Az aktuális kiválasztás megszerzése

Végezetül adjunk hozzá egyedi írásvédett tulajdonságot az `AddNewCarDialog` párbeszédablakhoz `SelectedCar` néven, amely új `Car` objektumot ad vissza a hívónak a rács kiválasztott sorának értékei alapján:

```
public Car SelectedCar
{
  get
  {
    // A kiválasztott elem kasztolása a rácson XmlElement típusra.
    System.Xml.XmlElement carRow =
              (System.Xml.XmlElement)lstCars.SelectedItem;

    // Győződjünk meg róla, hogy a felhasználó kiválasztott valamit!
    if (carRow == null)
    {
      return null;
    }
    else
    {
      // Véletlenszerű sebesség generálása.
      Random r = new Random();
      int speed = r.Next(100);

      // Új Car elemet adunk vissza a választott XmlElement/sebesség
      // adatai alapján.
      return new Car(speed, carRow["Make"].InnerText,
      carRow["Color"].InnerText, carRow["PetName"].InnerText);
    }
  }
}
```

Vegyük észre, hogy `XmlElement` típusra kasztoltuk a `SelectedItem` tulajdonság (amely `System.Object` típusú) visszatérési értékét. Ez azért lehetséges, mert a `ListView` típusunk valóban kapcsolódik az `Inventory.xml` fájlhoz az adatkötési műveletünk révén. Amint elcsíptük az aktuális `XmlElement` típust, hozzáférhetünk a `Make`, a `Color` és a `PetName` elemekhez (a típusindexelő használatával), és kibonthatjuk az értékeket az `InnerText` meghívásával.

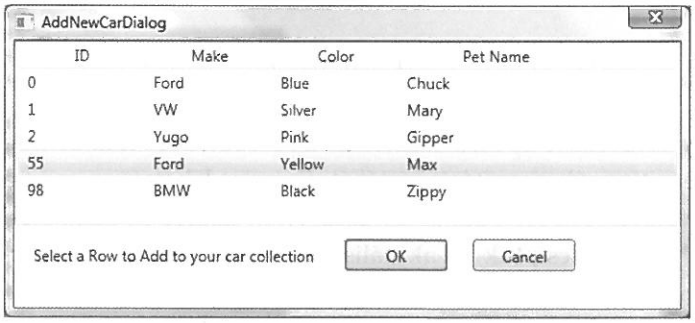
Megjegyzés Ha soha nem dolgoztunk a `System.Xml` névtér típusaival, akkor elég azt tudnunk, hogy az `InnerText` tulajdonság megszerzi az értéket egy XML-csomópont nyitó és záró elemei között. Például a `<Make>Ford</Make>` belső szövege Ford lenne.

Egyedi párbeszédablak megjelenítése

Most, hogy a párbeszédablakunk elkészült, elindíthatjuk azt a File > Add New Car menüelem kattintási eseménykezelőjéből:

```
private void AddNewCarWizard(object sender, RoutedEventArgs e)
{
  AddNewCarDialog dlg = new AddNewCarDialog();
  if (true == dlg.ShowDialog())
  {
    if (dlg.SelectedCar != null)
    {
      myCars.Add(dlg.SelectedCar);
    }
  }
}
```

A Windows Formshoz hasonlóan egy WPF-párbeszédablak megjelenhet modális párbeszédablakként (a `ShowDialog()` meghívásával) vagy nem modális párbeszédablakként (a `Show()` metódus meghívásával). Ha a `ShowDialog()` visszatérési értéke `true`, akkor kérjük az új Car objektumot a párbeszédablaktól, és adjuk hozzá az `ObservableCollection<T>` típusunkhoz. Mivel ez a gyűjteménytípus értesítéseket küld, ha a tartalma megváltozik, látni fogjuk, hogy a `ListBox` típusunk automatikusan frissíti magát, amikor új elemeket szúrunk be. A 29.35. ábra mutatja az egyedi párbeszédablakunk felhasználói felületét.



29.35. ábra: *Egyedi adatrács, hozzákötve egy XML-dokumentumhoz*

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

Ezzel befejeztük a WPF adatkötési motorjának és a felhasználói felület API-ban található alapvető vezérlőelemeinek vizsgálatát. A következő fejezetben lezárjuk a Windows Presentation Foundation tanulmányozását, megvizsgáljuk a grafikus renderelés, az erőforráskezelés és az egyedi témák készítésének szerepét.

Összefoglalás

Ez a fejezet áttekintette a WPF-vezérlőelemek néhány jellemzőjét, kezdve a függőségi tulajdonságok és a továbbított események elemzésével. Ezek a WPF-mechanizmusok nagyon fontosak a WPF-programozás több szempontjából, beleértve az adatkötést, az animációs szolgáltatásokat és más funkciókat. A fejezet folyamán konfigurálhattunk és átszabhattunk többféle vezérlőelemet, és megtanultuk elrendezni a felhasználói felület tartalmát különböző paneltípusokban.

Még fontosabb, hogy megvizsgáltuk a WPF-utasítások használatát. Emlékezzünk rá, hogy ezek a vezérlőelem-független események hozzácsatolhatók a felhasználói felület egy eleméhez vagy egy beviteli mozdulathoz, hogy automatikusan örököljék a kész működő szolgáltatásokat (mint a vágólap-műveletek). Beleástuk magunkat a WPF adatkötési motor mechanikájába is, és megtanultuk, hogyan köthetünk tulajdonságértékeket, egyedi objektumokat és XML-dokumentumokat a felhasználóifelület-réteghez. Megtanultuk azt is, hogyan készítsünk WPF-párbeszédablakokat, és felfedeztük az `IValueConverter` és az `ObservableCollection<T>` típusok szerepét.

HARMINCADIK FEJEZET

WPF 2D grafikus renderelés, erőforrások és témák

Ebben a fejezetben három független, de egymással kapcsolatban álló témával foglalkozunk, amelyek segítségével az előző két fejezetben vizsgált programoknál kifinomultabb Windows Presentation Foundation (WPF) alkalmazásokat készíthetünk. Elsőként a WPF kétdimenziós grafikus programozási API-kat tanulmányozzuk. Több módszert vizsgálunk meg, amelyekkel kétdimenziós geometriai képeket jeleníthetünk meg (alakzatokkal, rajzokkal és vizuális eszközökkel), és olyan egyszerű grafikai eszközökkel dolgozunk, mint az ecset és a toll. Vizsgálódásaink során megismerkedünk a WPF animációs szolgáltatásaival és a `System.Windows.Media.Animation` névtér típusaival.

Miután elsajátítottuk a WPF kétdimenziós grafikus renderelési/animációs alapismereteit, áttanulmányozzuk azokat a lehetőségeket, amelyeket az alkalmazás-erőforrások létrehozása, beágyazása és referenciái terén biztosít a WPF. Általános értelemben az „alkalmazás-erőforrás" sztringtáblázatokat, képfájlokat, ikonokat és az alkalmazások által használt, nem forráskód alapú entitásokat jelent. Míg ez a WPF esetén igaz, az „erőforrás" képviselhet egyedi grafikus és felhasználóifelület-objektumokat is, amelyeket későbbi használat céljából szeretnénk beágyazni egy szerelvénybe.

A fejezet utolsó témaköre kapcsolatot teremt két, látszólag nem összefüggő terület között: megvizsgáljuk, hogyan definiálhatunk stílusokat és sablonokat vezérlőelemeink számára. Látjuk majd, hogy a stílusok és a sablonok létrehozása szinte mindig a WPF grafikus renderelés/animációs szolgáltatásainak átfogó alkalmazását jelenti, valamint azt, hogy a szolgáltatásokat alkalmazás-erőforrásként a szerelvényünkbe csomagoljuk.

