz rendelt `DialogResult` értéket (a `Show()` metódus egyszerűen *void*ot ad vissza).

A `ShowDialog()` visszatérésével az űrlapot a továbbiakban nem lehet látni a képernyőn, ám a memóriában megmarad. Ezért az értékek minden `TextBox` típusból kiolvashatók. Ha azonban az alábbi kódot szeretnénk lefordítani:

```
private void orderAutomobileToolStripMenuItem_Click(object sender,
                                                    EventArgs e)
{
  // Hozzuk létre a párbeszédobjektumot.
  OrderAutoDialog dlg = new OrderAutoDialog();

  // Jelenítsük meg modális párbeszédablakként, és találjuk ki,
  // melyik gombra kattintottak; ehhez használjuk a DialogResult
  // visszaadott értékét.
  if (dlg.ShowDialog() == DialogResult.OK)
  {
    // Minden szövegdobozban van érték? Fordítási hiba!
    string orderInfo = string.Format("Make: {0}, Color: {1},
          Cost: {2}", dlg.txtMake.Text, dlg.txtColor.Text,
          dlg.txtPrice.Text);
    MessageBox.Show(orderInfo, "Information about your order!");
  }
}
```

akkor fordítási hibaüzenetet kapunk. Ennek az az oka, hogy a Visual Studio 2008 a Forms-tervezőhöz adott vezérlőelemeket az osztály *privát* tagváltozóiként deklarálja. Ezt a tényt megerősíthetjük például akkor, ha meg szeretnénk nyitni az `OrderAutoDialog.Designer.cs` fájlt. Bár a kifogástalan párbeszédablakok, ha ezekben a szövegdobozokban lévő értékek megadásához vagy felvételéhez nyilvános tulajdonságokat adnak, megőrizhetik az egységbe zárást, mi ennél rövidebb utat választunk, és a `public` kulcsszóval egyszerűen újradefiniáljuk őket:

```
partial class OrderAutoDialog
{
  ...
  // Az űrlap tagváltozóinak meghatározása a tervező által
  // fenntartott fájlban történik.
  public System.Windows.Forms.TextBox txtMake;
  public System.Windows.Forms.TextBox txtColor;
  public System.Windows.Forms.TextBox txtPrice;
}
```

Ekkor már lefordíthatjuk és futtathatjuk az alkalmazást. A párbeszédablak elindításakor a beviteli adatok egy üzenetdobozban vannak (ha az OK gombra kattintottunk).

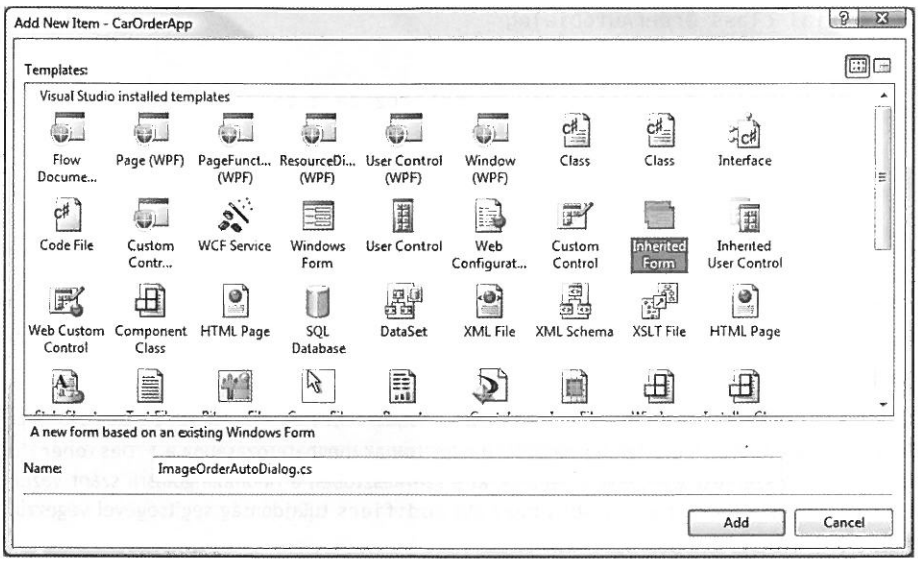
Megjegyzés A vezérlőelemek hozzáférési módosítóinak meghatározásához a `*.Designer.cs` fájl közvetlen szerkesztése helyett jelöljük ki a szerkesztőben a finomhangolásra szánt vezérlőelemet, majd a Properties ablakban található `Modifiers` tulajdonság segítségével végezzük el a műveletet.

Az űrlapok származtatása

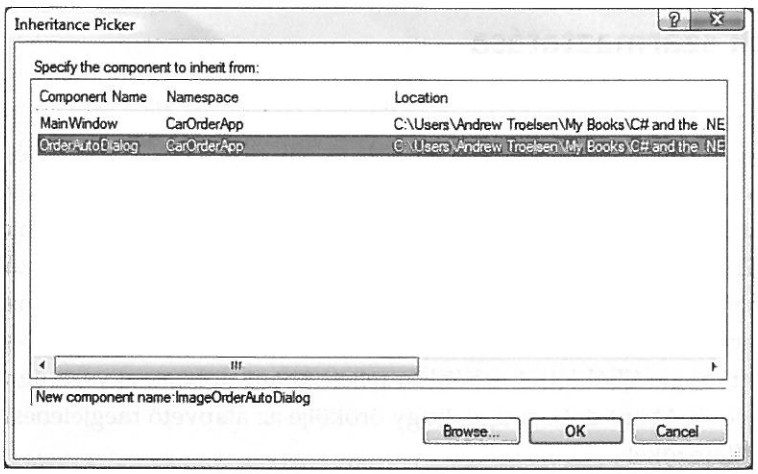
Eddig minden egyes egyedi ablakunk/párbeszédablakunk közvetlenül a `System.Windows.Forms.Form` leszármazottja volt. A Windows Forms-fejlesztés egyik érdekes aspektusa, hogy a `Form` típusok a származtatott `Form` típusok ősosztályaként is működhetnek. Tételezzük fel például, hogy létrehoztunk egy olyan .NET-kódkönyvtárat, amely tartalmazza a vállalatunkra vonatkozó alapvető párbeszédablakokat. Később úgy döntünk, hogy az About doboz kissé semmitmondó, ezért szeretnénk hozzáadni a vállalat emblémájának 3D-s képét. Ahelyett, hogy újból létrehoznánk a teljes About dobozt, egyszerűen bővítsük az eredeti About dobozt úgy, hogy örökölje az alapvető megjelenési és működési jellemzőket:

```
// A ThreeDAboutBox "az egy" AboutBox.
public class ThreeDAboutBox : AboutBox
{
  // A vállalat logójának renderéléséhez adjunk hozzá forráskódot...
}
```

Az űrlapszármaztatás működésének megtekintéséhez a Project > Add Form menüelem segítségével szúrjunk a projektbe egy új űrlapot. Ezúttal azonban válasszuk ki a Inherited Form ikont, és az új űrlapnak adjuk az `ImageOrderAutoDialog.cs` nevet (lásd a 27.20. ábrát).



27.20. ábra: *Származtatott űrlap hozzáadása a projekthez*

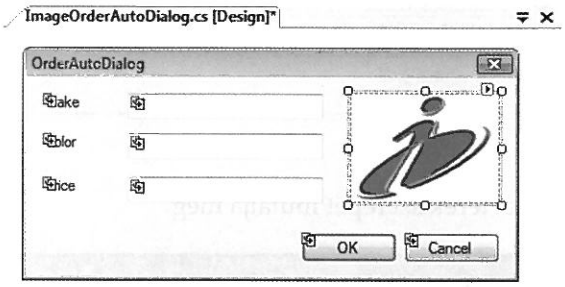


27.21. ábra: *Az Inheritance Picker párbeszédablak*

Ezzel megjelenítjük az Inheritance Picker párbeszédablakot, amely megmutatja az aktuális projekt valamennyi űrlapját. A Browse gomb lehetővé teszi a külső .NET-szerelvényekből történő űrlapválasztást is. Most egyszerűen válasszuk ki az `OrderAutoDialog` típust (lásd a 27.21. ábrát).

Megjegyzés A projektet legalább egyszer le kell fordítani annak érdekében, hogy az űrlapok láthatók legyenek az Inheritance Picker párbeszédablakban, hiszen ez az eszköz olvassa a szerelvény-metaadatokat a lehetőségeink megjelenítéséhez.

Az OK gombra kattintva azt látjuk, hogy a vizuális tervezőeszközök megjelenítik a szülők valamennyi alapvető vezérlőelemét, s a szülők mindegyike rendelkezik egy kicsi, a vezérlőelem bal felső sarkában látható nyíllal (a származtatás szimbolizálására). A származtatott párbeszédablak befejezéséhez keressük meg a Toolbox Common Controls részében található `PictureBox` vezérlőelemet, és adjuk a származtatott űrlaphoz. Ezt követően az Image tulajdonság segítségével válasszunk ki egy tetszőleges képfájlt. Az 27.22. ábrán egy lehetséges felhasználói felület látható, ahol az Intertech Training logóját használtuk.



27.22. ábra: *Az ImageOrderAutoDialog típus*

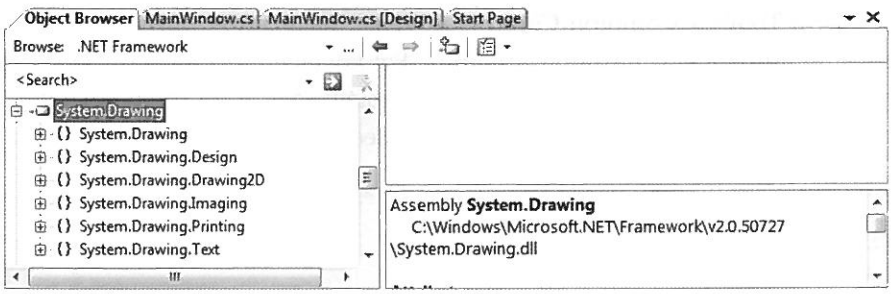
Most már módosíthatjuk a Tools > Order Automobile kattintási esemény kezelőjét, hogy létrehozzuk a származtatott osztály egy példányát az `OrderAutoDialog` ősosztály helyett:

```
private void orderAutomobileToolStripMenuItem_Click(object sender,
                                                    EventArgs e)
{
  // Hozzuk létre a származtatott párbeszédobjektumot.
  ImageOrderAutoDialog dlg = new ImageOrderAutoDialog();
  ...
}
```

Forráskód A CarOrderApp projektet a forráskódkönyvtár 27. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

GDI+-alapú grafikus adatok renderelése

Számos GUI-alkalmazáshoz szükség van az ablakok felületén megjelenítendő grafikus adatok dinamikus generálására. Ha például kiválasztottunk egy rekordhalmazt egy relációs adatbázisból, és szeretnénk egy olyan kördiagramot (vagy oszlopdiagramot) renderelni, amely megjeleníti a kiválasztott elemeket. Vagy például, ha a .NET platform segítségével szeretnénk valamilyen régi videojátékot újra elkészíteni. Ha az adatokat, a céltól függetlenül, grafikusan szeretnénk renderelni egy Windows Forms-alkalmazásban, arra a GDI+ a legmegfelelőbb API. Ez a technológia szorosan kapcsolódik a `System.Drawing.dll` szerelvényhez, amely számos névteret határoz meg (lásd a 27.23. ábrát).



27.23. ábra: *A System.Drawing.dll névterei*

A 27.10. táblázat az egyes GDI+-névterek szerepét mutatja meg.

Névtér	Jelentés
`System.Drawing`	Ez az alapvető GDI+-névtér az alapszintű előkészítés számos típusát (betűtípusok, tollak, egyszerű ecsetek stb.), valamint a `Graphics` típust definiálja.
`System.Drawing.Drawing2D`	Ez a névtér biztosítja a speciálisabb 2D-s/vektorgrafikus funkcionalitáshoz használt típusokat (pl. színátmenetes ecsetek, geometriai transzformációk stb.).
`System.Drawing.Imaging`	Ez a névtér határozza meg azokat a típusokat, amelyek lehetővé teszik a grafikus képek kezelését (pl. a színskála módosítását, a képi metaadatok lekérdezését, a metafájlok kezelését stb.).

Névtér	Jelentés
`System.Drawing.Printing`	Ez a névtér határozza meg azokat a típusokat, amelyek lehetővé teszik a képek renderelését a nyomtatott oldalra, a kommunikációt magával a nyomtatóval, valamint a nyomtatott verzió megjelenésének formázását.
`System.Drawing.Text`	Ez a névtér teszi lehetővé a betűtípus-gyűjtemények kezelését.

27.10. táblázat: *Alapvető GDI+-névterek*

A System.Drawing névtér

A GDI+-alkalmazások programozásakor használt típusok igen nagy része a `System.Drawing` névtérben található. Elvárásainknak megfelelően vannak olyan típusok, amelyek képeket, ecseteket, tollakat és betűkészleteket képviselnek. Emellett a `System.Drawing` számos kapcsolódó segédtípust (pl. a `Color`, `Print` és `Rectangle`) is meghatároz. A 27.11. táblázat néhány ilyen (de nem az összes) alapvető típust tartalmaz.

Típus	Jelentés
`Bitmap`	Ez a típus foglalja magába a képadatokat (`*.bmp` és egyebek).

`Brush`
`Brushes`
`SolidBrush`
`SystemBrushes`
`TextureBrush`
	Az ecsetobjektumok a grafikai alakzatok, mint például a téglalapok, az ellipszisek vagy a sokszögek belsejének kitöltésére szolgálnak.

`BufferedGraphics`	Ez a típus biztosítja a grafikai puffert a kettős pufferéléshez, amely a megjelenített felület újrarajzolásakor létrejövő vibrálás csökkentését vagy megszüntetését szolgálja.
`Color` `SystemColors`	A `Color` és a `SystemColors` típusok határozzák meg azokat a statikus írásvédett tulajdonságokat, amelyek a különböző tollak/ecsetek szerkesztéséhez használt színek megszerzését szolgálják.
`Font` `FontFamily`	A `Font` típus foglalja magába az adott betűtípus jellemzőit (mint pl. a típus neve, félkövér vagy dőlt betűs, pontméret stb.). A `FontFamily` biztosítja a hasonló megjelenésű, de stílusban eltérő betűtípusok csoportjainak absztrakcióját.

Típus	Jelentés
`Graphics`	Ez az alapvető osztály egy érvényes rajzfelületet, valamint számos, a szöveg, a képek és a geometriai minták renderelésével használt metódust reprezentál.
`Icon` `SystemIcons`	Ezek az osztályok egyedi ikonokat, valamint a rendszer által biztosított, standard ikonok készletét képviselik.
`Image` `ImageAnimator`	Az `Image` olyan absztrakt ősosztály, amely a `Bitmap`, az `Icon` és a `Cursor` típusok funkcionalitását biztosítja. Az `ImageAnimator` biztosítja az `Image`-leszármazott típusok egy adott intervallumban történő iterálásának a lehetőségét.

`Pen`
`Pens`
`SystemPens`
	Tollak, vonalak és görbék rajzolására szolgálnak. A `Pens` típus számos olyan statikus tulajdonságot definiál, amelyek egy adott szín új `Pen` típusát adják vissza.