Megjegyzés A 28. fejezetből emlékezhetünk arra, hogy a WPF széles körű támogatást biztosít a háromdimenziós programozás számára. Ez a téma azonban e könyv hatókörén kívül esik, ha a WPF ezen funkciójával kapcsolatban további részletekre lenne szükségünk, érdemes elolvasnunk Matthew MacDonald *Pro WPF in C# 2008: Windows Presentation Foundation with .NET 3.5, Second Edition* (Apress, 2008) című könyvét.

A WPF grafikus renderelési szolgáltatásának filozófiája

A WPF a grafikus renderelés különleges típusát, az úgynevezett *visszatartott üzemmódú grafikát* alkalmazza. Ez annyit jelent, hogy mivel XAML vagy forráskód segítségével generáljuk a grafikus rendereléseket, a WPF feladata ezen képi elemek rögzítése, valamint az elemek optimális újrarajzolásának és frissítésének biztosítása. Így a grafikus adatok renderelése során a WPF folyamatosan jelen van, függetlenül attól, hogy a felhasználó az ablak átméretezésével, kicsinyítésével, másik ablak kitakarásával elrejti-e a képet.

Ezzel éles ellentétben, a korábbi Microsoft grafikus renderelési API-k (a GDI+ is) *azonnali üzemmódú* grafikus rendszerek voltak. Ebben a modellben a programozónak kellett gondoskodnia arról, hogy a megjelenített képi elemekre a rendszer az alkalmazás élettartama alatt megfelelően „emlékezzen" és frissítse azokat. Emlékezzünk rá a 27. fejezetből, hogy a GDI+ alatt egy téglalap megjelenítéséhez kezelnünk kell a Paint eseményt (vagy felüldefiniálhatjuk a virtuális `OnPaint()` metódust). A téglalap rajzolásához be kell olvasnunk a `Graphics` objektumot, és rendkívül fontos elkészítenünk a megfelelő infrastruktúrát – például tagváltozókat kell létrehozni, amelyek a téglalap pozícióját képviselik, valamint a programban meg kell hívni az `Invalidate()` metódust –, amely biztosítja, hogy a rendszer rögzítse a képet, ha a felhasználó átméretezi az ablakot. Az azonnali üzemmódú grafikáról a visszatartott üzemmódú grafikára váltás valóban hasznos előrelépés volt, mert így a programozóknak kevesebb grafikus kódot kell írniuk és karbantartaniuk.

Azonban ez nem jelenti azt, hogy a WPF grafikus API *teljesen* különbözik a korábbi renderelési eszközrendszerektől. Például a GDI+ rendszerhez hasonlóan, a WPF különböző ecset- és tolltípusokat támogat, kódolás segítségével lehetővé teszi a grafikák futásidejű megjelenítését, találattesztelési módszereket biztosít, és így tovább. Ha rendelkezünk GDI+ (vagy C/C++-alapú GDI) ismeretekkel, akkor bizonyára sokat tudunk arról, hogyan lehet a WPF segítségével alapvető renderelési műveleteket végrehajtani.

A WPF grafikus renderelési lehetőségei

A WPF-fejlesztés más jellemzőihez hasonlóan, a XAML- vagy a C#-forráskód mellett több lehetőség áll rendelkezésünkre, amikor azt próbáljuk eldönteni, *hogyan* hajtsuk végre a grafikus renderelést. A WPF a grafikus adatok megjelenítésének három különböző módját biztosítja:

- `System.Windows.Shapes`: A névtér több típust definiál, amelyek kétdimenziós geometriai objektumok (téglalapok, ellipszisek, sokszögek) megjelenítését teszik lehetővé. A típusok használata rendkívül egyszerű, de alkalmazásuk jelentős többletköltséggel jár.
- `System.Windows.Media.Drawing`: Ez az absztrakt ősosztály több egyszerű szolgáltatáskészletet definiál leszármazott típusok (`GeometryDrawing`, `ImageDrawing` stb.) számára.
- `System.Windows.Media.Visual`: Ez az absztrakt ősosztály a grafikus adatok megjelenítésének lehető legkönnyebb megközelítését biztosítja; azonban tekintélyes mennyiségű forráskód elkészítését követeli meg.

A motiváció, amely ugyanazon feladat (például a grafikus adatok megjelenítése) végrehajtásának három különböző módját biztosítja, a memória használattal és az alkalmazás teljesítményével hozható összefüggésbe. Mivel a WPF grafikaiművelet-intenzív rendszer, az alkalmazások nem ritkán több száz különböző képet jelenítenek meg az ablak felületén, és a megvalósítás módjának kiválasztása (alakzatok, rajzok, vizuális eszközök) jelentős hatást gyakorol a végeredményre.

Készítsük elő a terepet a következő témakörök számára, és kezdjük az egyes lehetőségek rövid áttekintésével. Haladjunk a „legnehezebb"-től a „legkönnyebb" felé. Ha szeretnénk a lehetőségeket saját magunk kipróbálni, hozzuk létre a WPFGraphicsOptions nevű új WPF Windows alkalmazást, a kezdeti `Window` típusunk nevét módosítsuk MainWindow-ra, és az ablak kezdeti XAML `<Grid>` definícióját cseréljük ki `<StackPanel>`-re:

```
<Window x:Class="WPFGraphicsOptions.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="WPFGraphicsOptions" Height="300" Width="300" >
  <StackPanel>

  </StackPanel>
</Window>
```

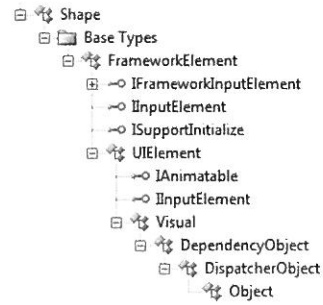
A Shape leszármazott típusainak használata

A `System.Windows.Shapes` névtér tagjai (`Ellipse`, `Line`, `Path`, `Polygon`, `Polyline` és `Rectangle`) biztosítják a kétdimenziós képek megjelenítésének legegyszerűbb módját, és alkalmasak viszonylag ritkán használt képek (például stilizált gomb egy régiójának) renderelésére vagy felhasználói együttműködést támogató grafikus kép alkalmazására. A típusok használata során rendszerint „ecsetet" kell választanunk a belső terület kitöltésére, és „tollat" a határvonal rajzolásához (a WPF ecset és toll opcióit a fejezet későbbi részében tárgyaljuk). Az alakzattípusok alapvető használatának bemutatására adjuk a következő kódot a `<StackPanel>` típusunkhoz, és eredményként egy kék körvonallal rendelkező világoskék téglalapot kapunk:

```
<!-- Rajzoljunk téglalapot a Shape típusok segítségével -->
<Rectangle Height="55" Width="105" Stroke="Blue"
           StrokeThickness="5" Fill="LightBlue"/>
```

Noha használatuk nevetségesen egyszerű, a típusok - szülőosztályuknak köszönhetően - elég terjedelmesek, mivel a `System.Windows.Shapes.Shape` osztály minden lehetséges igénynek igyekszik megfelelni. A származtatási láncban a `Shape` rengeteg szolgáltatást örököl a szülőosztályok hosszú sorától: a `Shape` „az-egy" `FrameworkElement`, amely egy `UIElement`, amely egy `Visual`, amely egy `DependencyObject`, `DispatcherObject` és `Object`!

Ezek az ősosztályok stílus- és sablon-, adatkötés-, valamint erőforráskezelés-támogatást biztosítanak a leszármazott típusok számára, és lehetővé teszik több esemény küldését, a billentyűzet- és az egérbemenet felügyeletét, valamint összetett elrendezéskezelési és animációs szolgáltatásokat nyújtanak. A 30.1. ábra a `Shape` típus származtatási láncát mutatja be a Visual Studio objektumböngészőjének segítségével.



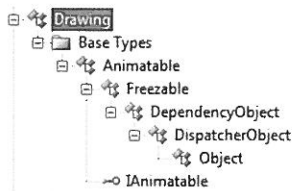
30.1. ábra: *A Shape leszármazott típusai átfogó funkcionalitást örökölnek szülőtípusaiktól*

Noha minden szülőosztály újabb funkcionalitással bővíti az osztályt, a `UIElement` kiemelt jelentőséggel bír. A `UIElement` például több mint *80 eseményt* definiál, amelyek a bemenet különböző formáit (egér, billentyűzet, Tablet PC-k stylusceruzája) kezelik. A `FrameworkElement` egy másik fontos típus, mivel az egérkurzor módosítását, az objektum élettartamát képviselő különböző eseményeket, környezetfüggő menük támogatását stb. biztosító tagokkal rendelkezik. Ennek fényben a `Shape`-leszármazott típusai ugyanolyan sokoldalúak, mint más felhasználóifelület-elemek, például a Button, a ProgressBar és más vezérlőelemek.

A lényeg, hogy a `Shape`-leszármazott típusainak használata egyszerű és hatékony, de ugyanezen tény miatt ezek a típusok nyújtják a kétdimenziós grafikus renderelés legnehézkesebb módját. A típusok alkalmazása helyénvaló „alkalmi renderelés" esetén (amelynek definíciója inkább megérzésen, nem tudományos alapokon nyugszik), illetve akkor, ha valóban olyan grafikus renderelésre van szükség, amely felhasználói együttműködést igényel. Ha grafikaiművelet-intenzív alkalmazást készítünk, valószínűleg találkozunk azokkal a teljesítménybeli előnyökkel, amelyeket a `Drawing` leszármazott típusainak használata biztosít.

A Drawing leszármazott típusainak használata

A `System.Windows.Media.Drawing` absztrakt ősosztály egy kétdimenziós felület lecsupaszított vázát képviseli. A leszármazott típusok (például a `GeometryDrawing`, az `ImageDrawing` és a `VideoDrawing`) egyszerűbbek, mint a `Shape`-leszármazott osztályai, mert származtatási láncukból a `UIElement` és a `FrameworkElement` is hiányzik. Emiatt a `Drawing` leszármazott típusai nem rendelkeznek belső támogatással a bemeneti események kezelésére (noha programozott módon végrehajthatunk találattesztelési logikát); azonban a típusok animálhatók, mivel az `Animatable` osztály a származtatási lánc tagja.



30.2. ábra: *A Drawing leszármazott típusai jóval egyszerűbbek, mint a Shape leszármazott típusai*

Másik lényeges különbség a `Drawing` és a `Shape` leszármazott típusai között, hogy a `Drawing` leszármazott típusai nem képesek önmaguk megjelenítésére, mivel nem a `UIElement` leszármazottai! A leszármazott típusokat a tartalmuk megjelenítéséhez egy hosztoló objektumban (`DrawingImage`, `DrawingBrush` vagy `DrawingVisual`) kell elhelyezni. Az összetevők szétválasztásának az az eredménye, hogy a `Drawing` leszármazott típusai egyszerűbbek a `Shape` leszármazott típusainál, és közben a típusok megtartják a kulcsfontosságú szolgáltatásokat.

Anélkül, hogy túlságosan elmerülnénk a részletekben, gondoljuk végig, az előző `Rectangle` objektumot hogyan lehetne rajzolásközpontú típusok segítségével renderelni (ha a könyv példáit követjük, helyezzük az alábbi markupot közvetlenül az előző `<Rectangle>` típusunk után):

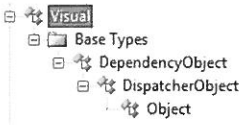
```
<!-- Rajzoljunk téglalapot a Drawing típusainak segítségével -->
<Image Height="55" Width="105">
  <Image.Source>
    <DrawingImage>
      <DrawingImage.Drawing>
        <GeometryDrawing Brush="LightBlue">
          <GeometryDrawing.Pen>
            <Pen Brush="Blue" Thickness="5"/>
          </GeometryDrawing.Pen>
          <GeometryDrawing.Geometry>
            <RectangleGeometry Rect="0,0,100,50"/>
          </GeometryDrawing.Geometry>
        </GeometryDrawing>
      </DrawingImage.Drawing>
    </DrawingImage>
  </Image.Source>
</Image>
```

A kimenet megegyezik az előző `<Rectangle>` típus kimenetével, de ez nyilvánvalóan sokkal részletesebb. A klasszikus „hosszabb kód a jobb teljesítmény érdekében" dilemmával állunk szemben. Szerencsére, ha XAML grafikus tervező eszközöket (például a Microsoft Expression Blend vagy a Microsoft Expression Design eszközt) alkalmazunk, a rendszer a színfalak mögött elkészíti a mögöttes markupot (a Microsoft Expression termékcsalád részletes ismertetését a 28. fejezetben találjuk).

A Visual leszármazott típusainak használata

Az absztrakt `System.Windows.Media.Visual` típus szolgáltatások minimális, de teljes készletét (renderelés, találattesztelés, átalakítás) biztosítja a leszármazott típus rendereléséhez, de nem nyújt támogatást további nem vizuális szolgáltatásokhoz, ez a forráskód felduzzadásához vezethet (bemeneti események, el-

rendezési szolgáltatások, stílusok és adatkötés). Ennek köszönhetően a `Visual` leszármazott típusai (`DrawingVisual`, `Viewport3DVisual` és `ContainerVisual`) a grafikus renderelési opciók közül a legegyszerűbb lehetőséget jelentik, és a legjobb teljesítményt biztosítják. Tanulmányozzuk a Visual típus egyszerű származtatási láncát a 30.3. ábrán!



30.3. ábra: *A Visual típus alapvető találattesztelést, koordináta-átalakítást és határolódoboz-számításokat biztosít*

Mivel a Visual típus „büszkélkedhet" a legszegényesebb funkcionalitással, csak korlátozottan támogatja a direkt XAML-definíciókat (hacsak nem olyan típusban alkalmazzuk a `Visual` típust, amely *kifejezhető* XAML-kóddal). Ezen típusok alkalmazása hasonlít a GDI/GDI+ renderelési API-k használatához, mivel a típusokat gyakran forráskód segítségével kezelhetjük. Ha így járunk el, az ablakot képviselő objektumgráfot manuálisan népesítjük be egyedi Visual-leszármazott típusokkal. Ehhez felül kell definiálnunk különböző virtuális metódusokat, amelyeket a WPF grafikus rendszer azért hív meg, hogy kiderítse, hány elemet kell renderelnie, és ismernie kell magát a megjeleníteni kívánt Visual elemet is.