`Point PointF`	Ezek a struktúrák egy (x,y) koordinátát képeznek le a mögöttes egész vagy lebegőpontos számra.
`Rectangle` `RectangleF`	Ezek a struktúrák egy téglalap méreteit adják meg (szintén a mögöttes egész vagy lebegőpontos számba leképezve).
`Size SizeF`	Ezek a struktúrák egy adott magasságot/szélességet határoznak meg (szintén a mögöttes egész vagy lebegőpontos számba leképezve).
`StringFormat`	Ez a típus a szöveges elrendezés (mint pl. az igazítás, a sorköz stb.) tulajdonságainak egységbe zárására szolgál.
`Region`	Ez a típus a téglalapokból és az útvonalakból álló geometrikus képek belső jellemzőit írja le.

27.11. táblázat: *A System.Drawing névtér alapvető típusai*

A Graphics típus szerepe

A `System.Drawing.Graphics` osztály az átjáró a GDI+-renderelő funkcionalitásához. Ez az osztály nemcsak a rajzfelületet (pl. az űrlap felületét, a vezérlőelemek felületét vagy a memóriaterületeket) tartalmazza, de több tucat olyan tagot is meghatároz, amelyek lehetővé teszik a szövegek, a képek (ikonok, bittérképek stb.) és számos geometriai minta renderelését is. A 27.12. táblázat részleges listát ad a tagokról.

Metódus	Jelentés

`FromHdc()`
`FromHwnd()`
`FromImage()`
	Ezek a statikus metódusok biztosítják az érvényes `Graphics` objektumok egy adott képből (pl. ikonból, bittérképből stb.) vagy GUI-céleszközből történő megszerzését.

`Clear()`	Ez a metódus adott színnel tölti ki a `Graphics` objektumot, és törli a folyamat aktuális rajzfelületét.

`DrawArc ()`
`DrawBeziers()`
`DrawCurve()`
`DrawEllipse()`
`DrawIcon()`
`DrawLine()`
`DrawLines()`
`DrawPie()`
`DrawPath()`
`DrawRectangle()`
`DrawRectangles()`
`DrawString()`
	Ezek a metódusok egy adott kép vagy geometrikus minta renderelésére szolgálnak. Valamennyi `DrawXXX()` metódus megköveteli a GDI+ `Pen` objektumok használatát.

`FillEllipse()`
`FillPie()`
`FillPolygon()`
`FillRectangle()`
`FillPath()`
	Ezek a metódusok egy adott geometrikus alakzat kitöltésére szolgálnak. Valamennyi `FillXXX()` metódus megköveteli a GDI+ `Brush` objektumok használatát.

27.12. táblázat: *A Graphics osztály tagjai*

A `Graphics` osztály nem hozható létre közvetlenül a `new` kulcsszó segítségével, ugyanis nincsenek nyilvánosan meghatározott konstruktorai. Ez esetben felmerül a kérdés: hogyan kapunk érvényes `Graphics` objektumot?

Graphics objektumok megszerzése a Paint eseményen keresztül

A `Graphics` objektumok legáltalánosabb megszerzési módja a `Paint` esemény kezelése abban az ablakban, amelyet renderelni szeretnénk a Visual Studio 2008 Properties ablakának használatával. Ezt az eseményt a `PaintEventHandler` metódusreferencia határozza meg. Ez a metódusreferencia bármely olyan metódusra képes rámutatni, amelynek első paramétere `System.Object`, a második pedig `PaintEventArgs`.

A PaintEventArgs paraméter tartalmazza azt a Graphics objektumot, amelyet az űrlap felületére kell renderelni. Ennek illusztrálására hozzunk létre egy új Windows Application-projektet PaintEventApp névvel. A Solution Explorer segítségével nevezzük át a kezdeti Form.cs fájlt MainWindow.cs-re, és kezeljük a Paint eseményt a Properties ablakban. Az eredmény az alábbi két kódcsonk lesz:

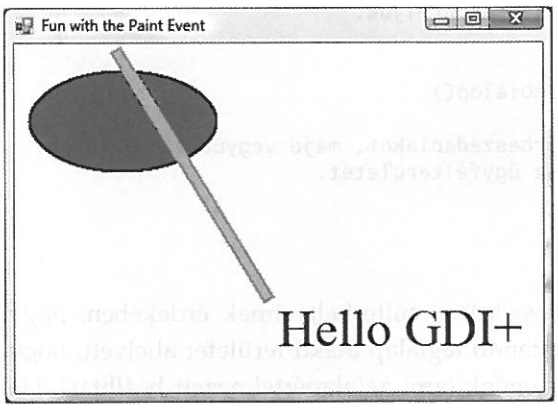
```
public partial class MainWindow : Form
{
  public MainWindow()
  {
    InitializeComponent();
  }

  private void MainWindow_Paint(object sender, PaintEventArgs e)
  {
  }
}
```

A Paint esemény kezelése után, kérdés, hogy mikor fog bekövetkezni. A Paint esemény mindig bekövetkezik akkor, amikor az ablak „piszkos" lesz. Az ablak akkor „piszkos", amikor átméretezzük, amikor egy másik ablak (részben vagy egészben) eltakarja, vagy amikor kis méretűvé tettük, majd visszaállítottuk az előző méretre. Minden ilyen esetben, amikor az űrlapot újra kell rajzolni, a .NET platform automatikusan meghívja a Paint eseménykezelőt. Nézzük meg a MainWindow_Paint() metódus alábbi implementációját:

```
private void MainWindow_Paint(object sender, PaintEventArgs e)
{
  // Vegyük az aktuális űrlap grafikus objektumát.
  Graphics g = e.Graphics;

  // Rajzoljunk egy kört.
  g.FillEllipse(Brushes.Blue, 10, 20, 150, 80);

  // Egyedi betűtípussal hozzunk létre egy sztringet.
  g.DrawString("Hello GDI+", new Font("Times New Roman", 30),
               Brushes.Red, 200, 200);

  // Egyedi tollal húzzunk egy vonalat.
  using (Pen p = new Pen(Color.YellowGreen, 10))
  {
    g.DrawLine(p, 80, 4, 200, 200);
  }
}
```

Ha a `Graphics` objektumot megszereztük a bemeneti `PaintEventArgs` paraméterből, a következő lépés a `FillEllipse()` metódus hívása. Ennek a metódusnak (mint minden `Fill` prefixumos metódusnak) az első paramétere `Brush`-leszármazott kell, hogy legyen. Bár a `System.Drawing.Drawing2D` névtérből számos érdekes ecsetobjektumot hozhatnánk létre (`HatchBrush`, `LinearGradientBrush` stb.), a `Brushes` segédosztály egyszerű hozzáférést biztosít számos egyszínű ecsettípushoz.

A következő lépésben hívjuk meg a `DrawString()` metódust, amelynek első paramétere egy renderelendő sztring kell, hogy legyen. Ennek tükrében a GDI+ biztosítja a `Font` típust, amely nemcsak a szöveges adatok renderelésével használt betűtípus nevét adja meg, hanem a kapcsolódó jellemzőket is, mint például a pontméretet (ez jelen esetben 30). A `DrawString()` egy `Brush` típust is igényel, mert ami a GDI+-t illeti, a „Hello GDI+" a képernyőn kitöltendő geometrikus minták egyszerű gyűjteményét jelenti. Végül pedig egy egyedi `Pen` típust használó, 10 képpontnyi széles vonal renderelésével hívjuk meg a `DrawLine()` metódust. A 27.24. ábra ennek az renderelési logikának a kimenetét ábrázolja.



27.24. ábra: *Egyszerű GDI+ renderelési művelet*

Megjegyzés Az előző kódban egyértelműen rendelkezünk a `Pen` objektummal. Aranyszabályként elmondhatjuk, hogy ha közvetlenül hozunk létre `IDisposable` metódust implementáló GDI+-típust, amint készen vagyunk az objektummal, rögtön hívjuk meg a `Dispose()` metódust. Így a lehető leghamarabb felszabadíthatjuk a mögöttes erőforrásokat. Ha nem e szerint járunk el, az erőforrásokat végül nem determinisztikus módon a szemétgyűjtő szabadítja fel.

Az űrlap felületének érvénytelenítése

A Windows Forms-alkalmazás során szükségünk lehet a `Paint` eseménynek a kódból történő közvetlen kiváltására ahelyett, hogy arra várnánk, hogy az ablak a végfelhasználói műveletek során „magától piszkos" lesz. Ha például olyan programot készítünk, amely lehetővé teszi a felhasználó számára, hogy egyedi párbeszédablak segítségével kiválasszon néhány előre meghatározott képet. A párbeszédablakból való kilépést követően meg kell rajzolnunk az újonnan kijelölt képet az űrlap ügyfélterületére. Nyilvánvaló, hogy ha arra várnánk, hogy az ablak „automatikusan váljon piszkossá", a felhasználó nem látná a változást egészen addig, amíg át nem méretezné az ablakot, vagy egy másik ablak el nem takarná az előzőt. Annak kikényszerítésére, hogy az ablak programozottan újrafesse önmagát, hívjuk meg az `Invalidate()` metódust:

```
public partial class MainForm: Form
{
  ...
  private void MainForm_Paint(object sender, PaintEventArgs e)
  {
    Graphics g = e.Graphics;
    // A megfelelő képet itt rendereljük.
  }

  private void GetImageFromDialog()
  {
    // Jelenítsük meg a párbeszédablakot, majd vegyük az új képet.
    // Fessük újra az egész ügyfélterületet.
    Invalidate();
  }
}
```

Az `Invalidate()` metódus többszörösen túlterhelt annak érdekében, hogy meghatározhassunk egy újrafestendő téglalap alakú területet ahelyett, hogy az egész ügyfélterületet újrafestenénk (ami az alapértelmezett beállítás). Ha az ügyfélterületnek csak a bal felső téglalap alakú területét szeretnénk frissíteni, írjuk az alábbiakat:

```
// Az űrlap adott téglalap alakú területének újrafestése.
private void UpdateUpperArea()
{
  Rectangle myRect = new Rectangle(0, 0, 75, 150);
  Invalidate(myRect);
}
```

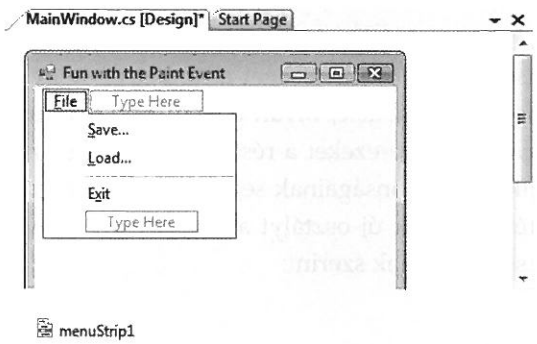
Forráskód A PaintEventApp projektet a forráskódkönyvtár 27. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Teljes Windows Forms-alkalmazás létrehozása

A Windows Forms és a GDI+ API-k bevezetésének lezárásaként hozzunk létre egy olyan teljes GUI-alkalmazást, amely számos, a jelen fejezetben tárgyalt módszert mint összefüggő egészet tartalmaz. A létrehozandó program egy olyan alapvető rajzolóprogram lesz, amely lehetővé teszi a felhasználó számára, hogy kétfajta alakzat (az egyszerűség kedvéért egy kör és egy téglalap) közül válasszon, és a kiválasztott alakzatot tetszőleges színben elhelyezze az űrlapon. Emellett lehetővé tesszük, hogy a végfelhasználó a képeket helyi fájlként a merevlemezre mentse, későbbi felhasználásra az objektumsorosító szolgáltatásokkal.

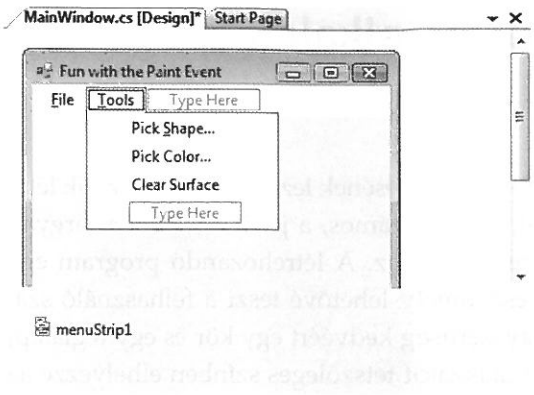
A főmenürendszer készítése

Először hozzunk létre egy új Windows Forms-alkalmazást MyPaintProgram névvel, és nevezzük át a kezdeti `Form.cs` fájlt `MainWindow.cs`-re. Tervezzük meg a kezdeti ablak menürendszerét, amely támogatja a Save, a Load és az Exit almenüket biztosító, felső File menüt (lásd a 27.25. ábrát).



27.25. ábra: *A File menürendszer*

Ezt követően hozzuk létre a második felső menüt, a Tools menüt, amelynek segítségével kiválasztható a rendereléshez használatos alakzat és szín, valamint törölhetők a grafikus adatok az űrlapról (lásd a 27.26. ábrát).