Vizsgáljuk meg, hogyan alkalmazhatjuk a `Visual` leszármazott típusait kétdimenziós adatok megjelenítéséhez, ehhez nyissuk meg a főablak forráskódfájlját, majd alakítsuk megjegyzésre a teljes definíciót (így később gyorsan és kevés erőfeszítéssel visszaállíthatjuk a kódot):

```
/*
public partial class MainWindow : System.Windows.Window
{
  public MainWindow()
  {
    InitializeComponent();
  }
}
*/
```

Hozzuk létre a következő `Window`-leszármazott típust, amely közvetlenül az ablak felületén jelenít meg egy téglalapot, és kihagyja a XAML-markupban definiált tartalmat (az előző XAML-leírásokat a rendszer figyelmen kívül hagyja, és nem jeleníti meg):

```
public partial class MainWindow : System.Windows.Window
{
  // Az egyetlen rajzolási vizuális elem.
  private DrawingVisual rectVisual = new DrawingVisual();
  private const int NumberOfVisualItems = 1;

  public MainWindow()
  {
    InitializeComponent();

    // A téglalap létrehozásának segédfüggvénye.
    CreateRectVisual();
  }

  private void CreateRectVisual()
  {
    using (DrawingContext drawCtx = rectVisual.RenderOpen())
    {
      // A téglalap felső, bal oldali, alsó és jobb oldali
      // pozíciója.
      Rect rect = new Rect(50, 50, 105, 55);
      drawCtx.DrawRectangle(Brushes.AliceBlue,
                            new Pen(Brushes.Blue, 5), rect);
    }

    // Regisztráljuk a vizuális elemet az objektumfában,
    // így biztosítva, hogy támogassa az irányított eseményeket,
    // a találattesztelést stb.
    AddVisualChild(rectVisual);
    AddLogicalChild(rectVisual);
  }

  // A szükséges felüldefiniálások. A WPF grafikus rendszer
  // meghívja ezeket a metódusokat, és kideríti, hogy
  // hány elemet és pontosan mit kell renderelnie.
  protected override int VisualChildrenCount
  {
    get { return NumberOfVisualItems; }
  }

  protected override Visual GetVisualChild(int index)
  {
    // A gyűjtemény 0 alapú, tehát vonjunk le 1-et.
    if (index != (NumberOfVisualItems - 1))
      throw new ArgumentOutOfRangeException("index",
                              "Don't have that visual!");
    return rectVisual;
  }
}
```

Vegyük észre, hogy a `DrawingVisual` típus (`rectVisual`) a `RenderOpen()` metódust biztosítja, amely `DrawingContext` objektumot ad vissza. A GDI+ `Graphics` objektumához hasonlóan a `DrawingContext` több olyan metódussal rendelkezik, amely segítségével az elemek széles köre (`DrawRectangle()`, `DrawEllipse()` stb.) renderelhető. Ha elkészítettük a téglalapot, ezt követően két örökölt metódust (`AddVisualChild()` és `AddLogicalChild()`) hívunk meg, amelyek használata opcionális, de biztosítják, hogy az egyedi Visual-leszármazott típusunkat a rendszer az ablak objektumfájába integrálja.

Végül, de nem utolsósorban felül kell definiálnunk a `VisualChildrenCount` írásvédett tulajdonságot és a `GetVisualChild()` metódust. Ezeket a tagokat a WPF grafikus motor hívja, hogy meghatározhassa, pontosan mit kell renderelnie (ebben a példában egyetlen `DrawingVisual` elemet).

Mint látjuk, amint belépünk a Visual-leszármazott típusok használatának világába, rengeteg forráskóddal találkozunk, ennek azonban az az előnye, hogy általában nálunk az irányítás (és az ezzel együtt járó összetettség is).

Egyedi vizuális renderelési program készítése

Úgy állítottuk be az aktuális egyedi Visual renderelési műveletet, hogy az ablak tartalmát (például a `<StackPanel>` típust) szétrobbantottuk, ezért a rendszer azt nem jelenítette meg, csak a kódolt `DrawingVisual`-t. Most ismerjük meg egy kicsit mélyebben a Visual programozási rétegeit! Mi lenne, ha az ablak renderelésének nagy részét XAML-leírásokkal valósítanánk meg, és a Visual-réteget csak a felhasználói felület egy apró részének megvalósítására használnánk?

Egyik megközelítési lehetőség, ha a `FrameworkElement` egyedi leszármazott osztályát definiáljuk, és felüldefiniáljuk a virtuális `OnRender()` metódust. Ez a metódus (amelyet a `Shape`-leszármazott típusok a kimenetük megjelenítésére használnak) az előző `CreateRectVisual()` segédmetódusunk forráskódjához hasonló kódot foglalhat magában. Ha definiáltuk ezt az egyedi osztályt, akkor hivatkozhatunk az egyedi osztálytípusunkra az ablak XAML-leírásában.

Ennek bemutatására adjuk az aktuális projektünkhöz a `MyCustomRenderer` osztályt, amely kibővíti a `FrameworkElement` ősosztályt (ne felejtsük el a fájlunkba importálni a `System.Windows` és a `System.Windows.Media` névtereket), és a következőképpen valósítsuk meg a típust:

```
public class MyCustomRenderer : FrameworkElement
{
  // A téglalap alapértelmezett mérete.
  double rectWidth = 105, rectHeight = 55;
```

```
    // Tegyük lehetővé a felhasználó számára az alapértelmezett
    // értékek módosítását.
    public double RectHeight
    {
      set { rectHeight = value; }
      get { return rectHeight; }
    }
    public double RectWidth
    {
      set { rectWidth = value; }
      get { return rectWidth; }
    }

    protected override void OnRender(DrawingContext drawCtx)
    {
      // Először a szülő rendereléséről gondoskodjunk.
      base.OnRender(drawCtx);

      // Egyedi renderelésünk hozzáadása.
      Rect rect = new Rect();
      rect.Width = rectWidth;
      rect.Height = rectHeight;
      drawCtx.DrawRectangle(Brushes.LightBlue,
                            new Pen(Brushes.Blue, 5), rect);
    }
  }
```

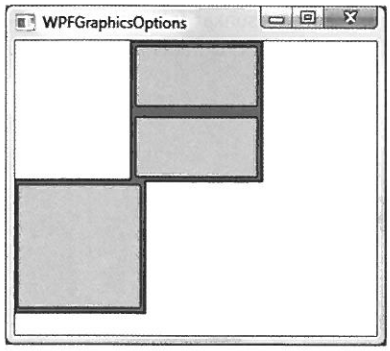
Az egyedi típusunkkal kapcsolatos események nagy része az `OnRender()` implementációban játszódik le. Vegyük észre, hogy a lokális `Rect` változó méretét a `rectHeight` és a `rectWidth` tagok alapján állítottuk be, amelyek nem feltétlenül szükségesek, de lehetővé teszik a kép méretének definiálását.

Úgy férhetünk hozzá ehhez a típushoz a XAML-kódon keresztül, ha olyan egyedi XML-névteret definiálunk, amely a típusunk nevéhez képeződik le, és ezt a prefixumot használjuk a típusunk létrehozásához. Tekintsük meg a `<StackPanel>` típusunk releváns módosításait (emlékezzünk arra a 28. fejezetből, hogy az XML-névtereket, amelyek saját .NET-névtereinkhez képezhetőek le, a `clr-namespace` tokennel kell definiálni):

```
<StackPanel xmlns:custom = "clr-namespace:WPFGraphicsOptions">
  ...
  <custom:MyCustomRenderer RectHeight ="100" RectWidth ="100"/>
</StackPanel>
```

Az alkalmazás futtatása előtt az eredeti főablak definíciónkat *tegyük ismét a forráskód részévé* – ne legyen megjegyzés –, viszont az egyedi ablak definíciója *megjegyzés legyen*. Ha ezt követően futtatjuk az alkalmazásunkat, a képernyőn három téglalap jelenik meg, amelyek mindegyike a WPF kétdimenziós programozási módszereinek egyikével készült (lásd a 30.4. ábrát).

Forráskód A WPFGraphicsOptions kódfájlokat a forráskódkönyvtár 30. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.



30.4. ábra: *Három téglalap, három különböző megközelítés*

A megfelelő megoldás kiválasztása

Megismerkedtünk három különböző megközelítéssel, amelyek segítségével a WPF kétdimenziós renderelési szolgáltatásaival (alakzatokkal, rajzokkal és vizuális eszközökkel) dolgozhatunk. A grafikák megjelenítésére Visual-leszármazott típusokkal valószínűleg akkor lehet szükség, ha egyedi vezérlőelemeket készítünk vagy nagyobb befolyást szeretnénk a renderelési folyamat felett. Ez azért jó, mert a `Visual` és a többi típus leszármazottaival folytatott munka az egyszerű XAML-leírásokhoz képest rengeteg erőfeszítéssel jár. Ennek fényében a fejezet ezen pontján nem merülünk el mélyebben a `Visual` renderelési API-k rejtelmeiben (további információkért lapozzuk fel a .NET Framework 3.5 SDK dokumentációját).

A `Drawing` leszármazott típusainak alkalmazása a tökéletes „arany középút" megközelítés, mivel ezek a `Shape` típusokhoz képest kevesebb erőfeszítést igényelnek ahhoz, hogy támogassák az alapvető, nem felhasználói felület szolgáltatásokat (például a találattesztelést). Ez a megközelítés a `Shape` típusok alkalmazásához képest több markup elkészítését igényli, de az így elkészült alkalmazás többletköltsége alacsonyabb. A fejezet későbbi részében megvizsgáljuk a `Drawing` leszármazott típusait részletesen.

A Shape típusok teljesen tökéletes megközelítést biztosítanak, ha egy meghatározott ablakban néhány kétdimenziós képet kell megjelenítenünk. Emlékezzünk rá, hogy ha olyan kétdimenziós alakzatokkal kell dolgoznunk, amelyek a hagyományos felhasználóifelület-elemek funkcionalitásával rendelkeznek, a Shape típusok tökéletes választást jelentenek, mivel a szükséges infrastruktúra már rendelkezésünkre áll.

Megjegyzés Soha ne feledjük, hogy a renderelési szolgáltatás választása rányomja bélyegét az alkalmazás teljesítményére! Szerencsére rendelkezésünkre áll különböző WPF profilkialakítási segédprogramok gyűjteménye, amelyekkel az aktuális alkalmazásunkat felügyelhetjük. Erről a .NET Framework 3.5 SDK dokumentációjának „Performance Profiling Tools for WPF" című részében további információkat találunk.

A Shape leszármazott típusainak felfedezése

A kétdimenziós grafikus renderelés vizsgálatának folytatásaként vizsgáljuk meg részletesen a System.Windows.Shapes névtér tagjait. Emlékezzünk rá, hogy ezek a típusok biztosítják a legegyértelműbb, de egyszersmind legterjedelmesebb módját a kétdimenziós képek megjelenítésének. Ez a névtér (amelyet a PresentationFramework.dll szerelvény definiál) viszonylag kicsi, és csak hat lezárt típust foglal magában, amelyek a Shape ősosztályt bővítik ki: az Ellipse, a Line, a Path, a Polygon, a Polyline és a Rectangle típusokat.

Mivel a Shape ősosztály „az-egy" FrameworkElement, a leszármazott típusokat tartalomként, XAML- vagy C#-kód segítségével hozzárendelhetjük, és e közben nem kell megküzdenünk a geometriai formák rajzolásának összetettségével:

```
<Window x:Class="SomeShapes.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="300" Width="300" >

  <!-- Ablak, amelynek tartalma egy kör -->
  <Ellipse Height = "100" Width = "100" Fill = "Black" />
</Window>
```

A többi felhasználóifelület-elemhez hasonlóan, ha olyan ablakot készítünk, amelyben több vezérlőt szeretnénk elhelyezni, a kétdimenziós típusokat a 29. fejezetben ismertetett módon, egy panel típusban kell definiálnunk.

A Shape ősosztály funkcionalitása

Míg a Shape funkcionalitását szülőosztályok hosszú sora biztosítja, a típus definiál néhány olyan tulajdonságot (melyek többsége függőségi tulajdonság), amelyeken a gyermektípusok is osztoznak. A tulajdonságok közül a legérdekesebbeket a 30.1. táblázat ismerteti.

Tulajdonságok	Jelentés
Fill	Lehetővé teszi, hogy a leszármazott típus belső részének megjelenítéséhez ecsettípust válasszunk.
GeometryTransform	Lehetővé teszi, hogy a leszármazott típus megjelenítésén átalakítást hajtsunk végre.
Stretch	Leírja, hogy a rendszer a lefoglalt helyen hogyan töltse ki az alakzatot. A tulajdonságot a megfelelő System.Windows.Media.Stretch felsorolt típus segítségével állíthatjuk be.
Stroke, StrokeDashArray StrokeEndLineCap, StrokeThickness	Ezen (és még további) vonásközpontú tulajdonságok határozzák meg az alakzat határvonalának megrajzolásakor a vonalak beállításait.

30.1. táblázat: *A Shape ősosztály kulcsfontosságú tulajdonságai*

Emlékezzünk rá, hogy a Shape leszármazott osztályai támogatják a találattesztelést, a témákat és stílusokat, az eszköztippeket és sok más szolgáltatást.

A Rectangle, az Ellipse és a Line típusok használata

Ha szeretnénk kipróbálni a leszármazott típusokat, hozzuk létre a FunWithSystemWindowsShapes új Visual Studio WPF Windows alkalmazást. A Rectangle, az Ellipse és a Line típusok XAML-deklarációja elég egyértelmű, és kevés magyarázatot igényel. A Rectangle típus egyik érdekes jellemzője, hogy a RadiusX és a RadiusY tulajdonságok definiálásával szükség esetén lehetővé teszi lekerekített sarkok renderelését. A Line az X1, X2, Y1 és Y2 tulajdonságokkal ábrázolja a kezdő- és végpontjait (vonalak leírása esetén a „magasság" és a „szélesség" adatoknak nem sok értelme lenne): a túl sok magyarázat helyett tanulmányozzuk a következő <StackPanel> típust:

```
<StackPanel>
  <!-- A sor ellenőrzi, hogy az alakzat területére húztuk-e
       az egeret -->
    <Line Name ="SimpleLine" X1 ="0" X2 ="50" Y1 ="0" Y2 ="50"
          Stroke ="DarkOliveGreen" StrokeThickness ="5"
          ToolTip ="This is a line!"
          MouseEnter ="SimpleLine_MouseEnter"/>

  <!-- Lekerekített sarokkal rendelkező téglalap -->
    <Rectangle RadiusX ="20" RadiusY ="50"
               Fill ="DarkBlue" Width ="150" Height ="50"/>
</StackPanel>
```

Amikor a kurzort a Line objektumra húztuk, a `SimpleLine` objektum `MouseEnter` eseménye az egérkurzor pozíciójával frissítette az ablak `Title` tulajdonságát:

```
protected void SimpleLine_MouseEnter(object sender,
                                     MouseEventArgs args)
{
  this.Title = String.Format("Mouse entered at: {0}",
                    args.GetPosition(SimpleLine));
}
```

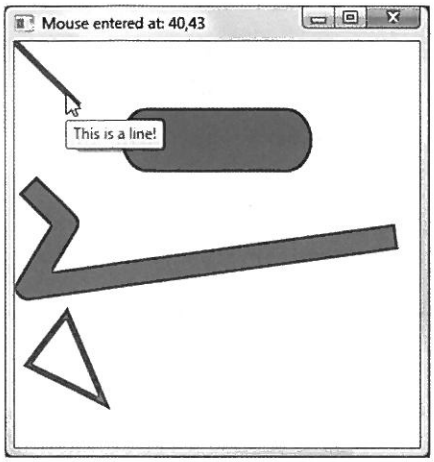
A Polyline, a Polygon és a Path típusok használata

A `Polyline` típus segítségével (x, y) koordináták gyűjteményét definiálhatjuk (a `Points` tulajdonság révén), és kapcsolt vonalszegmenseket rajzolhatunk, amelyeknek nem feltétlenül kell összekötni a végpontjait. A `Polygon` típus hasonló; azonban a típus definíciója szerint mindig összeköti a kezdő- és végpontokat. Vizsgáljuk meg az aktuális `<StackPanel>` alábbi bővítését:

```
<!-- A Polyline típusok nem rendelkeznek kapcsolódó
     végpontokkal -->
<Polyline Stroke ="Red" StrokeThickness ="20"
          StrokeLineJoin ="Round"
          Points ="10,10 40,40 10,90 300,50"/>

<!-- A Polygon mindig összeköti a végpontokat -->
<Polygon Fill ="AliceBlue" StrokeThickness ="5" Stroke ="Green"
         Points ="40,10 70,80 10,50" />
```

A 30.5. ábrán látható a `Shape`-leszármazott típusok megjelenített kimenete.