27.26. ábra: *A Tools menürendszer*

Végül pedig kezeljük ezeknek az alelemeknek a `Click` eseményét. A példa során minden egyes kezelőt implementálni fogunk, a File > Exit menükezelőt viszont az `Application.Exit()` metódus hívásával is lezárjuk:

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
  Application.Exit();
}
```

A ShapeData típus meghatározása

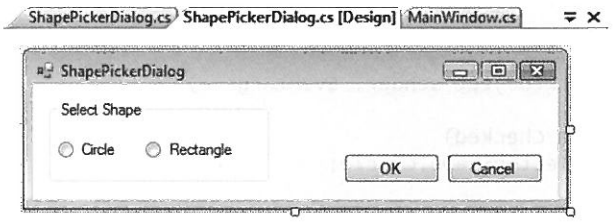
Az alkalmazásunk lehetővé teszi, hogy a végfelhasználó két adott színű, előre meghatározott alakzat közül válasszon. Mivel lehetőséget adunk arra, hogy a felhasználó a grafikus adatokat fájlként elmentse, olyan egyedi osztálytípust kell meghatároznunk, amely magába foglalja ezeket a részleteket. Az egyszerűség kedvéért ezt a C# automatikus tulajdonságainak segítségével végezzük el (lásd az előző kötet 13. fejezetét). Adjunk új osztályt a `ShapeData.cs` nevű projekthez. Implementáljuk a típust az alábbiak szerint:

```
[Serializable]
class ShapeData
{
  // A rajzolandó alakzat bal felső része.
  public Point UpperLeftPoint { get; set; }

  // A rajzolandó alakzat aktuális színe.
  public Color Color { get; set; }

  // Az alakzat típusa.
  public SelectedShape ShapeType { get; set; }
}
```

Ezen a ponton a `ShapeData` három automatikus tulajdonságot használ fel, amelyek közül kettő (a `Point` és a `Color`) meghatározása a `System.Drawing` névtérben található, ezért ne felejtsük ezt a névteret a forráskódba importálni. Ez a típus `[Serializable]` attribútummal is rendelkezik. Egy későbbi lépésben majd konfigurálni fogjuk a `MainWindow` típust a `ShapeData` típusok listájának karbantartásához, amelyek az objektumsorosító szolgáltatások segítségével elmenthetők (lásd a 21. fejezetet).

A ShapePickerDialog típus meghatározása

Annak érdekében, hogy a végfelhasználó a kör és a téglalap alakú képtípusok közül választani tudjon, most egy egyszerű, egyedi párbeszédablakot hozunk létre ShapePickerDialog névvel (szúrjuk be ezt az új űrlapot). A kötelező OK és Cancel gombon kívül (amelyek mindegyikéhez a megfelelő `DialogResult` értéket rendeltük), a párbeszédablak olyan önálló `GroupBox` típust is használni fog, amely két `RadioButtont` tartalmaz. Ezek az objektumok a `radioButtonCircle` és a `radioButtonRect`. A 27.27. ábra egy ilyen lehetséges megoldást ábrázol.



27.27. ábra: *A ShapePickerDialog típus*

Nyissuk meg a párbeszédablakunk forráskódját, ehhez jobb egérgombbal kattintsunk az űrlaptervezőre, majd válasszuk a View Code menüelemet. A `MyPaintProgram` névtérben határozzunk meg egy olyan felsorolt típust (`SelectedShape` névvel), amely minden lehetséges alakzatnevet definiál:

```
public enum SelectedShape
{
  Circle, Rectangle
}
```

Az alábbiak szerint módosítsuk az aktuális `ShapePickerDialog` osztálytípust:

- Adjunk a `SelectedShape` típushoz automatikus tulajdonságot. A hívó ezen tulajdonság segítségével meg tudja határozni, melyik alakzatot kell renderelni.
- A Properties ablakban kezeljük az OK gombhoz kapcsolódó `Click` eseményt.
- Implementáljuk az eseménykezelőt annak meghatározására, hogy a kör rádiógombját jelölték-e ki (a `Checked` tulajdonságon keresztül). Amennyiben igen, a `currentShape` változó legyen `SelectedShape.Circle` típus, egyéb esetben a tagváltozó legyen `SelectedShape.Rectangle`.

A teljes kód a következőképpen néz ki:

```
public partial class ShapePickerDialog : Form
{
  public SelectedShape SelectedShape { get; set; }
  public ShapePickerDialog()
  {
    InitializeComponent();
  }

  private void btnOK_Click(object sender, EventArgs e)
  {
    if (radioButtonCircle.Checked)
      SelectedShape = SelectedShape.Circle;
    else
      SelectedShape = SelectedShape.Rectangle;
  }
}
```

Ezzel elkészítettük a program infrastruktúráját. Most egyszerűen implementáljuk a főablak többi menüeleméhez kapcsolódó `Click` eseménykezelőket.

Infrastruktúra hozzáadása a MainWindow típushoz

A főablak szerkesztésének folytatásaként adjunk három új tagváltozót az űrlaphoz, amelyek lehetővé teszik, hogy nyomon kövessük a választott alakzatot (a `SelectedShape` felsorolt típuson keresztül), a választott színt (a `System.Drawing.Color` típuson keresztül), valamint a renderelt képek mindegyéket a generikus `List<T>`-n keresztül:

```
public partial class MainWindow : Form
{
  // Aktuálisan rajzolandó alakzat / szín.
  private SelectedShape currentShape;
  private Color currentColor = Color.DarkBlue;

  // Ez tartja karban valamennyi ShapeData típust.
  private List<ShapeData> shapes = new List<ShapeData>();
  ...
}
```

A Properties ablakban kezeljük a `Form`-leszármazott típushoz kapcsolódó `MouseDown` és `Paint` eseményeket. Ezeket majd egy későbbi lépésben implementáljuk. Egyelőre az a lényeges, hogy az IDE az alábbi kódcsonkot generálta:

```
private void MainWindow_Paint(object sender, PaintEventArgs e)
{
}

private void MainWindow_MouseClick(object sender, MouseEventArgs e)
{
}
```

A Tools menü funkcionalitásának implementálása

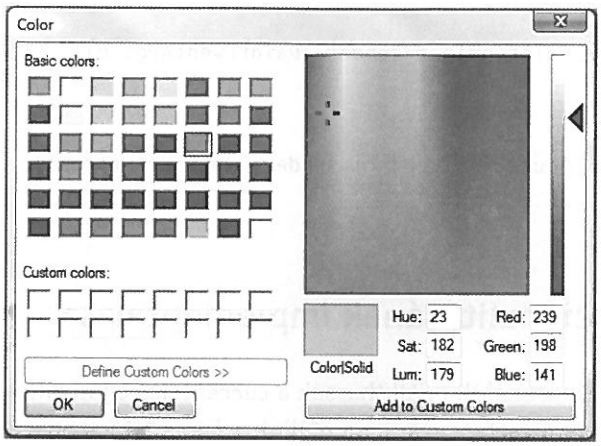
Annak érdekében, hogy a felhasználók beállíthassák a `currentShape` tagváltozót, a párbeszédablak megjelenítéséhez, implementáljuk a Tools > Pick Shape menüelemhez kapcsolódó `Click` kezelőt, és a felhasználó választásának megfelelően rendeljük hozzá ezt a tagváltozót:

```
private void pickShapeToolStripMenuItem_Click(object sender,
                                              EventArgs e)
{
  // Töltsük be a párbeszédablakot, és állítsuk be a megfelelő
  // alakzattípust.
  ShapePickerDialog dlg = new ShapePickerDialog();
  if (DialogResult.OK == dlg.ShowDialog())
  {
    currentShape = dlg.SelectedShape;
  }
}
```

Annak érdekében, hogy a felhasználó beállíthassa a `currentColor` tagváltozót, a `System.Windows.Forms.ColorDialog` típus használatához implementáljuk a Tools > Pick Color menüelemhez kapcsolódó kattintási esemény kezelőjét:

```
private void pickColorToolStripMenuItem_Click(object sender,
                                              EventArgs e)
{
  ColorDialog dlg = new ColorDialog();
  if (dlg.ShowDialog() == DialogResult.OK)
  {
    currentColor = dlg.Color;
  }
}
```

Ha a programot mostani állapotában szeretnénk futtatni, és kijelölnénk a Tools > Pick Color menüelemet, akkor a 27.28. ábrán látható párbeszédablak jelenne meg.



27.28. ábra: *A ColorDialog típus készlete*

Végül a `List<T>` tagváltozó tartalmának kiürítéséhez és a `Paint` esemény `Invalidate()` metóduson keresztüli, programozottan történő kiváltáshoz implementáljuk a Tools > Clear Surface menükezelőt:

```
private void clearSurfaceToolStripMenuItem_Click(object sender,
                                                 EventArgs e)
{
  shapes.Clear();

  // Ez kiváltja a Paint eseményt.
  Invalidate();
}
```

A grafikus kimenet rögzítése és renderelése

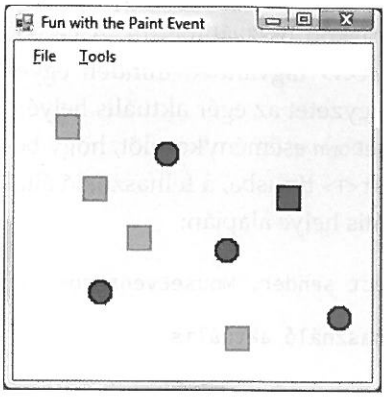
Mivel az `Invalidate()` hívása elsüti a `Paint` eseményt, mindenképpen szükség van a `Paint` eseménykezelőben lévő forráskód összeállítására. A cél az, hogy végigiteráljunk a (most még üres) `List<T>` tagváltozó minden egyes elemén, és rendereljünk egy kört vagy egy négyzetet az egér aktuális helyén. Az első lépés az, hogy implementáljuk a `MouseDown` eseménykezelőt, hogy beszúrjuk az új `ShapeData` típust a generikus `List<T>` típusba, a felhasználó által kijelölt szín, az alakzat típusa és az egér aktuális helye alapján:

```
private void MainWindow_MouseClick(object sender, MouseEventArgs e)
{
  // Készítsünk ShapeData típust a felhasználó aktuális
  // választása alapján.
  ShapeData sd = new ShapeData();
  sd.ShapeType = currentShape;
  sd.Color = currentColor;
  sd.UpperLeftPoint = new Point(e.X, e.Y);

  // Adjuk a List<T> típushoz, és kényszerítsük az űrlapot,
  // hogy újrafesse önmagát.
  shapes.Add(sd);
  Invalidate();
}
```

Implementáljuk a `Paint` eseménykezelőt az alábbiak szerint:

```
private void MainWindow_Paint(object sender, PaintEventArgs e)
{
  // Vegyük az ablak Graphics típusát.
  Graphics g = e.Graphics;

  // Rendereljük az alakzatokat a választott színben.
  foreach (ShapeData s in shapes)
  {
    // Rendereljünk egy 20 x 20 képpont nagyságú téglalapot vagy
    // kört a megfelelő szín felhasználásával.
    if (s.ShapeType == SelectedShape.Rectangle)
      g.FillRectangle(new SolidBrush(s.Color),
                      s.UpperLeftPoint.X,
                      s.UpperLeftPoint.Y, 20, 20);
    else
      g.FillEllipse(new SolidBrush(s.Color),
                    s.UpperLeftPoint.X,
                    s.UpperLeftPoint.Y, 20, 20);
  }
}
```

Ha most futtatnánk az alkalmazást, immár tetszés szerinti számú, különböző színű alakzatot tudnánk renderelni (lásd a 27.29. ábrát).



27.29. ábra: *A MyPaintProgram működés közben*

A sorosítási logika implementálása

A projekt utolsó fázisában implementáljuk a File > Save... és a File > Load... menüelemek kattintásiesemény-kezelőjét. Mivel a `ShapeData` rendelkezik a `[Serialization]` attribútummal (és mivel maga a `List<T>` sorosítható), a Windows Forms `SaveFileDialog` típus segítségével gyorsan kimenthetjük az aktuális grafikus adatokat. A `System.Runtime.Serialization.Formatters.Binary` és a `System.IO` névterek meghatározásához először egészítsük ki a `using` direktívákat.

```
// A bináris formázóhoz.
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
```

Ezzel frissítsük a File > Save... kezelőt az alábbiak szerint:

```
private void saveToolStripMenuItem_Click(object sender, EventArgs e)
{
  using (SaveFileDialog saveDlg = new SaveFileDialog())
  {
    // Konfiguráljuk a mentés párbeszédablak megjelenését és
    // működését.
    saveDlg.InitialDirectory = ".";
    saveDlg.Filter = "Shape files (*.shapes)|*.shapes";
    saveDlg.RestoreDirectory = true;
    saveDlg.FileName = "MyShapes";
```

```
      // Ha az OK gombra kattintanak, megnyílik az új
      // fájl, és megtörténik a List<T> sorosítása.
      if (saveDlg.ShowDialog() == DialogResult.OK)
      {
        Stream myStream = saveDlg.OpenFile();
        if ((myStream != null))
        {
          // Mentsük el az alakzatokat.
          BinaryFormatter myBinaryFormat = new BinaryFormatter();
          myBinaryFormat.Serialize(myStream, shapes);
          myStream.Close();
        }
      }
    }
  }
```

A File > Load eseménykezelő megnyitja a kijelölt fájlt, és a Windows Forms `OpenFileDialog` típusa segítségével visszaállítja az adatokat a `List<T>` tagváltozóba:

```
private void loadToolStripMenuItem_Click(object sender, EventArgs e)
{
  using (OpenFileDialog openDlg = new OpenFileDialog())
  {
    openDlg.InitialDirectory = ".";
    openDlg.Filter = "Shape files (*.shapes)|*.shapes";
    openDlg.RestoreDirectory = true;
    openDlg.FileName = "MyShapes";