Az utolsó típus, a `Path` (amelyet most nem vizsgálunk meg) tulajdonképpen a `Rectangle`, az `Ellipse`, a `Polyline` és a `Polygon` típusok befoglaló halmaza, mivel a `Path` alkalmas ezen típusok bármelyikének megjelenítésére. Valójában a kétdimenziós típusokat kizárólag a `Path` segítségével is renderelhetnénk (de ez többletmunkát eredményezne).



30.5. ábra: *Renderelt Shape-leszármazott típusok*

Forráskód A FunWithSystemWindowsShapes kódfájlokat a forráskódkönyvtár 30. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A WPF-ecsettípusok használata

A WPF grafikus renderelési lehetőségeinek mindegyike (alakzatok, rajzok és vizuális eszközök) sokszor *ecseteket* használ, amelyek segítségével szabályozhatjuk, hogy a rendszer hogyan töltse ki a kétdimenziós felületek belsejét. A WPF hat különböző ecsettípust biztosít, amelyek mindegyike a `System.Windows.Media.Brush` osztályt bővíti ki. A `Brush` absztrakt osztály, azonban a 30.2. táblázatban ismertetett leszármazottainak a segítségével egy területet az összes rendelkezésünkre álló opció felhasználásával kitölthetünk.

Ecset típusa	Jelentés
`DrawingBrush`	A `Drawing` leszármazott objektumával (`GeometryDrawing`, `ImageDrawing` vagy `VideoDrawing`) fest ki egy területet.
`ImageBrush`	Egy képpel (amelyet az `ImageSource` objektum képvisel) tölt ki egy területet.
`LinearGradientBrush`	Egy területet lineáris átmenettel kifestő ecset.

Ecset típusa	Jelentés
`RadialGradientBrush`	Egy területet sugaras átmenettel kifestő ecset.
`SolidColorBrush`	Egyetlen színt tartalmazó ecset. A színt a `Color` tulajdonsággal lehet beállítani.
`VisualBrush`	A Visual leszármazott objektumával (`DrawingVisual`, `Viewport3DVisual` és `ContainerVisual`) fest ki egy területet

30.2. táblázat: *A WPF Brush leszármazott típusai*

A `DrawingBrush` és a `VisualBrush` típusok lehetővé teszik, hogy a fejezet elején vizsgált `Drawing`- vagy `Visual`-leszármazott típusok segítségével készítsünk ecsetet. A további ecsettípusok használata eléggé egyértelmű, és funkcionalitásuk hasonlít a GDI+ ecsettípusainak működéséhez. A következő részekben megvizsgáljuk a `SolidColorBrush`, a `LinearGradientBrush`, a `RadialGradientBrush` és az `ImageBrush` ecsettípusokat.

Megjegyzés Mivel nem kezelnek eseményeket, a következő példákat közvetlenül a 28. fejezetben elkészített XAML-nézegetőbe írhatjuk, és nem kell új Visual Studio 2008 WPF projektmunkaterületeket létrehoznunk az alkalmazásban.

Egyszínű ecsettípusok készítése

A `SolidColorBrush` típus `Color` tulajdonságával egyszínű ecsettípusokat készíthetünk. A `Color` tulajdonság a `System.Windows.Media.Color` típussal dolgozik, amely különböző tulajdonságok (például A, R, G és B) segítségével állítja be a színt. Az egyszínű ecsetek hasznosak lehetnek, de a sors iróniája, hogy általában nem kell explicit módon `SolidColorBrush` típust létrehoznunk, mivel a XAML támogat egy olyan típusátalakítót, amely a háttérben az ismert színneveket (például „Blue") `SolidColorBrush` objektumokra képezi le. Tanulmányozzuk a következő lehetőségeket, amely segítségével egy színnel tölthetünk ki egy `Ellipse` objektumot:

```
<StackPanel>
  <!-- A SolidColorBrush típus használata -->
  <Ellipse Fill ="DarkRed" Height ="50" Width ="50"/>
```

```
<!-- Típusátalakítóval működő egyszínű ecset -->
<Ellipse Height ="50" Width ="50">
  <Ellipse.Fill>
    <SolidColorBrush Color ="DarkGoldenrod"/>
  </Ellipse.Fill>
</Ellipse>

<!-- A SolidColorBrush és a Color típusok alkalmazása -->
<Ellipse Height ="50" Width ="50">
  <Ellipse.Fill>
    <SolidColorBrush>
      <SolidColorBrush.Color>
        <Color A ="40" R ="100" G ="87" B ="98"/>
      </SolidColorBrush.Color>
    </SolidColorBrush>
  </Ellipse.Fill>
</Ellipse>
</StackPanel>
```

A kimenet megfelel a várakozásoknak (három különböző színű kör); azonban a színséma meghatározásának megközelítése attól függ, hogy mekkora rugalmasságot várunk el az alkalmazásunktól. Ha például az `Opacity` tulajdonság (amely az átlátszóságot szabályozza) értékét kell módosítanunk, a `<SolidColorBrush>` elem deklarálásával közvetlenül hozzáférünk annak tagjaihoz. Más esetekben egyszerű sztring értéket hívhatunk segítségül, amelyet a `Fill` tulajdonsághoz rendelünk.

Megjegyzés A WPF grafikus API a `Brushes` segédosztályt biztosítja, amely meghatározza több tucat előre definiált szín tulajdonságait. Ez akkor bizonyul hasznos lehetőségnek, ha eljáráskódunkban egyszínű ecsetre van szükségünk.