    if (openDlg.ShowDialog() == DialogResult.OK)
    {
      Stream myStream = openDlg.OpenFile();
      if ((myStream != null))
      {
        // Vegyük az alakzatokat.
        BinaryFormatter myBinaryFormat = new BinaryFormatter();
        shapes =
            (List<ShapeData>)myBinaryFormat.Deserialize(myStream);
        myStream.Close();
        Invalidate();
      }
    }
  }
}
```

A sorosítás logikája már ismerős a 21. fejezet alapján. Érdemes hangsúlyoznunk, hogy a `SaveFileDialog` és az `OpenFileDialog` típusok mindegyike támogatja a meglehetősen rejtélyes sztringértéket tartalmazó `Filter` tulajdonságot.

Ez a szűrő számos beállítást, mint például a fájlok kiterjesztését (`*.shapes`) vezérli a mentés/megnyitás párbeszédablakokban. A `FileName` tulajdonság pedig azt vezérli, hogy mi legyen a létrehozandó fájl alapértelmezett neve. Ez a név ebben a példában `MyShapes`.

Ezzel elkészült a rajzolóalkalmazás. Az aktuális grafikus adatokat most tetszőleges számú `*.shapes` fájlba menthetjük le vagy azokból tölthetjük be. Ha szeretnénk bővíteni ezt a Windows Forms-alkalmazást, használhatunk további alakzatokat, vagy tegyük lehetővé, hogy a felhasználók meghatározzák a rajzolt alakzatok méreteit, esetleg kiválaszthassák a mentett adatok formátumát (bináris, XML vagy SOAP).

Összefoglalás

A jelen fejezet célja az volt, hogy megvizsgáljuk a tradicionális asztali alkalmazások Windows Forms és a GDI+ API-k segítségével történő készítésének folyamatát, amelyek az 1.0 verzió óta a .NET keretrendszer részét képezik. A Windows Forms-alkalmazások minimum egy a `Formot` kiegészítő típussal és egy `Main()` metódussal rendelkeznek, ez utóbbi kommunikál az `Application` típussal.

Ha az űrlapot felhasználóifelület-elemekkel (pl. menürendszerek, GUI-beviteli vezérlők stb.) szeretnénk feltölteni, mindezt úgy tehetjük meg, hogy új objektumokat szúrunk be a származtatott `Controls` gyűjteménybe. Azt is bemutattuk, hogyan kezelhetjük az egér-, a billentyűzet- és a renderelési eseményeket. Továbbá megismerkedhettünk a `Graphics` típussal, valamint a grafikus adatok futásidejű generálásának számos módjával.

A Windows Forms API-t (valamilyen formában) felváltotta a .NET 3.0 bevezetésével megjelenő WPF API (ezt lásd a következő fejezetben). Amíg a WPF lesz a felturbózott grafikus felületek választott eszközrendszere, a Windows Forms API használata a standard üzleti alkalmazások, házilag készített alkalmazások és egyszerű konfigurációs segédprogramok összeállításának a legegyszerűbb (és sokszor a legközvetlenebb) módja. Így a Windows Forms még hosszú évekig a .NET-alaposztálykönyvtárak részét fogja képezni.

HUSZONNYOLCADIK FEJEZET

A WPF és az XAML

Az előző fejezetben megismerkedtünk a `System.Windows.Forms.dll` és a `System.Drawing.dll` szerelvényekben rendelkezésünkre álló funkcionalitással. Mint láttuk, a Windows Forms API a .NET platform eredeti GUI-eszközrendszere, amelynek típusait kifinomult felhasználói felületek készítése során alkalmazhatjuk. Miközben a .NET 3.5 teljes mértékben támogatja a Windows Forms/GDI+-technológiákat, a .NET 3.0 verziójától kezdve a Microsoft egy teljesen új asztali API-t, a Windows Presentation Foundationt (WPF) jelentetett meg.

Ez a fejezet bevezető a WPF használatához, az új felhasználói felület keretrendszer létrejöttének okát vizsgálja, és rövid áttekintést nyújt a különböző, API által támogatott WPF-alkalmazástípusokról. Ezt követően megismerkedünk a WPF programozási modell lényegével, és megvizsgáljuk az `Application` és a `Window` típusok szerepét, valamint az kulcsfontosságú WFP-szerelvényeket és -névtereket.

A fejezet további részében egy vadonatúj XML-alapú nyelvet tanulmányozunk: a XAML-t (Extensible Application Markup Language – bővíthető alkalmazás jelölőnyelv). Látjuk majd, hogy a XAML a WPF-fejlesztők számára lehetővé teszi a felhasználói felület definícióinak elkülönítését az őket vezérlő logikától. A fejezetben rendkívül fontos témaköröket vizsgálunk, például a csatolt tulajdonság szintaxist, a típusátalakítókat, a markup bővítőket, és megtanuljuk, hogyan elemezzünk futásidőben XAML-kódot. A fejezet végén különböző WPF-specifikus eszközökkel ismerkedünk meg, amelyeket a Visual Studio 2008 IDE foglal magában, és betekintést nyerünk a Microsoft Expression Blend szerepébe is.

A WPF mozgatórugója

Az évek alatt a Microsoft több grafikus felhasználói felület eszközrendszert készített (nyers C/C++/Win32 API fejlesztés, VB6, MFC stb.) asztali alkalmazások fejlesztéséhez. Az API-k a grafikus felülettel rendelkező alkalmazások alapelemeinek – például a főablakok, a párbeszédablakok, a vezérlőele-

mek, a menürendszerek és egyéb fontos összetevők - megjelenítéséhez biztosították a alapot. A .NET platform első verziójának megjelenésével a Windows Forms API (lásd a 27. fejezetet) hamar a felhasználói felület fejlesztés közkedvelt modellje lett, mivel egyszerű és rendkívül hatékony objektummodellel rendelkezik.

Bár sikeresen létrehoztak számos teljes értékű asztali alkalmazást a Windows Forms segítségével, igazság szerint ez a programozási modell elég *aszimmetrikus*. A `System.Windows.Forms.dll` és a `System.Drawing.dll` nem támogat közvetlenül több olyan technológiát, amelyek kész asztali alkalmazások készítéséhez szükségesek. Gondoljunk csak a WPF megjelenése előtti GUI-fejlesztés (például a .NET 2.0) ad hoc természetére (lásd a 28.1. táblázatot).

Elvárt működés	.NET 2.0 megoldás
Vezérlőelemekkel rendelkező űrlapok készítése	Windows Forms
2D grafika támogatása	GDI+ (`System.Drawing.dll`)
3D grafika támogatása	DirectX API-k
Videofolyam támogatása	Windows Media Player API-k
Folyamatstílusú dokumentumok támogatása	PDF-fájlok kezelése programozott módon

28.1. táblázat: *A .NET 2.0 megoldásai az elvárt működésre*

Mint látjuk, a Windows Forms fejlesztőknek több különböző API és objektummodell típusait kell alkalmazniuk. Noha a különböző API-k használatának módja szintaktikai szempontból hasonlónak tűnhet (végül is C#-kódról van szó), ahhoz azonban nem fér kétség, hogy a technológiák gyökeresen eltérő gondolkodásmódot igényelnek. A DirectX segítségével készült 3D animációk létrehozásához például teljesen más tudás szükséges, mint az adatkapcsolatok felépítéséhez. Annyi bizonyos, hogy egy Windows Forms programozó számára nem kis feladat eligazodni az API-k sokféleségén.

A különböző API-k egységesítése

A WPF célja az volt, hogy egységesített objektummodellben ötvözze ezeket a korábban nem kapcsolódó programozási feladatokat. Ha tehát 3D animációt kell készítenünk, nem kell kézzel programoznunk a DirectX API-t (de, ha

szeretnénk, megtehetjük), mivel a 3D funkcionalitást a WPF magában foglalja. A letisztult lehetőségeket a 28.2. táblázatban láthatjuk, amely bemutatja a .NET 3.0 megjelenésével rendelkezésünkre álló, asztali gépre történő fejlesztés modelljét.

Elvárt működés	Megoldás a .NET 3.0-ban és a későbbi változatokban
Vezérlőelemekkel rendelkező űrlapok készítése	WPF
2D grafika támogatása	WPF
3D grafika támogatása	WPF
Videofolyam támogatása	WPF
Folyamatstílusú dokumentumok támogatása	WPF

28.2. táblázat: *A .NET 3.0 megoldásai az elvárt működésre*

Kapcsolatok elkülönítése a XAML segítségével

A WPF egyik legvonzóbb előnye, hogy biztosítja egy Windows-alkalmazás megjelenésének és funkcionalitásának, valamint mögöttes programozási logikájának elkülönítését. A XAML nyelvben a *markup* segítségével definiálhatjuk az alkalmazás felhasználói felületét. A markup (amelyet ideális esetben művészi érzékkel megáldott fejlesztők az erre kifejlesztett eszközökkel hoznak létre) felügyelt programkódhoz kapcsolható, ezzel biztosíthatjuk a program funkcionalitását.

Megjegyzés A XAML nem csak a WPF-alkalmazásoknál használható! Bármely alkalmazás leírhatja a .NET-objektumok hierarchiáját a XAML segítségével, még akkor is, ha azoknak semmi közük a látható felhasználói felülethez. Például a XAML használatával egyedi tevékenységeket készíthetünk Windows Workflow Foundation alkalmazások számára.

Ha elmélyülünk a WPF rejtelmeiben, meglepetéssel tapasztalhatjuk, hogy az „asztali markup" milyen rugalmas lehetőségeket biztosít. A XAML segítségével nemcsak egyszerű felhasználói felület elemeket (gombokat, rácsokat, listamezőket stb.) definiálhatunk a markupban, hanem grafikus renderelést, animációkat, adatkötési logikát és multimédia-funkcionalitást (például vi-

deolejátszást) is. Egy kerek gomb – amely a vállalat logójának animációja – definiálása például csak néhány sornyi markupot igényel. A WPF-elemeket stílusokon és sablonokon keresztül megváltoztathatjuk, ami lehetővé teszi, hogy az alkalmazás központi feldolgozási forráskódjától függetlenül, a lehető legkevesebb erőfeszítéssel módosítsuk az alkalmazás külső megjelenését és funkcionalitását.

Ezeknek köszönhetően a WPF-rendszerekben az egyedi vezérlőelemek készítése szükségtelenné válik. A Windows Forms fejlesztéstől eltérően az egyedi WPF vezérlőelem-könyvtárak készítésének egyetlen elfogadható oka, ha a vezérlőelemek *viselkedését* kell megváltoztatnunk (egyedi metódusokat, tulajdonságokat vagy eseményeket kell hozzáadnunk; a virtuális tagok felüldefiniálásához létező vezérlőelemeket kell elhelyeznünk alosztályban; stb.). Ha pusztán a vezérlőelem (például a kerek gomb az animációval) *megjelenését és funkcionalitását* kell módosítanunk, ezt a markup segítségével megoldhatjuk.

Megjegyzés Egyedi WPF-vezérlőelemek készítésének másik oka lehet a bináris újrafelhasználás (a WPF vezérlőelem-könyvtáron keresztül), illetve olyan vezérlőelemek létrehozása, amelyek egyedi tervezési idejű funkcionalitást és a Visual Studio 2008 IDE környezettel integrációt biztosítanak.

Optimalizált renderelési modell biztosítása

Ne feledjük, hogy a WPF a Windows Vista operációs rendszerek által támogatott új videoillesztőprogram-modellt optimalizált módon alkalmazza. Noha a WPF-alkalmazásokat fejleszthetjük és telepíthetjük Windows XP operációs rendszerre (valamint Windows Server 2003-ra) is, ugyanaz az alkalmazás Vista operációs rendszeren jobb teljesítményt fog nyújtani, különösen animációs/multimédia-szolgáltatások esetében. Ez annak köszönhető, hogy a WPF megjelenítési szolgáltatásait a DirectX motor kezeli, amely hatékony hardveres és szoftveres renderelést biztosít.

Megjegyzés Ismét fontos kiemelni: a WPF nem korlátozódik a Windows Vista operációs rendszerre! Noha a Vista operációs rendszerben már a telepítőben rendelkezésünkre állnak a .NET 3.0 könyvtárak (a WPF is), a .NET Framework 3.5 SDK (fejlesztők számára) és a .NET 3.5 futtatórendszer (végfelhasználók számára) telepítését követően XP és Windows Server 2003 rendszerekben is készíthetünk és futtathatunk WPF-alkalmazásokat.

A WPF-alkalmazások fegyelmezettebb viselkedést tanúsítanak Windows Vista rendszereken. Ha egy grafikus elemekben gazdag alkalmazás összeomlik, akkor az alkalmazás nem rántja magával az egész operációs rendszert (a kék halál módján); ehelyett a rendetlenkedő alkalmazás egyszerűen leáll. Köztudott, a hírhedt kék halál oka gyakran az összeférhetetlen video-illesztőprogramokra vezethető vissza.

További WPF-központú hasznos tulajdonságok

A Windows Presentation Foundation egy olyan új API asztali alkalmazások készítésére, amely egyetlen objektummodellben egyesíti a különböző asztali API-kat, és a XAML segítségével világosan elkülöníti a kapcsolatokat. Ezen lényeges pontokon kívül a WPF-alkalmazások egyéb vonzó tulajdonságokkal is rendelkeznek, ezekkel a következő fejezetekben ismerkedhetünk meg. A következő felsorolás a kulcsfontosságú szolgáltatásokat foglalja össze röviden:

- Különböző elrendezéskezelők (jóval több mint a Windows Forms esetén), amelyek a tartalom el- és áthelyezéséhez rendkívül rugalmas vezérlést biztosítanak.
- Fejlett adatkötési motor alkalmazása, amellyel a tartalom különböző módon köthető a felhasználói felület elemeihez.
- Beépített stílusmotor, amely segítségével a WPF-alkalmazás számára „témák" definiálhatók.
- Vektorgrafika alkalmazása, amely segítségével az alkalmazást hosztoló képernyő méretének és felbontásának megfelelően a tartalom automatikusan újraméretezhető.
- 2D és 3D grafika, animációk, video- és audiolejátszás támogatása.
- Gazdag tipográfia API, például az XPS- (**X**ML **P**aper **S**pecification, dokumentumok, a rögzített dokumentumok (WYSIWYG), a folyamatos dokumentumok és a dokumentumjegyzetek (például a Sticky Notes API) támogatása.
- Együttműködés korábbi GUI-modellekkel (például Windows Forms, ActiveX, Win32 HWND-k).

A WPF-alkalmazások különböző típusai

A WPF API segítségével sokféle GUI-központú alkalmazás készíthető, amelyek alapvetően csak navigációs szerkezetükben és telepítési modelljükben térnek el egymástól. A következő részek rövid áttekintést nyújtanak az egyes alkalmazástípusokról.

Hagyományos asztali alkalmazások

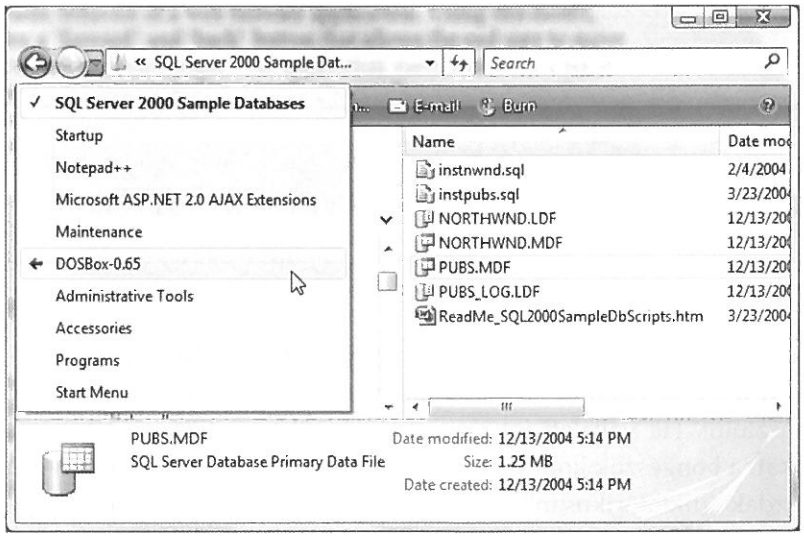
Az első (és talán legelterjedtebb) típus a hagyományos végrehajtható szerelvény, amely a helyi számítógépen fut. A WPF segítségével például szövegszerkesztőt, rajzoló programot vagy multimédiás programot – digitális zenelejátszót, fényképnézegetőt stb. – készíthetünk. A többi asztali alkalmazáshoz hasonlóan ezeket az `*.exe` fájlokat hagyományos eszközökkel (telepítőprogramokkal, Windows Installer csomagokkal stb.) telepíthetjük, vagy a ClickOnce technológia révén lehetővé tehetjük az asztali alkalmazások megosztását és telepítését távoli webkiszolgálón keresztül.

Mindezek ismeretében kijelenthetjük, hogy a WPF egy új API, amellyel hagyományos asztali alkalmazásokat készíthetünk. Programozási szempontból az ilyen WPF-alkalmazások a párbeszédablakok, az eszköztárak, az állapotsávok, a menürendszerek és egyéb felhasználóifelület-elemek megszokott készletén kívül (legalább) a Windows és az `Application` típusokat alkalmazzák.

Navigációalapú WPF-alkalmazások

A WPF-alkalmazások igény szerint működhetnek navigációalapú szerkezettel is, és így a hagyományos asztali alkalmazás egy webböngésző-alkalmazás alapvető funkcionalitásával rendelkezik. Ezen modell révén az elkészült asztali `*.exe` „előre" és „vissza" gombjaival a felhasználó előre és hátra lépkedhet a felhasználói felület különböző képernyői, más néven *oldalai* között. Az alkalmazás nyilvántartja az oldalak listáját, biztosítja az oldalak közötti navigációhoz szükséges infrastruktúrát és az oldalak közötti adatátvitelt (a webes alkalmazásokhoz hasonlóan), valamint karbantartja az előzménylistát. A 28.1. ábrán látható példa jól mutatja az imént említett funkcionalitást, a Vista Intézője épp ezt használja. Figyeljük meg a navigációs gombokat (és az előzménylistát) az ablak bal felső sarkában.

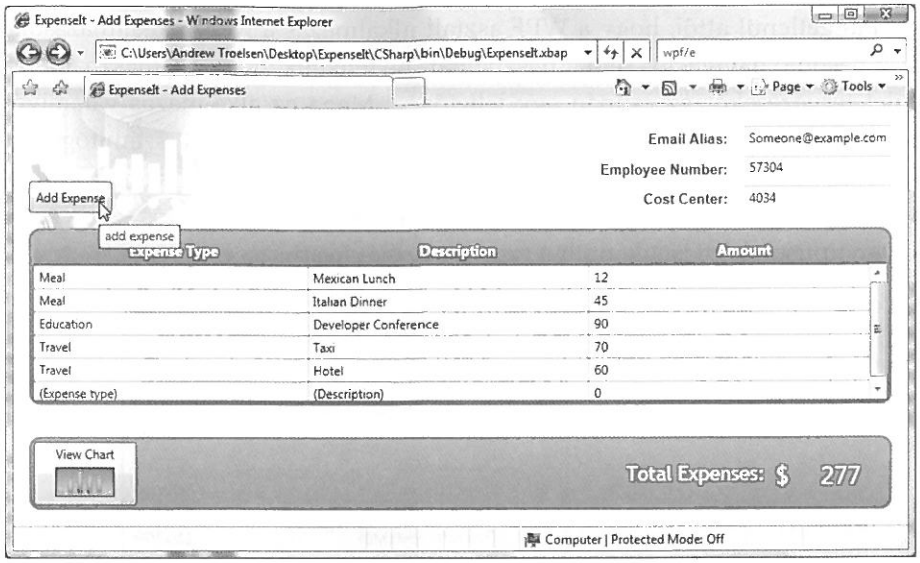
Függetlenül attól, hogy a WPF asztali alkalmazás a webes alkalmazásokhoz hasonló navigációs sémát használhat, ne feledjük, hogy a felhasználói felület tervezésekor ez csupán egy lehetőség. Maga az alkalmazás valamivel több egy helyi végrehajtható szerelvénynél, amely asztali számítógépen üzemel, és a hasonló megjelenésen és funkcionalitáson kívül semmi köze a webes alkalmazásokhoz. Programozási szempontból a navigációs szerkezetet olyan típusok képviselik, mint a `Page`, a `NavigationWindow` és a `Frame`.



28.1. ábra: *Navigációalapú asztali program*

XBAP-alkalmazások

A WPF lehetővé teszi olyan alkalmazások készítését, amelyeket *webböngészőben* hosztolhatunk. A WPF-alkalmazások ezen típusa az úgynevezett XAML-böngészőalkalmazás (XAML browser application), röviden XBAP. Ebben a modellben a felhasználó a meghatározott URL-re navigál, és ekkor az XBAP-alkalmazás (amely `*.xbap` fájlkiterjesztéssel rendelkezik) a háttérben letöltődik és települ a helyi számítógépre. A végrehajtható alkalmazások hagyományos ClickOnce telepítésétől eltérően az XBAP-programot közvetlenül a böngésző hosztolja, és az alkalmazás követi a böngésző belső navigációs rendszerét. A 28.2. ábrán egy XBAP-alkalmazást láthatunk működés közben (konkrétan az ExpensIt WPF példaprogramot, amelyet .NET Framework 3.5 SDK tartalmaz).



28.2. ábra: *Az XBAP-programok letöltődnek a helyi számítógépre, és a webböngésző hosztolja őket*

A WPF ezen típusának egyik lehetséges hátránya, hogy az XBAP-programokat Microsoft Internet Explorer 6.0 (és újabb verziói) vagy Firefox-böngészőben kell hosztolni. Ha vállalati intraneten keresztül telepítjük ezeket az alkalmazásokat, a böngészők kompatibilitása nem jelenthet problémát, mivel a rendszergazdák „diktatórikusan" meghatározhatják, hogy a felhasználók gépére milyen böngészőket kell telepíteni. Ha azonban a külvilág számára szeretnénk elérhetővé tenni az XBAP-t, nem várhatjuk el, hogy az összes felhasználó Internet Explorer/Firefox-böngészőt használjon, így néhány külső felhasználó nem fogja látni WPF XBAP alkalmazásunkat.

A másik probléma, amellyel tisztában kell lennünk, hogy az XBAP-alkalmazások biztonságos futtatókörnyezetben, úgynevezett *internetzónában* üzemelnek. A 20. fejezetből emlékezhetünk rá, hogy az ilyen virtuális környezetbe töltött .NET-szerelvények korlátozott módon férnek hozzá a rendszer erőforrásaihoz (például a helyi fájlrendszerhez vagy a rendszerleíró adatbázishoz), és nem alkalmazhatják szabadon bizonyos .NET API-k olyan tulajdonságait, amelyek biztonsági rést jelentenének. Az XBAP nem alkalmas a következő feladatok végrehajtására:

- Egyedülálló ablakok létrehozása és megjelenítése.
- Az alkalmazás által definiált párbeszédablakok megjelenítése.
- Az XBAP által indított Save párbeszédablak megjelenítése.

- Fájlrendszer-hozzáférés (elkülönített tárolóhely használata megengedett).
- Korábbi felhasználóifelület-modellek (Windows Forms, ActiveX) alkalmazása vagy nemfelügyelt kód hívása.

A másodlagos ablakok (vagy párbeszédablakok) létrehozásának hiánya első pillantásra komoly korlátozásnak tűnhet. Valójában, az XBAP a korábban említett oldalnavigációs modell segítségével több felhasználói felület megjelenítésére képes.

Silverlight-alkalmazások

A WPF és a XAML alapot biztosít a Silverlight *platformközi*, WPF-központú bővítmény számára. A Silverlight SDK segítségével böngészőalapú alkalmazásokat készíthetünk, amelyeket MAC OS X, valamint Microsoft Windows hosztolhat (a keretrendszer további operációs rendszereket is támogat).

A Silverlight segítségével rendkívül sokoldalú (és interaktív) webes alkalmazásokat építhetünk. A WPF-hez hasonlóan a Silverlight is rendelkezik vektoralapú grafikus rendszerrel, animáció támogatással, gazdag szöveges dokumentummodellel és multimédia-támogatással. Továbbá, a Silverlight 1.1 verziójától kezdve néhány .NET alaposztály-könyvtárat is felhasználhatunk alkalmazásainkban. Ezek a könyvtárak WPF-vezérlőelemeket, LINQ-támogatást, generikus gyűjtemény típusokat, webszolgáltatás-támogatást és az `mscorlib.dll` tekintélyes részét (fájl I/O, XML-műveletek stb.) foglalják magukban.

Megjegyzés A könyv jelen kiadása nem foglalkozik a Silverlight-technológiával. Az API-val kapcsolatban bővebb információkat a http://www.microsoft.com/silverlight webhelyen találunk. A webhelyről letölthetjük az ingyenes Silverlight SDK-t (magát a Silverlightot is), mintaprojekteket tanulmányozhatunk, és további ismereteket szerezhetünk a WPF-fejlesztés ezen izgalmas típusáról.

A WPF-szerelvények vizsgálata

A fejleszteni kívánt WPF-alkalmazás típusától függetlenül a WPF végeredményben alig több a .NET-szerelvényekbe csomagolt típusok gyűjteményénél. A 28.3. táblázat a WPF-alkalmazások készítéséhez használt kulcsfontosságú szerelvényeket ismerteti, amelyekre új projektek létrehozása során hivatkoznunk kell (reményeinknek megfelelően a Visual Studio 2008 WPF-projektjei automatikusan hivatkoznak a szükséges szerelvényekre).

Szerelvény	Jelentés
`PresentationCore.dll`	A szerelvény a WPF GUI rétegének alapját képező típusok nagy részét definiálja. Ez a szerelvény magában foglalja például a WPF Ink API (a Pocket PC-k és a Tablet PC-k stylusceruza bemenetének programozásához), több animációs primitív (a `System.Windows.Media.Animation` névtéren keresztül) és számos grafikus renderelési típus (a `System.Windows.Media` névtér segítségével) támogatását.
`PresentationFoundation.dll`	Ebben a szerelvényben WPF-vezérlőelem készletet, további animációs és multimédia-típusokat, adatkötéstámogatást és olyan típusokat találunk, amelyek lehetővé teszik a XAML és egyéb WPF-szolgáltatások programozott hozzáférését.
`WindowsBase.dll`	Ez a szerelvény definiálja azokat az alapvető (és sok esetben alacsonyszintű) típusokat, amelyek a WPF API infrastruktúráját képezik. Itt találjuk a WPF száltípusait, a biztonsági típusokat, a különféle típusátalakítókat, valamint más alapvető programozási primitíveket (`Point`, `Vector`, `Rect` stb.) képviselő típusokat.

28.3. táblázat: *Alapvető WPF-szerelvények*

Ez a három szerelvény együttesen számos új névteret és több száz új .NET-osztályt, -interfészt, -struktúrát, -felsorolást és -metódusreferenciát definiál. A .NET Framework 3.5 SDK dokumentációja a teljes referenciát magában foglalja, a 28.4. táblázat pedig a kulcsfontosságú névtereknek (de nem az összesnek) a szerepét ismerteti.

Névtér	Jelentés
`System.Windows`	Ez a WPF gyökérnévtere. A névtérben kulcsfontosságú típusokat találunk (például az `Application` és a `Window` típust), amelyekre a WPF asztali projekteknek szüksége lehet.
`System.Windows.Controls`	Itt találjuk az összes WPF-vezérlőelemet, például a menürendszerek készítéséhez szükséges típusokat, az eszköztippeket és több elrendezéskezelőt.

Névtér	Jelentés
`System.Windows.Markup`	A névtér több olyan típust definiál, amely lehetővé teszi a XAML-markup (és a megfelelő bináris formátum, a BAML) programozott elemzését és feldolgozását.
`System.Windows.Media`	Több médiaközpontú névtér gyökérnévterének neve. Ezekben a névterekben találjuk az animációkkal, a 3D rendereléssel, a szövegrendereléssel és egyéb multimédia-primitívekkel való munkához szükséges típusokat.
`System.Windows.Navigation`	A névtér a XAML-böngészőalkalmazások (XBAP), valamint a standard asztali alkalmazások – amelyek a navigációs oldalmodellel dolgoznak – által alkalmazott navigációs logikát biztosító típusokat foglalja magában.
`System.Windows.Shapes`	Ez a névtér többféle 2D grafikai típust (`Rectangle`, `Polygon` stb.) definiál, amelyeket a WPF-keretrendszer különböző összetevői használnak.

28.4. táblázat: *Alapvető WPF-névterek*

A fejezet további részében részletesen megismerkedünk a WPF programozási modelljével, és megvizsgáljuk a `System.Windows` névtér két tagját, amelyet minden hagyományos asztali fejlesztésben megtalálunk: az `Application` és a `Window` tagot.

Az Application osztály szerepe

A `System.Windows.Application` osztálytípus a futó WPF-alkalmazás globális példányát képviseli. A Windows Forms megfelelőjéhez hasonlóan ez a típus biztosítja a `Run()` metódust (az alkalmazás indításához), az események sorozatát, amelyek segítségével az alkalmazás élettartamát befolyásolhatjuk (például a `Startup` és az `Exit`), és a XAML-böngészőalkalmazásokra jellemző tagokat (például eseményeket, amelyek kiváltódnak, amikor a felhasználó az oldalak között navigál). A 28.5. táblázat a kulcsfontosságú tagok közül ismertet néhányat.

Tulajdonság	Jelentés
`Current`	Ez a statikus tulajdonság lehetővé teszi, hogy a kódunk bármely pontján hozzáférjünk a futtatott `Application` objektumhoz. A tulajdonság akkor hasznos, ha egy ablaknak vagy egy párbeszédablaknak hozzá kell férnie ahhoz az `Application` objektumhoz, amely létrehozta.
`MainWindow`	Ez a tulajdonság lehetővé teszi, hogy programozott módon lekérdezzük vagy beállítsuk az alkalmazás főablakát.
`Properties`	A tulajdonság segítségével lekérdezhetjük és beállíthatjuk az adatokat, amelyek a WPF-alkalmazás minden összetevője (ablakok, párbeszédablakok stb.) számára hozzáférhetők. Ez a tulajdonság több szempontból nagyon hasonlít az ASP.NET webes alkalmazásokban használt alkalmazásváltozókra.
`StartupUri`	Ez a tulajdonság beolvassa vagy beállítja az URI-t, amely meghatározza az alkalmazás indítása során automatikusan megjelenő ablakot vagy oldalt.
`Windows`	A tulajdonság a `WindowCollection` típust adja vissza, amely az `Application` objektumot létrehozó szálon belül készült összes ablakhoz hozzáférést biztosít. A tulajdonság akkor lehet rendkívül hasznos, ha az alkalmazás megnyitott ablakain kell végigiterálnunk és módosítanunk az egyes ablakok állapotát (például az összes ablakot kisméretűre állítjuk).

28.5. táblázat: *Az Application típus alapvető tulajdonságai*

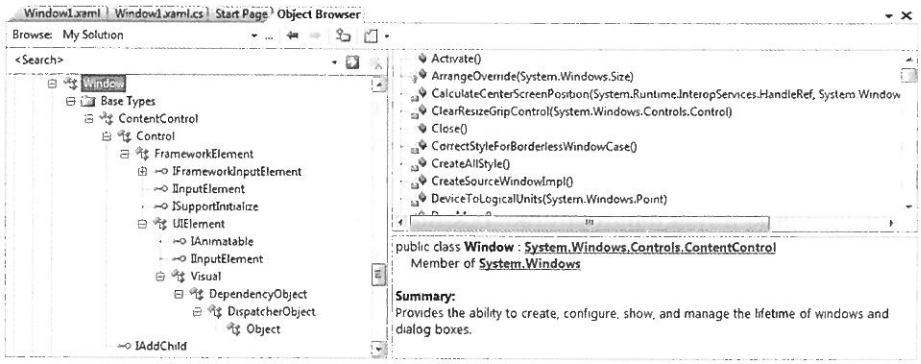
A Windows Forms megfelelőjétől eltérően a WPF `Application` típus nem kizárólag a statikus tagokon keresztül mutatja meg a funkcionalitását. A WPF-programok inkább egy osztályt definiálnak, amely kiterjeszti ezt a típust, és az belépési pontként szolgál a végrehajtható fájl számára, például a következőképpen:

```
// A WPF-program globális
// alkalmazásobjektumának definíciója.
class MyApp : Application
{
  [STAThread]
  static void Main()
  {
    // Események kezelése, az alkalmazás futtatása,
    // a főablak indítása stb.
  }
}
```

Az egyik következő példában egy teljes típust készítünk, amely az `Application` típusból származik. Addig is tanulmányozzuk a `Window` típus alapvető funkcionalitását, és közben ismerkedjünk meg néhány kulcsfontosságú WPF-alaposztállyal.

A Window osztály szerepe

A `System.Windows.Window` típus egyetlen ablakot képvisel – az ablak tulajdonosa az `Application` típusból származó típus –, amely a főablak által megjelenített párbeszédablakokat is magában foglalja. Mint az elvárható, a `Window` típus több szülőosztállyal rendelkezik, amelyek mindegyike további funkcionalitást biztosít. Tanulmányozzuk a 28.3. ábrát, amely a Visual Studio 2008 objektumböngészőjében a `System.Windows.Window` típus származtatási láncát (és a megvalósított interfészeket) mutatja be.



28.3. ábra: *A Window típus hierarchiája*

Az alaposztályok által biztosított funkcionalitást ez és a következő fejezet mutatja be, a következő részekben az egyes alaposztályok funkcionalitását elemezzük (erről a .NET Framework 3.5 SDK dokumentációjában bővebb információkat találunk).

A System.Windows.Controls.ContentControl alaposztály szerepe

A `Window` közvetlen szülője a `ContentControl`. Az alaposztály *tartalom* hosztolására alkalmas származtatott típusokat biztosít, ami tulajdonképpen a vezérlőelem felületén elhelyezett objektumok gyűjteményét jelenti. A WPF-tartalommodellben a tartalom vezérlőelem az egyszerű sztring-adatok mellett rengeteg felhasználóifelület-elemet tartalmazhat. Definiálhatunk például

olyan `Button` elemet, amely tartalomként beágyazott `ScrollBar` elemet foglal magában. A `ContentControl` alaposztály erre a célra (nem meglepően) a `Content` kulcsfontosságú tulajdonságot biztosítja.

Ha az értéket, amelyet a `Content` tulajdonsághoz szeretnénk rendelni, képviselheti egy egyszerű sztring-literál, akkor a `Content` tulajdonságot attribútumként, *explicit módon* beállíthatjuk az elem nyitó definíciójában:

```
<!-- A Content tulajdonság értékének beállítása explicit módon -->
<Button Height="80" Width="100" Content="ClickMe"/>
```

A `Content` tulajdonságot *implicit módon* is beállíthatjuk, ha a tartalom vezérlőelem definíciójának hatókörén belül meghatározunk egy értéket. Nézzük meg az előző gomb funkcionális szempontból egyenértékű XAML-leírását:

```
<!-- A Content tulajdonság értékének beállítása implicit módon -->
<Button Height="80" Width="100">
  ClickMe
</Button>
```

Ha azonban az érték, amelyet a `Content` tulajdonsághoz szeretnénk rendelni, nem írható le egyszerű karaktertömbbel, a vezérlőelem nyitó definíciójában nem rendelhetjük hozzá attribútum segítségével a `Content` tulajdonságot: ebben az esetben a tartalmat implicit módon vagy a *tulajdonság-elem szintaxis* segítségével kell meghatároznunk. Tekintsük meg a következő, funkcionális szempontból egyenértékű XAML-definíciót, amely `ScrollBar` típusra állítja a `Content` tulajdonságot (a fejezet későbbi részében elmélyedünk a XAML rejtelmeiben, most a részletekkel nem foglalkozunk):

```
<!-- Button elem, amely ScrollBar implicit tartalmat foglal
     magában -->
<Button Height = "80" Width = "100">
  <ScrollBar Width = "75" Height = "40"/>
</Button>

<!-- Button elem, amely tulajdonság-elem szintaxissal definiált
     ScrollBar elemet foglal magában -->
<Button Height = "80" Width = "100">
  <Button.Content>
    <ScrollBar Width = "75" Height = "40"/>
  </Button.Content>
</Button>
```

Ne feledjük, hogy nem minden WPF-vezérlőelem származik a `ContentControl` alaposztályból, ezért csupán a vezérlőelemek kis csoportja támogatja ezt az egyedi tartalommodellt. A `Frame`, a `GroupItem`, a `HeaderedContentControl`, a `Label`, a `ListBoxItem`, a `ButtonBase`, a `StatusBarItem`, a `ScrollViewer`, a `ToolTip`, a `UserControl` vagy a `Window` alaposztályból származó osztályok alkalmazhatják ezt a tartalommodellt. Bármely más típussal, amellyel erre törekszünk, a markupban fordítási idejű hibát kapunk. Például nézzük meg a következő, hibás `ScrollBar` típust:

```
<!-- Hiba! A Scrollbar nem a ContentControl alaposztályból
     származik! -->
<ScrollBar Height = "80" Width = "100">
  <Button Width = "75" Height = "40"/>
</ScrollBar >
```

Az új tartalommodellel kapcsolatos másik lényeges szempont, hogy a `ContentControl` alaposztályból származó vezérlőelemek (a Windows típust is beleértve) *egyetlen* értéket rendelhetnek a `Content` tulajdonsághoz. Ezért a következő XAML `Button` definíció szintén szabálytalan, mivel a `Content` tulajdonságot implicit módon kétszer állítja be:

```
<!-- TextBox és Ellipse elemet próbálunk a Button elemhez hozzáadni?
     Hiba! -->
<Button Height = "200" Width = "200">
  <Ellipse Fill = "Green" Height = "80" Width = "80"/>
  <TextBox Width = "50" Height = "40"/>
</Button >
```

Ez első pillantásra erősen korlátozónak tűnhet (képzeljük el, hogy egy párbeszédablak egyetlen gombbal milyen működésképtelen lenne!). A panelek segítségével szerencsére több elemet is hozzáadhatunk a `ContentControl` alaposztályból származó típusokhoz. Ehhez a tartalom minden egyes részét a WPF-paneltípusok egyikébe kell rendezni, és ezt követően a panel válik azon egyetlen értékké, amelyet a `Content` tulajdonsághoz rendelünk. A WPF-tartalommodellel, valamint a különböző paneltípusokkal (és a típusok vezérlőelemeivel) a 29. fejezetben ismerkedünk meg.

Megjegyzés A `System.Windows.ContentControl` osztály a `HasContent` tulajdonság segítségével állapítja meg, hogy a `Content` értéke pillanatnyilag null vagy sem.

A System.Windows.Controls.Control alaposztály szerepe

A `ContentControl` alaposztálytól eltérően a WPF-vezérlőelemek közös szülője a `Control` ősosztály. Ez az alaposztály több kulcsfontosságú tagot foglal magában, amelyek a felhasználói felület alapvető funkcionalitását biztosítják. A `Control` például olyan tulajdonságokat definiál, amelyekkel meghatározhatjuk a vezérlőelem méretét, átlátszóságát, tabulátorsorrendjének logikáját, a megjelenített kurzort, a háttér színét és egyéb jellemzőket. Továbbá ez a szülőosztály támogatja a *sablonozási szolgáltatásokat*. Mint a 30. fejezetben láthatjuk majd, a WPF-vezérlőelemek dinamikusan módosíthatják megjelenésüket sablonok, stílusok és témák segítségével. A 28.6. táblázat a `Control` típus néhány kulcsfontosságú tagját ismerteti, a tagokat a kapcsolódó működés szerint csoportosítja.

Tagok	Jelentés
`Background`, `Foreground`, `BorderBrush`, `BorderThickness`, `Padding`, `HorizontalContentAlignment`, `VerticalContentAlignment`	Ezek a tulajdonságok a vezérlőelem megjelenésével és elhelyezésével kapcsolatos alapbeállítások meghatározását teszik lehetővé.
`FontFamily`, `FontSize`, `FontStretch`, `FontWeight`	Ezek a tulajdonságok a betűtípussal kapcsolatos különböző beállításokat szabályozzák.
`IsTabStop`, `TabIndex`	Ezen tulajdonságok segítségével meghatározhatjuk az ablakban a vezérlőelemek tabulátorsorrendjét.
`MouseDoubleClick`, `PreviewMouseDoubleClick`	Ezek az események kezelik a vezérlőelemek dupla kattintásának műveletét.
`Template`	Ezen tulajdonság segítségével lekérdezhetjük és beállíthatjuk a vezérlőelem sablonját, amellyel módosíthatjuk a vezérlőelem renderelési kimenetét.

28.6. táblázat: *A Control típus kulcsfontosságú tagjai*

A System.Windows.FrameworkElement alaposztály szerepe

Az alaposztály több alacsonyszintű tagot biztosít, amelyeket a WPF-keretrendszer egészében alkalmazhatunk, például a forgatókönyv (storyboard) támogatását (animációkban), az adatkötés támogatását, a tagok elnevezését (a `Name` tulajdonság révén), a származtatott típus által meghatározott erőforrások megszerzését és a származtatott típus általános dimenzióinak meghatározását. A 28.7. táblázat a legfontosabb tagokat ismerteti.

Tagok	Jelentés
`ActualHeight, ActualWidth, MaxHeight, MaxWidth, MinHeight, MinWidth, Height, Width`	A származtatott típus méretét szabályozzák (nem meglepő módon).
`ContextMenu`	A származtatott típushoz tartozó előugró menüt lekérdezi vagy beállítja.
`Cursor`	A származtatott típushoz tartozó egérkurzort lekérdezi vagy beállítja.
`HorizontalAlignment, VerticalAlignment`	A típus elhelyezését szabályozzák a tárolóban (például a panelben vagy a listamezőben).
`Name`	Lehetővé teszi, hogy nevet rendeljünk a típushoz, és így a kódfájlban hozzáférjünk a típus funkcionalitásához.
`Resources`	Hozzáférést biztosít a típus által definiált erőforrásokhoz (a WPF-erőforrásrendszert részletesen a 30. fejezet ismerteti).
`ToolTip`	A származtatott típushoz tartozó eszköztippet lekérdezi vagy beállítja.

28.7. táblázat: *A FrameworkElement típus kulcsfontosságú tagjai*

A System.Windows.UIElement alaposztály szerepe

A `Window` származtatási láncában az össze típus közül az `UIElement` ősosztály biztosítja a leggazdagabb funkcionalitást. Az `UIElement` kulcsfontosságú feladata, hogy a származtatott típus számára olyan eseményeket biztosít, amelyekkel a típus lekérdezi az előteret és feldolgozza a bemeneti kéréseket. Ez az osztály például a húzd-és-dobd műveletek, az egérmozgatás, a billentyűzetbemenet és a stylusceruza-bemenet (a Pocket PC-k és a Tablet PC-k számára) kezeléséhez biztosít több eseményt.

A 29. fejezet részletesen bemutatja a WPF-eseménymodellt; azonban több alapvető esemény ismerősnek tűnhet (`MouseMove`, `KeyUp`, `MouseDown`, `MouseEnter`, `MouseLeave` stb.). A tucatnyi esemény definiálásán kívül ez a szülőosztály rengeteg tulajdonságot biztosít, amelyek a fókusz beállításáról, az engedélyezettségről, a láthatóságról és a találat tesztelési logikáról gondoskodnak. A tulajdonságokat a 28.8. táblázat ismerteti.

Tagok	Jelentés
`Focusable, IsFocused`	Ezek a tulajdonságok lehetővé teszik, hogy a származtatott típusra állítsuk a fókuszt.
`IsEnabled`	Ezzel a tulajdonsággal szabályozhatjuk, hogy a származtatott típus engedélyezett vagy letiltott.
`IsMouseDirectlyOver, IsMouseOver`	Ezek a tulajdonságok a találattesztelési logika végrehajtásának egyszerű módját biztosítják.
`IsVisible, Visibility, Visible`	Ezek a tulajdonságok lehetővé teszik, hogy a származtatott típus láthatóságának beállításaival dolgozzunk.
`RenderTransform`	A tulajdonság segítségével transzformációt készíthetünk, amelyet a rendszer a származtatott típus renderelése során alkalmaz.

28.8. táblázat: *Az UIElement típus kulcsfontosságú tagjai*

A System.Windows.Media.Visual szerepe

A Visual osztálytípus a WPF alapvető renderelési támogatását biztosítja. A támogatás magában foglalja a renderelt adatok találattesztelését, a koordinátatranszformációkat és a határolódoboz-számításokat. Valójában ez a típus jelenti a kapcsolódási pontot a felügyelt WPF-szerelvény verem és a nem felügyelt `milcore.dll` bináris között, amely a DirectX alrendszerrel kommunikál.

A 30. fejezetben láthatjuk majd, a WPF a grafikus adatok renderelésének háromféle módját biztosítja, amelyek funkcionalitás és teljesítmény tekintetében különböznek egymástól. A `Visual` típus (és gyermekei, például a `DrawingVisual`) alkalmazása jelenti a grafikus adatok renderelésének legkönnyebb módját, de ez a típus igényli a legtöbb manuális kódolást az összes szükséges szolgáltatás biztosításához. További részleteket erről a 30. fejezetben találunk.

A System.Windows.DependencyObject osztály szerepe

A WPF támogatja a .NET-tulajdonságok speciális típusát, az úgynevezett *függőségi tulajdonságokat*. Ez a megközelítés lehetővé teszi, hogy a típus más tulajdonságok értéke alapján kiszámítsa a tulajdonság értékét (innen a „függőség"). A típus akkor lehet része ennek az új tulajdonságsémának, ha a `DependencyObject` ősosztályból származik. Ezenkívül a `DependencyObject` lehetővé teszi a származtatott típusok számára, hogy *csatolt tulajdonságokat* támogassanak, amelyek a függőségi tulajdonságok rendkívül hasznos típusát jelentik a WPF adatkötési modell programozása, valamint a különböző WPF-paneltípusokban a felhasználói felület elemeinek elrendezése során.

A `DependencyObject` ősosztály két kulcsfontosságú metódust biztosít az összes származtatott típus számára: a `GetValue()` és a `SetValue()` metódust. A tagok segítségével létrehozhatjuk magát a tulajdonságot. Az infrastruktúra további elemei lehetővé teszik, hogy „regisztráljuk", ki használhatja a kérdéses függőségi/csatolt tulajdonságot. Noha a függőségi tulajdonságok a WPF-fejlesztés kulcsfontosságú szempontját jelentik, részleteik az esetek többségében rejtve maradnak. Az új tulajdonságtípus részleteiben a 29. fejezetben merülhetünk el.

A System.Windows.Threading.DispatcherObject szerepe

A `Window` típus utolsó ősosztálya (a `System.Object` osztályon túl, amely ezen a ponton nem szorul mélyebb magyarázatra) a `DispatcherObject` osztály. Ez a típus egy érdekes tulajdonságot, a `Dispatcher` tulajdonságot foglalja magában, amely a kapcsolódó `System.Windows.Threading.Dispatcher` objektumot adja vissza. A `Dispatcher` típus a WPF-alkalmazás eseménysorának belépési pontja, és a párhuzamosság, valamint a szálkezeléshez szükséges alapvető szerkezeteket biztosítja. Ezt az alacsonyszintű osztályt WPF-alkalmazásaink túlnyomó részében figyelmen kívül hagyhatjuk.

(XAML-mentes) WPF-alkalmazás készítése

A `Window` típus szülőosztályainak funkcionalitásával az alkalmazásunkban közvetlenül a `Window` típus létrehozásával vagy az erősen típusos leszármazott típus szülőjeként megadott `Window` osztály segítségével készíthetünk ablakokat. A következő kódpéldában vizsgáljuk meg mindkét megközelítést! Noha a legtöbb WPF-alkalmazás használja a XAML-t, a nyelv használata tetszőleges. Bármi, ami a XAML segítségével leírható, az kifejezhető kódban is (az esetek nagy részében), és fordítva. Igény szerint a mögöttes objektummodell és az imperatív C#-kód segítségével elkészíthetjük a teljes WPF-projektünket.

Ennek bemutatására készítsünk egyszerű, de teljes alkalmazást XAML használata *nélkül*, az `Application` és a `Window` típusok közvetlen használatával. Vizsgáljuk meg a következő C#-kódfájlt (`SimpleWPFApp.cs`), amely szerény, egy funkcionalitással rendelkező főablakot hoz létre:

```
// Egyszerű WPF-alkalmazás, XAML-kód felhasználása nélkül.
using System;
using System.Windows;
using System.Windows.Controls;

namespace SimpleWPFApp
{
  // Az első példában egyetlen osztálytípust definiálunk, amely
  // magát az alkalmazást és a főablakot képviseli.
  class MyWPFApp : Application
  {
    [STAThread]
    static void Main()
    {
      // A Startup és az Exit események kezelése, az alkalmazás
      // futtatása.
      MyWPFApp app = new MyWPFApp();
      app.Startup += AppStartUp;
      app.Exit += AppExit;
      app.Run(); // Fires the Startup event.
    }

    static void AppExit(object sender, ExitEventArgs e)
    {
      MessageBox.Show("App has exited");
    }

    static void AppStartUp(object sender, StartupEventArgs e)
    {
      // Window objektum létrehozása és néhány alapvető tulajdonság
      // beállítása.
      Window mainWindow = new Window();
      mainWindow.Title = "My First WPF App!";
      mainWindow.Height = 200;
      mainWindow.Width = 300;
      mainWindow.WindowStartupLocation =
                              WindowStartupLocation.CenterScreen;
      mainWindow.Show();
    }
  }
}
```

Megjegyzés A WPF-alkalmazás Main() metódusát a [STAThread] attribútummal kell minősíteni, amely biztosítja, hogy az alkalmazásunkban felhasznált korábbi COM-objektumok szálbiztosak legyenek. Ha a Main() metódust nem látjuk el ezzel a jelöléssel, az eredmény futásidejű kivétel lesz!

Vegyük észre, hogy a MyWPFApp osztály kibővíti a System.Windows.Application típust. A Main() metódusban létrehozzuk az alkalmazásobjektum egy példányát, és a metóduscsoportátalakítás-szintaxis segítségével kezeljük a Startup

és az Exit eseményeket. Emlékezzünk rá a 11. fejezetből, hogy a rövidítés alkalmazásával nem szükséges manuálisan meghatároznunk az adott esemény által használt mögöttes metódusreferenciákat.

Ha szeretnénk, közvetlenül a nevük segítségével is megadhatjuk a mögöttes metódusreferenciákat. A következő, módosított Main() metódusban a Startup esemény a StartupEventHandler metódusreferenciával együtt működik. A metódusreferencia csak olyan metódusra mutathat, amelynek első paramétere az Object, második paramétere pedig a StartupEventArgs. Másrészről, az Exit esemény együttműködik az ExitEventHandler metódusreferenciával, amely megköveteli, hogy a hivatkozott metódus második paramétere az ExitEventArgs legyen:

```
[STAThread]
static void Main()
{
  // Ezúttal meghatározzuk a mögöttes metódusreferenciákat.
  MyWPFApp app = new MyWPFApp();
  app.Startup += new StartupEventHandler(AppStartUp);
  app.Exit += new ExitEventHandler(AppExit);
  app.Run();  // Kiváltja a Startup eseményt.
}
```

Az AppStartUp() metódus létrehozza a Window típust, beállít néhány alapvető tulajdonságot, és a Show() metódus hívásával nem modálisan jeleníti meg a képernyőn az ablakot (a Windows Formshoz hasonlóan a ShowDialog() metódus segítségével indítható modális dialógus). Az AppExit() metódus a WPF MessageBox típus segítségével diagnosztikai üzenetet jelenít meg, ha az alkalmazást leállítjuk.

A C#-kód végrehajtható WPF-alkalmazásra fordítása során tételezzük fel, hogy létrehoztuk a build.rsp C#-reakciófájlt, amely hivatkozik az összes WPF-szerelvényre. Vegyük észre, hogy az egyes szerelvények elérési útvonalát egy sorban kell meghatároznunk (a reakciófájlokkal és a parancssoros fordító használatával kapcsolatos bővebb információkat lásd a 2. fejezetben):

```
# build.rsp
#
/target:winexe
/out:SimpleWPFApp.exe
/r:"C:\Program Files\Reference
Assemblies\Microsoft\Framework\v3.0\WindowsBase.dll"
/r:"C:\Program Files\Reference Assemblies\Microsoft\Framework
      \v3.0\PresentationCore.dll"
/r:"C:\Program Files\Reference Assemblies\Microsoft\Framework
      \v3.0\PresentationFramework.dll"
*.cs
```

A WPF-programot a parancssorban a következőképpen fordíthatjuk le:

```
csc @build.rsp
```

A program futtatása során egyszerű főablak jelenik meg, amelyet kisméretűre vagy teljes méretűre állíthatunk, és bezárhatunk. Egy kicsit „megfűszerezhetjük" az alkalmazást, és hozzáadhatunk néhány interfészelemet. De mindezek előtt szervezzük át a kódunkat, és gondoskodjunk egy erősen típusos és jól egységbezárt `Window`-leszármazott osztályról.

A Window osztálytípus kibővítése

Pillanatnyilag az `Application` osztályból származó osztályunk az alkalmazás indítása során közvetlenül létrehozza a `Window` típus egy példányát. Ideális esetben létrehozunk egy osztályt, amely a `Window` osztályból származik, és egységbe zárjuk ezen osztály funkcionalitását. Tételezzük fel, hogy az aktuális `SimpleWPFApp` névterünkben létrehoztuk a következő osztálydefiníciót:

```
class MainWindow : Window
{
  public MainWindow(string windowTitle, int height, int width)
  {
    this.Title = windowTitle;
    this.WindowStartupLocation = WindowStartupLocation.CenterScreen;
    this.Height = height;
    this.Width = width;
    this.Show();
  }
}
```

Most már módosíthatjuk a `Startup` eseménykezelőt, hogy az közvetlenül létrehozza a `MainWindow` egy példányát:

```
static void AppStartUp(object sender, StartupEventArgs e)
{
  // MainWindow objektum létrehozása.
  MainWindow wnd = new MainWindow("My better WPF App!", 200, 300);
}
```

A program ismételt fordítása és futtatása során láthatjuk, hogy a kimenet megegyezik. A megoldás nyilvánvaló előnye, hogy a további fejlesztéshez egy erősen típusos osztály áll rendelkezésünkre.

Megjegyzés Ha `Window` (vagy a `Window` osztályból származó) objektumot hozunk létre, a rendszer (a `Window` osztályban található konstruktorlogika segítségével) automatikusan hozzáadja az `Application` típus belső ablak gyűjteményéhez. Ennek köszönhetően az ablak egészen a leállításig a memóriában marad, illetve addig, amíg a gyűjteményből explicit módon el nem távolítjuk az `Application.Windows` tulajdonsággal.

Egyszerű felhasználói felület létrehozása

A `Window`-leszármazott típusban a felhasználóifelület-elemek hozzáadása hasonlít (de nem egyezik meg teljesen) a `System.Windows.Forms.Form` osztályból származó típusokban a felhasználóifelület-elemek hozzáadásához:

1. Definiáljuk a szükséges vezérlőt képviselő tagváltozót!
2. A `Window` típus létrehozása során állítsuk be a változó megjelenését és funkcionalitását!
3. Az `AddChild()` metódus hívásával adjuk a `Window` ügyfélterületéhez a vezérlőt!

Noha a folyamat hasonlít a Windows Forms fejlesztéshez, azonban van egy nyilvánvaló különbség: a WPF által alkalmazott felhasználóifelület-vezérlőelemeket nem a `System.Windows.Forms`, hanem a `System.Windows.Controls` névtér definiálja (szerencsére a nevük és funkcionalitásuk sok esetben hasonlít Windows Forms megfelelőjükhöz).

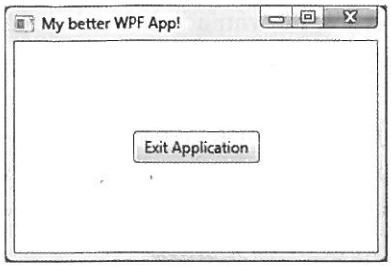
Szembetűnőbb eltérés a Windows Formstól, hogy a `Window` osztályból származó típus kizárólag *egyetlen gyermekelemet* foglalhat magában (a WPF-tartalommodellnek köszönhetően). Ha az ablakban több felhasználóifelület-elemet szeretnénk használni (és ez szinte minden ablakra igaz), akkor elrendezéskezelő, például a `DockPanel`, a `Grid`, a `Canvas` vagy a `StackPanel` segítségével szabályozhatjuk az elemek elhelyezését.

Ebben a példában egyetlen `Button` típust adunk a `Window` osztályból származó típushoz. Ha erre a gombra kattintunk, hozzáférünk a globális alkalmazásobjektumhoz (az `Application.Current` tulajdonság révén), meghívjuk a `Shutdown()` metódust, és leállítjuk az alkalmazást. Gondoljuk végig a `MainWindow` osztály következő módosítását:

```
class MainWindow : Window
{
  // A felhasználóifelület-elemünk.
  private Button btnExitApp = new Button();

  public MainWindow(string windowTitle, int height, int width)
  {
    // Gomb konfigurálása, és a gyermek-vezérlőelem beállítása.
    btnExitApp.Click += new RoutedEventHandler(btnExitApp_Clicked);
    btnExitApp.Content = "Exit Application";
    btnExitApp.Height = 25;
    btnExitApp.Width = 100;

    // Az ablak tartalmának beállítása (egyetlen gomb).
    this.AddChild(btnExitApp);

    // Az ablak beállítása.
    this.Title = windowTitle;
    this.WindowStartupLocation = WindowStartupLocation.CenterScreen;
    this.Height = height;
    this.Width = width;
    this.Show();
  }

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

Mivel a 27. fejezetben már dolgoztunk a Windows Formsszal, az ablak konstruktorának kódja nem lehet ijesztő. Vegyük észre, hogy a WPF gomb kattintási eseménye a `RoutedEventHandler` metódusreferenciával együtt működik, és ez azonnal felveti a kérdést: mik azok az irányított események? A WPF-eseménymodell részleteit a következő fejezetben vizsgáljuk meg; pillanatnyilag elég megértenünk, hogy a `RoutedEventHandler` metódusreferencia célpontjainak első paraméterként objektumot, második paraméterként pedig a `RoutedEventArgs` típust kell meghatározniuk.

Ha ismét lefordítjuk és futtatjuk az alkalmazást, a 28.4. ábrán bemutatott testre szabott ablakot láthatjuk. Vegyük észre, hogy a gombunk az ablak ügyfélterületének közepén jelenik meg. Ha a tartalmat nem helyezzük el WPF-paneltípusban, ez az alapértelmezett viselkedés.



28.4. ábra: *Érdekes WPF-alkalmazás*

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

Az Application típus további jellemzői

Miután a tisztán kódalapú megközelítéssel létrehoztunk egy egyszerű WPF-programot, vizsgáljuk meg az `Application` típus további jellemzőit, és kezdjük a teljes alkalmazásra vonatkozó adatok felépítésével. Ehhez az előző `SimpleWPFApp` alkalmazást újabb funkcionalitással fogjuk gazdagítani.

Az alkalmazás adatai és a parancssori argumentumok feldolgozása

Emlékezzünk rá, hogy az `Application` típus meghatározza a `Properties` tulajdonságot, amely lehetővé teszi, hogy a típusindexelővel kulcs/érték párok gyűjteményét definiáljuk. Mivel az indexelő definíciója szerint `System.Object` típuson működik, a gyűjteményben bármilyen elemet tárolhatunk (az egyedi osztályainkat is beleértve), és az elemeket később a „barátságos" moniker segítségével visszaolvashatjuk. A megközelítés segítségével a WPF-alkalmazás ablakai között könnyen lehet adatokat megosztani.

Ennek bemutatására módosítsuk az aktuális `Startup` eseménykezelőt, hogy ellenőrizze a /GODMODE értéket a bejövő parancssori paraméterekben (ez a PC-s videojátékok gyakori csaláskódja). Ha ilyen tokent találunk, a tulajdonsággyűjteményben megegyező névvel rendelkező logikai értéket állítsunk igaz értékre (egyébként az értéket állítsuk hamisra).

Egyszerűnek hangzik, de felmerülhet a kérdés, hogyan adjuk át a bejövő parancssori paramétereket (amelyek tipikusan a `Main()` metódustól kell megszerezni) a `Startup` eseménykezelőnknek? Az egyik lehetőség, hogy meghívjuk a statikus `Environment.GetCommandLineArgs()` metódust. Azonban ugyanezeket az argumentumokat a rendszer automatikusan hozzáadja a bejövő `StartupEventArgs` paraméterhez, és az `Args` tulajdonság segítségével hozzájuk férhetünk. Mindezek után lássuk az első módosítást:

```
static void AppStartUp(object sender, StartupEventArgs e)
{
  // A bejövő parancssori paramétereknél
  // a /GODMODE kapcsoló ellenőrzése.
  Application.Current.Properties["GodMode"] = false;

  foreach(string arg in e.Args)
  {
    if (arg.ToLower() == "/godmode")
    {
      Application.Current.Properties["GodMode"] = true;
      break;
    }
  }

  // MainWindow objektum létrehozása.
  MainWindow wnd = new MainWindow("My better WPF App!", 200, 300);
}
```

Emlékezzünk rá, hogy ehhez az új név/érték párhoz a WPF-alkalmazáson belül bárhonnan hozzáférünk. Csupán annyit kell tennünk, hogy megszerezzük a globális alkalmazásobjektum hozzáférési pontját (az `Application.Current` segítségével), és megvizsgáljuk a gyűjteményt. A főablak gombjának kattintási eseménykezelőjét például a következőképpen módosíthatjuk:

```
private void btnExitApp_Clicked(object sender, RoutedEventArgs e)
{
  // A felhasználó engedélyezte a /godmode kapcsolót?
  if((bool)Application.Current.Properties["GodMode"])
    MessageBox.Show("Cheater!");

  // Az aktuális alkalmazás kezelőjének megszerzése,
  // és az alkalmazás leállítása.
  Application.Current.Shutdown();
}
```

Ha a felhasználó elindítja a programot a

```
SimpleWPFApp.exe /godmode
```

paranccsal, akkor az alkalmazás leállításakor megjelenik a képernyőn az üzenetünket tartalmazó üzenetdoboz.

Az Application típus Windows gyűjteményének feldolgozása

Az `Application` típus másik érdekes tulajdonsága a `Windows`, amely hozzáférést biztosít az aktuális WPF-alkalmazás memóriába töltött ablakait képviselő gyűjteményhez. Emlékezzünk rá, hogy amikor új `Window` típusokat hozunk létre, a rendszer automatikusan hozzáadja a típusokat a globális alkalmazásobjektum `Windows` gyűjteményéhez. Az aktuális példánkat ennek bemutatására nem kell módosítanunk; lássunk azonban egy példametódust, amely kis méretűre állítja az alkalmazás összes ablakát (válaszul a felhasználó által kezdeményezett meghatározott billentyűzetműveletre vagy menüopcióra):

```
static void MinimizeAllWindows()
{
  foreach (Window wnd in Application.Current.Windows)
  {
    wnd.WindowState = WindowState.Minimized;
  }
}
```

Az Application típus további eseményei

A .NET alaposztály-könyvtár több tagjához hasonlóan az `Application` is definiál egy eseménykészletet, amelynek tagjait elkaphatjuk. A `Startup` és az `Event` eseményeket már láttuk működés közben. Fontos ismernünk az `Activated` és a `Deactivated` eseményeket is. Ezek az események első pillantásra megtévesztőnek tűnhetnek, mivel a `Window` típus megegyező nevű metódusokat biztosít. A felhasználói felület megfelelőitől eltérően az `Activated` és a `Deactivated` események akkor sülnek el, ha az alkalmazásobjektum által karbantartott *bármely* ablak megszerzi vagy elveszíti a fókuszt (ellentétben a `Window` típus ugyanezen elemeivel, amelyek az adott `Window` objektum alapján egyediek).

Az aktuális példánkban nem szükséges a két eseményt kezelnünk, de, ha muszáj, ne feledjük, hogy mindkét esemény a `System.EventHandler` metódusreferenciával együtt működik, ezért az eseménykezelő első paramétere egy `Object`, második paramétere egy `System.EventArgs` (az `EventHandler` metódusreferenciával kapcsolatos ismereteinket a 27. fejezetben frissíthetjük fel).

Megjegyzés Az `Application` típus további eseményeinek többsége a navigációalapú WPF-alkalmazásra jellemző. Az események segítségével programunkban nyomon követhetjük a `Page` objektumok közötti mozgás folyamatát.

A Window típus további jellemzői

Mint a fejezet korábbi részében láttuk, a `Window` típus szülőosztályai és a megvalósított interfészek révén gazdag funkcionalitással rendelkezik. A következő fejezetekben egyre több információt kapunk az alaposztályok funkcionalitásáról; fontos újra megvizsgálnunk magát a `Window` típust, és meg kell ismerkednünk néhány alapvető szolgáltatással, amelyeket mindennapi munkánk során alkalmaznunk kell. Kezdjük azzal az eseménykészlettel, amelyet a rendszer az objektum élettartama során kivált.

A Window objektum élettartama

A `System.Windows.Forms.Form` típushoz hasonlóan a `System.Windows.Window` is rendelkezik eseménykészlettel. Az eseményeket a rendszer az objektum élettartama során süti el. Ha ezen események egyikét (vagy mindegyiket) kezeljük, kényelmes módszer áll rendelkezésünkre az egyedi logika végrehajtásához, miközben a `Window` objektum végzi a saját dolgát. Mivel az ablak tulajdonképpen osztálytípus, az inicializálás legelső lépése egy meghatározott konstruktor hívását vonja maga után. Ezt követően az első WPF-központú esemény – mely kiváltódik – a `SourceInitialized`, amely akkor bizonyul hasznosnak, ha a `Window` objektumunk különböző együttműködési szolgáltatásokat alkalmaz (például korábbi ActiveX vezérlőelemeket használ WPF-alkalmazásban). Még ebben az is csak ritkán kell kezelnünk az eseményt (erről bővebb információkat a .NET Framework 3.5 dokumentációban találunk).

Az első közvetlenül használható esemény, amelyet a `Window` konstruktora után elsül: az `Activate`. Az esemény a `System.EventHandler` metódusreferenciával együtt működik. Az eseményt a rendszer akkor váltja ki, ha az ablak megszerzi a fókuszt, és így az előtérbeli ablakká lép elő. Ezen esemény párja a `Deactivate` (szintén a `System.EventHandler` metódusreferenciával működik együtt), amely akkor sül el, amikor az ablak elveszíti a fókuszt.

Vizsgáljuk meg a `Window` osztályból származó típusunk módosítását, amely tájékoztató üzenetet ad egy privát sztringváltozóhoz (a sztringváltozó hasznát rövidesen megtudjuk), amikor az `Activate` és a `Deactivate` események bekövetkeznek:

```
class MainWindow : Window
{
  private Button btnExitApp = new Button();

  // Ez a sztring rögzíti, hogy mely események
  // mely időpontban váltódnak ki.
  private string lifeTimeData = String.Empty;

  protected void MainWindow_Activated(object sender, EventArgs e)
  {
    lifeTimeData += "Activate Event Fired!\n";
  }

  protected void MainWindow_Deactivated(object sender, EventArgs e)
  {
    lifeTimeData += "Deactivated Event Fired!\n";
  }

  public MainWindow(string windowTitle, int height, int width)
  {
    // Feliratkozás az eseményekre.
    this.Activated += MainWindow_Activated;
    this.Deactivated += MainWindow_Deactivated;
    ...
  }
}
```

Megjegyzés Emlékezzünk rá, hogy az `Application.Activated` és az `Application.Deactivated` eseményekkel az összes ablak alkalmazásszintű aktiválását és deaktiválását kezelhetjük.

Ha az `Activated` esemény elsül, a rendszer kiváltja a `Loaded` eseményt (az esemény a `RoutedEventHandler` metódusreferenciával dolgozik), amely jelzi, hogy az ablak elrendezése és renderelése teljes, és készen áll a felhasználói bemenet kezelésére.

Megjegyzés Míg a fókusz megszerzésével és elvesztésével a rendszer többször kiválthatja az `Activated` eseményt, a `Loaded` esemény csak egyszer váltódik ki az ablak élete során.

Tételezzük fel, hogy a `MainWindow` típus kezeli ezt az eseményt, és a következő eseménykezelőt definiálja:

```
protected void MainWindow_Loaded(object sender, RoutedEventArgs e)
{
  lifeTimeData += "Loaded Event Fired!\n";
}
```

Megjegyzés Ha szükség lenne rá (ez az egyedi WPF-vezérlőelemek esetén előfordulhat), elkaphatjuk a pillanatot, amikor az ablak tartalmát a `ContentRendered` esemény kezelője betölti.

A Window objektum Closing eseményének kezelése

A felhasználók több beépített rendszerszintű módszer segítségével leállíthatnak egy ablakot, (például, ha az ablakkereten az „X" bezárás gombra kattintanak), illetve közvetetten valamilyen felhasználói beavatkozásra (például a File > Exit műveletre) adott válaszként a `Close()` metódus meghívásával. A WPF két eseményt biztosít, amelyek elkapásával eldönthetjük, hogy a felhasználó valóban kész az ablak leállítására és az ablak memóriából való eltávolítására. Az elsőként kiváltott esemény a `Closing`, amely a `CancelEventHandler` metódusreferenciával együtt dolgozik.

A metódusreferencia megköveteli, hogy a célmetódusok második paramétere a `System.ComponentModel.CancelEventArgs` legyen. A `CancelEventArgs` a `Cancel` tulajdonsággal rendelkezik, amely igaz értéke megakadályozza, hogy a rendszer valóban bezárja az ablakot (ez akkor hasznos, amikor megkérdezzük a felhasználót, hogy tényleg szeretné-e bezárni az ablakot, vagy előtte még szeretné menteni a munkáját).

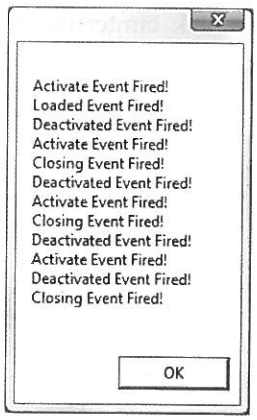
Ha a felhasználó valóban szeretné bezárni az ablakot, a `CancelEventArgs.Cancel` tulajdonság értéke hamisra[1] állítható. Ekkor elsül a `Closed` esemény (amely a `System.EventHandler` metódusreferenciával dolgozik), és ebben a pillanatban az ablak készen áll arra, hogy a rendszer végérvényesen bezárja.

[1] Az eredeti szövegbe itt az *igaz* érték szerepel (CancelEventArgs.Cancel can be set to *true*), ezt azonban javítottuk, lévén nyilvánvaló tévedés, ami az alábbi kódrészlettel támasztható alá:
// Ha a felhasználó nem akarja bezárni az ablakot, a zárási kérelmet töröljük.
e.Cancel = true – a kiadó.

Tételezzük fel, hogy a `MainWindow` típus kezeli ezt a két eseményt, és vizsgáljuk meg az utolsó eseménykezelőket:

```
protected void MainWindow_Closing(object sender,
                                  System.ComponentModel.CancelEventArgs e)
{
  lifeTimeData += "Closing Event Fired!\n";

  // Ellenőrizzük, hogy a felhasználó tényleg szeretné bezárni
  // az ablakot.
  string msg = "Do you want to close without saving?";
  MessageBoxResult result = MessageBox.Show(msg,
        "My App", MessageBoxButton.YesNo, MessageBoxImage.Warning);
  if (result == MessageBoxResult.No)
  {
    // Ha a felhasználó nem akarja bezárni az ablakot, a zárási
    // kérelmet töröljük.
    e.Cancel = true;
  }
}

protected void MainWindow_Closed(object sender, EventArgs e)
{
  lifeTimeData += "Closing Event Fired!\n";
  MessageBox.Show(lifeTimeData);
}
```

Ha lefordítjuk és futtatjuk az alkalmazást, az ablakot néhányszor helyezzük fókuszba, majd vegyük el a fókuszt. Próbáljuk meg egyszer vagy kétszer bezárni az ablakot. Ha valóban végleg bezárjuk az ablakot, a képernyőn megjelenő üzenetdobozban találjuk azon események listáját, amelyek az ablak élete során működésbe léptek (a 28.5. ábrán egy lehetséges tesztfutás eredménye látható).



28.5. ábra: *A System.Windows.Window élete*

Ablakszintű egéreseményekkezelése

A Windows Formshoz hasonlóan a WPF API több eseményt biztosít, amelyeket elkaphatunk, és így együttműködhetünk az egérrel. Az `UIElement` alaposztály több egérközpontú eseményt definiál; például a `MouseMove`, a `MouseUp`, a `MouseDown`, a `MouseEnter` és a `MouseLeave` eseményeket.

Tanulmányozzuk például a `MouseMove` esemény kezelését. Ez az esemény a `System.Windows.Input.MouseEventHandler` metódusreferenciával együtt működik, amely megköveteli, hogy a célmetódus második paramétere egy `System.Windows.Input.MouseEventArgs` típus legyen. A `MouseEventArgs` segítségével (a Windows Forms alkalmazáshoz hasonlóan) megszerezhetjük az egér *(x,y)* pozícióját és más kapcsolódó részletet. Vizsgáljuk meg a következő definíciórészletet:

```
public class MouseEventArgs : InputEventArgs
{
  ...
  public Point GetPosition(IInputElement relativeTo);
  public MouseButtonState LeftButton { get; }
  public MouseButtonState MiddleButton { get; }
  public MouseDevice MouseDevice { get; }
  public MouseButtonState RightButton { get; }
  public StylusDevice StylusDevice { get; }
  public MouseButtonState XButton1 { get; }
  public MouseButtonState XButton2 { get; }
}
```

A `GetPosition()` metódus segítségével kinyerhetjük az ablakban a felhasználóifelület-eleméhez viszonyított *(x,y)* értéket. Ha az aktivált ablakhoz viszonyított pozíciót szeretnénk rögzíteni, csak ezt az értéket kell átadnunk. A következő példa a `MouseMove` eseménykezelője, amely az ablak címterületén megjeleníti az egér pozícióját (vegyük észre, hogy a visszakapott `Point` típust a `ToString()` metódussal sztring értékké alakítjuk):

```
protected void MainWindow_MouseMove(object sender,
                    System.Windows.Input.MouseEventArgs e)
{
  // Az ablak címének beállítása az egér aktuális x,y pozíciójára.
  this.Title = e.GetPosition(this).ToString();
}
```

Ablakszintű billentyűzetesemények kezelése

A billentyűzetbemenet feldolgozása nagyon hasonló az előbbiekhez. Az `UIElement` több eseményt definiál, amelyeket elkaphatunk, és az aktív elemen elfoghatjuk a billentyűzetről érkező gombnyomásokat (például `KeyUp`, `KeyDown` stb.). A `KeyUp` és a `KeyDown` események a `System.Windows.Input.KeyEventHandler` metódusreferenciával dolgoznak, amely megköveteli, hogy a célmetódus második paramétere `KeyEventArgs` típusú legyen. A típus fontos nyilvános tulajdonságokat definiál:

```
public class KeyEventArgs : KeyboardEventArgs
{
  ...
  public bool IsDown { get; }
  public bool IsRepeat { get; }
  public bool IsToggled { get; }
  public bool IsUp { get; }
  public Key Key { get; }
  public KeyStates KeyStates { get; }
  public Key SystemKey { get; }
}
```

A `KeyUp` esemény kezelésének bemutatására a következő eseménykezelő az ablak címsorában megjeleníti az előzőleg megnyomott gombot:

```
protected void MainWindow_KeyUp(object sender,
                    System.Windows.Input.KeyEventArgs e)
{
  // Gombnyomás megjelenítése.
  this.Title = e.Key.ToString();
}
```

A fejezet ezen pontján a WPF pusztán egy új grafikusfelület-modellnek tűnhet, amely (nagyjából) ugyanazokat a szolgáltatásokat biztosítja, mint a `System.Windows.Forms.dll` szerelvény. Ha valóban ez lenne a helyzet, akkor joggal megkérdőjelezhetnénk egy újabb felhasználóifelület-eszközrendszer létjogosultságát. Hogy lássuk, mi teszi a WPF-et egyedivé, ismerkedjünk meg az új XML-alapú nyelvvel, a XAML-lel.

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

(XAML-központú) WPF-alkalmazás készítése

A XAML (**E**xtensible **A**pplication **M**arkup **L**anguage - bővíthető alkalmazás jelölőnyelv) egy XML-alapú nyelv, amely lehetővé teszi, hogy markupban definiáljuk a .NET-objektumok hierarchiájának állapotát (és bizonyos mértékben a funkcionalitását is). Noha a XAML-t gyakran használjuk a WPF-ben felhasználói felületek készítésére, a nyelv segítségével a *nem absztrakt* .NET-típusok bármely hierarchiáját leírhatjuk (az egyedi .NET-szerelvényünkben definiált egyedi típusokat is), ha a típusok mindegyike támogat egy alapértelmezett konstruktort. Mint látjuk, a `*.xaml` fájlban található markup átalakul olyan teljes objektummodellé, amely közvetlenül leképezi az objektumokat a kapcsolódó .NET-névtér típusaihoz.

Mivel a XAML egy XML-alapú nyelv, az XML által biztosított előnyök és hátrányok egyaránt megtalálhatók benne. Hasznos tulajdonság, hogy a XAML-fájlok önleíróak (mint az XML-dokumentumok általában). A XAML-fájlban nagyjából minden elem az adott .NET-névtérben egy típus nevét képviseli (például a `Button`, a `Window` vagy az `Application` típust). A nyitó elem hatókörében az attribútumok a meghatározott típus tulajdonságaira (`Height`, `Width` stb.) vagy eseményeire (`Startup`, `Click` stb.) képezhetők le.

Mivel a XAML az objektumok állapotának deklaratív meghatározására szolgál, markup vagy forráskód segítségével definiálhatunk WPF-vezérlőt. Például a következő XAML-kód:

```
<!-- WPF Button típus definíciója XAML-ben -->
<Button Name = "btnClickMe" Height = "40" Width = "100"
        Content = "Click Me" />
```

programozott módon a következőképpen jelenik meg:

```
// Ugyanazon WPF Button típus definíciója C#-kóddal.
Button btnClickMe = new Button();
btnClickMe.Height = 40;
btnClickMe.Width = 100;
btnClickMe.Content = "Click Me";
```

Hátrány viszont, hogy a XAML túl részletes lehet, és (az XML-dokumentumokhoz hasonlóan) érzékeny a kis- és nagybetűkre, ezért az összetett XAML-definíciók terjedelmes markupot eredményezhetnek. A legtöbb fejlesztőnek nem kell kézzel megírnia a WPF-alkalmazása teljes XAML-leírását. Ehelyett a