Átmenetes ecsetek használata

A két átmenetes ecsettípus (a `LinearGradientBrush` és a `RadialGradientBrush`) lehetővé teszi, hogy két (vagy több) szín átmenetével töltsünk ki egy területet. A két típus között annyi a különbség, hogy amíg a `LinearGradientBrush` mindig egyenes vonal (amelyet tetszőleges pozícióba forgathatunk grafikus átalakítással vagy a kezdő- és végpont beállításával) segítségével képez a színek között átmenetet, a `RadialGradientBrush` meghatározott kezdőponttól kifelé elliptikus határvonal mentén képzi a színátmenetet. Nézzük meg a következő kódrészletet:

```
<!-- Lineárisan kitöltött téglalap -->
<Rectangle RadiusX ="15" RadiusY ="15" Height ="40" Width ="100">
  <Rectangle.Fill>
    <LinearGradientBrush StartPoint="0,0.5" EndPoint="1,0.5">
      <GradientStop Color="LimeGreen" Offset="0.0" />
      <GradientStop Color="Orange" Offset="0.25" />
      <GradientStop Color="Yellow" Offset="0.75" />
      <GradientStop Color="Blue" Offset="1.0" />
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>

<!--  Ellipszis sugaras színátmenet kitöltéssel -->
<Ellipse Height ="75" Width ="75">
  <Ellipse.Fill>
    <RadialGradientBrush GradientOrigin="0.5,0.5"
                         Center="0.5,0.5"
                         RadiusX="0.5" RadiusY="0.5">
      <GradientStop Color="Yellow" Offset="0" />
      <GradientStop Color="Red" Offset="0.25" />
      <GradientStop Color="Blue" Offset="0.75" />
      <GradientStop Color="LimeGreen" Offset="1" />
    </RadialGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

Vegyük észre, hogy az ecsettípusok `<GradientStop>` típusok listáját tartják karban (a listában tetszőleges számú típus lehet), amelyek a színt és az eltolás értékét határozzák meg. Az érték jelöli ki a képen, hogy a következő szín hol kezd összeolvadni az előző színnel.

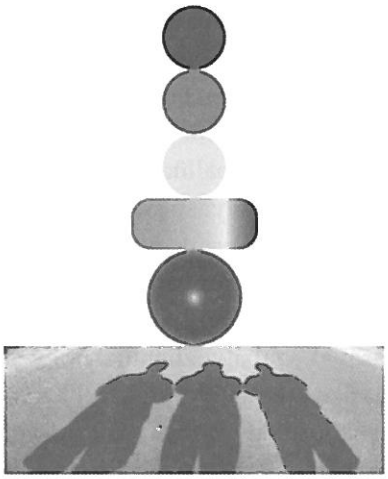
Az ImageBrush típus

Az utolsó ecsettípus, amelyet ebben a részben megvizsgálunk, az `ImageBrush`. Mint a neve is sugallja, a típus lehetővé teszi, hogy külső képfájlt (vagy beágyazott képerőforrást) töltsünk be az ecsettípus alapjaként. Ahhoz, hogy egy külső fájlt rendeljünk az `ImageBrush` típushoz, az egyik lehetőség az, ha az `ImageSource` tulajdonság értékét érvényes `BitmapImage` objektumra állítjuk. Vizsgáljuk meg a következő egyszerű definíciót, amely feltételezi, hogy az alábbi `*.xaml` fájl mellett számítógépünkön található a `Gooseberry0007.JPG` fájl is:

```
<!-- Képi ecsettel készült nagy téglalap -->
<Rectangle Height ="100" Width ="300">
  <Rectangle.Fill>
    <ImageBrush>
      <ImageBrush.ImageSource>
```

```
        <BitmapImage UriSource ="Gooseberry0007.JPG"/>
      </ImageBrush.ImageSource>
    </ImageBrush>
  </Rectangle.Fill>
</Rectangle>
```

A 30.6 ábrán az ecsettípusainkat láthatjuk működés közben.



30.6. ábra: *Különböző ecsettípusok működés közben*

Forráskód A FunWithBrushes.xaml kódfájlt a forráskódkönyvtár 30. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A WPF-tollak használata

Az ecsetekhez képest a tollak témaköre egészen egyszerű, mivel a `Pen` típus tulajdonképpen álruhába bújtatott `Brush` típus. A Pen olyan ecsettípust képvisel, amely egy `double` érték által meghatározott vastagsággal rendelkezik. Ennek ismeretében létrehozhatunk egy Pen típust, amelynek `Thickness` tulajdonsága olyan nagy értéket kap, hogy a típus valójában egy ecset! Rendszerint a `Pen` típus szerényebb vastagsággal rendelkezik, amely meghatározza, hogyan kell a kétdimenziós kép körvonalát megjeleníteni.

Az esetek többségében nem kell majd közvetlenül létrehoznunk `Pen` típust, mivel ezt a rendszer indirekt módon végrehajtja, ha olyan tulajdonságoknak, mint például a `StrokeThickness`, értéket megadunk. Egyedi `Pen` típus

készítése rendkívül hasznos lehet, ha a `Drawing` leszármazott típusaival dolgozunk (hamarosan látjuk, miért). Mielőtt megvizsgálnánk a tollak testre szabását, tanulmányozzuk a következő példát:

```
<Pen Thickness="10" LineJoin="Round" EndLineCap="Triangle"
     StartLineCap="Round" />
```

Ez a `Pen` típus beállítja a `LineJoin` tulajdonságot, amely szabályozza, hogyan kell két vonal között megjeleníteni a kapcsolódási pontot (például a sarkokat). Az `EndLineCap` – mint azt a neve is sejteti – szabályozza, hogyan kell egy vonal végpontját (ebben az esetben háromszög) megjeleníteni, a `StartLineCap` pedig a vonal kezdőpontjánál határozza meg ugyanezt a beállítást.

A `Pen` típusnál beállíthatjuk a *vonal stílusát*, amely befolyásolja, hogy a toll hogyan húzza a vonalat. Az alapértelmezett beállítás szerint egyetlen színt alkalmazunk (ahogyan azt az adott ecset megköveteli); azonban a `DashStyle` tulajdonságot bármely egyedi `DashStyle` objektumhoz hozzárendelhetjük. Míg egyedi `DashStyle` objektumok létrehozásával szabályozhatjuk, hogy a Pen típus hogyan jelenítse meg az adatait, a `DashStyle` segédosztály több statikus tagot definiál, amelyek alapértelmezett stílusokat biztosítanak. Mivel ezek egy *osztály statikus tagjai*, nem egy felsorolt típus értékei, alkalmaznunk kell a XAML `{x:Static}` markupbővítőt:

```
<Pen Thickness="10" LineJoin="Round" EndLineCap="Triangle"
StartLineCap="Round"
        DashStyle = "{x:Static DashStyles.DashDotDot}" />
```

Már rendelkezünk ismeretekkel a Pen típusról, ezért most használjuk a típusokat különböző `Drawing`-leszármazott típusokban.

A Drawing leszármazott típusainak vizsgálata

Emlékezzünk rá, hogy a `Shape` típusok segítségével bármilyen interaktív kétdimenziós felületet előállíthatunk, de a típusok terjedelmes származtatási láncának köszönhetően használatuk jelentős többletköltséggel jár. Alternatív megoldásként a WPF kifinomult rajz- és geometriaprogramozási interfészt biztosít, amellyel egyszerűbb kétdimenziós képeket megjeleníteni. Az API belépési pontja az absztrakt `System.Windows.Media.Drawing` osztály, amely

önmagában pusztán egy határoló téglalapot definiál a megjelenített kép befoglalására. A WPF öt típust biztosít, amelyek kibővítik a `Drawing` típust. Mint azt a 30.3. táblázatban láthatjuk, a típusok mindegyike a tartalom rajzolásának speciális fajtáját képviseli.

Típus	Jelentés
`DrawingGroup`	Segítségével különálló `Drawing`-leszármazott típusok gyűjteményét lehet egyetlen összetett renderelésben egyesíteni.
`GeometryDrawing`	Segítségével kétdimenziós alakzatokat lehet megjeleníteni.
`GlyphRunDrawing`	Segítségével szöveges adatokat lehet a WPF grafikus renderelési szolgáltatásaival megjeleníteni.
`ImageDrawing`	Segítségével határoló téglalapban lehet képfájlt megjeleníteni.
`VideoDrawing`	Segítségével audio- vagy videofájlt lehet lejátszani (nem „rajzolni"). A típus funkcionalitását teljes mértékben csak eljáráskóddal aknázhatjuk ki. Ha a XAML segítségével szeretnénk videofájlokat lejátszani, a MediaPlayer típus jobb választás.

30.3. táblázat: *A WPF Drawing leszármazott típusai*

Önmagában a típusok mindegyike rendkívül hasznos, de kétdimenziós képek megjelenítéséhez a `GeometryDrawing` típus a legjobb választás, a következő részben ezt a típust mutatjuk be részletesen. Röviden, a `GeometryDrawing` típus egy *geometriai típust* jelent, amely a kétdimenziós kép szerkezetét részletezi, `Brush` leszármazott típus tölti ki a belsejét, és Pen típus rajzolja meg a határvonalát.

A Geometry típusok szerepe

A `GeometryDrawing` típussal leírt geometriai szerkezet valójában a WPF geometriaközpontú osztálytípusainak egyike, vagy geometriaközpontú típusok gyűjteménye, amelyek egyetlen egységként működnek. A geometriák mindegyikét leírhatjuk XAML- vagy C#-forráskóddal is. A geometriák a `System.Windows.Media.Geometry` ősosztályból származnak, amely a leszármazott típusokban közös tagokat definiál. Néhány tagot megtekinthetünk a 30.4. táblázatban.

Tag	Jelentés
`Bounds`	Tulajdonság, amely segítségével létrehozhatjuk az aktuális határoló téglalapot.
`FillContains()`	Lehetővé teszi annak eldöntését, hogy egy meghatározott `Point` (vagy egyéb `Geometry` típus) egy adott `Geometry`-leszármazott típus határain belül található-e. Ez a tag a találattesztelési számításokhoz nyújt segítséget.
`GetArea ()`	`Double` értéket ad vissza, amely a `Geometry`-leszármazott típus által lefoglalt teljes területet képviseli.
`GetRenderBounds()`	`Rect` típust ad vissza, amely a `Geometry`-leszármazott típus megjelenítéséhez alkalmazható legkisebb befoglaló téglalapot határozza meg.
`Transform`	Lehetővé teszi, hogy a renderelés módosításához `Transform` objektumot rendeljünk a geometriához.

30.4. táblázat: *A System.Windows.Media.Geometry típus tagjai*

A WPF rengeteg `Geometry`-leszármazott típust biztosít. A típusokat két egyszerű kategóriába sorolhatjuk: az alapvető alakzatok és az útvonalak csoportjába. A geometriai típusok első csoportjának segítségével – ide tartozik a `RectangleGeometry`, az `EllipseGeometry`, a `LineGeometry` és a `PathGeometry` – alapvető alakzatokat jeleníthetünk meg. Szerencsére ez a négy típus a már vizsgált `System.Windows.Media.Shapes` típusok funkcionalitását másolja (és sok esetben azonos tagokkal rendelkezik).

A renderelési műveleteink nagy részében az alapvető alakzatokkal remekül boldogulhatunk. Ne feledjük, ha különleges geometriák alkalmazására van szükség, a WPF több olyan kiegészítő típust biztosít, amelyek a `PathGeometry` típussal együtt működnek. A `PathGeometry` „útvonaldarabok" gyűjteményét tartja karban, és a következőket foglalja magában: `ArchSegment`, `BezierSegment`, `LineSegment`, `PolyBezierSegment`, `PolyLineSegment`, `PolyQuadraticBezierSegment` és `QuadraticBezierSegment`.

Egyszerű rajzoló geometria szétdarabolása

Vizsgáljuk meg közelebbről a fejezet elején létrehozott <GeometryDrawing> típust:

```
<GeometryDrawing Brush ="LightBlue">
  <GeometryDrawing.Pen>
    <Pen Brush ="Blue" Thickness ="5"/>
  </GeometryDrawing.Pen>
  <GeometryDrawing.Geometry>
    <RectangleGeometry Rect="0,0,100,50"/>
  </GeometryDrawing.Geometry>
</GeometryDrawing>
```

Emlékezzünk rá, hogy a <GeometryDrawing> típus ecsetet, tollat és tetszőleges WPF-geometriatípusokat foglalhat magában. A példában a nyitó elemben a Brush tulajdonság segítségével indirekt módon definiáltuk a világoskék SolidColorBrush típust. A Pen típust a tulajdonság-elem szintaxissal deklaráljuk, és létrehozzuk a kék ecsetet, amely meghatározott vastagsággal rendelkezik. A <GeometryDrawing> típus Geometry tulajdonságához a <RectangleGeometry> értéket rendeljük.

Ha ezt a XAML-markupot közvetlenül a <Page> hatókörében (vagy bármely ContentControl-leszármazott típusban) szeretnénk elkészíteni a xamlpad.exe segítségével, markuphibát kapunk, amely lényegében azt tudatja velünk, hogy a <GeometryDrawing> nem bővíti ki a UIElement ősosztályt, és ezért értékként nem rendelhetjük hozzá a Content tulajdonsághoz.

Ez egy roppant érdekes kérdést vet fel a Drawing-leszármazott típusok használatával kapcsolatban: a típusok nem rendelkeznek semmilyen felhasználói interfésszel! Ezen típusok, mint például a <GeometryDrawing> leírják, hogy a kétdimenziós elem *hogyan jelenne meg*, ha megfelelő tárolóba helyeznénk. A WPF három különböző hosztoló objektumot biztosít a Drawing objektumok számára: a DrawingImage, a DrawingBrush és a DrawingVisual objektumot.

Drawing típusok a DrawingImage típusban

A DrawingImage típus segítségével a rajzoló geometriát WPF <Image> vezérlőelembe helyezhetjük. Így, ha az előző <GeometryDrawing> típust szeretnénk megjeleníteni, a következő módon kell a típust becsomagolnunk:

```
<Image>
  <Image.Source>
    <DrawingImage>
      <DrawingImage.Drawing>
        <GeometryDrawing Brush ="LightBlue">
          ...
        </GeometryDrawing>
      </DrawingImage.Drawing>
    </DrawingImage>
  </Image.Source>
</Image>
```

Vegyük észre, hogy az Image típus `Source` tulajdonságához rendeltünk egy `Drawing`-leszármazott típust.

Drawing típusok a DrawingBrush típusban

Ha inkább `DrawingBrush` típussal csomagoljuk a `<GeometryDrawing>` típust, gyakorlatilag összetett egyedi ecsetet hozunk létre, mivel a `DrawingBrush` valójában a `Brush`-leszármazott típusok egyike (az ecsetekkel kapcsolatos bővebb információkat a következő részben találjuk). Bárhol alkalmazhatjuk, ahol `Brush` típus szükséges, például a `<Window>` típus `Background` tulajdonságánál:

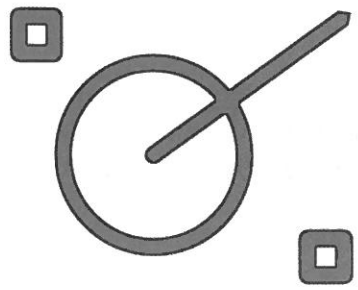
```
<Window x:Class="FunWithDrawingAndGeometries.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="FunWithDrawingAndGeometries" Height="190" Width="224">

  <!-- Az ablak hátterét állítsuk egyedi DrawingBrush típusra -->
  <Window.Background>
    <DrawingBrush>
      <DrawingBrush.Drawing>
        <GeometryDrawing Brush ="LightBlue">
          <GeometryDrawing.Pen>
            <Pen Brush ="Blue" Thickness ="5"/>
          </GeometryDrawing.Pen>
          <GeometryDrawing.Geometry>
            <RectangleGeometry Rect="0,0,100,50"/>
          </GeometryDrawing.Geometry>
        </GeometryDrawing>
      </DrawingBrush.Drawing>
    </DrawingBrush>
  </Window.Background>
</Window>
```

Összetett rajzoló geometria

A `DrawingImage` objektum több különálló `Drawing` objektumból állhat, amelyeket a `<DrawingGroup>` típusba helyezünk bonyolultabb kétdimenziós képek előállításához. Vizsgáljuk meg a következő Image típust, amely forrásaként a `<DrawingImage>` típust alkalmazza:

```
<Image>
  <Image.Source>
    <DrawingImage>
      <DrawingImage.Drawing>
        <!-- Különböző geometriák csoportja -->
          <DrawingGroup>
            <GeometryDrawing>
              <GeometryDrawing.Geometry>
                <GeometryGroup>
                  <RectangleGeometry Rect="0,0,20,20" />
                  <RectangleGeometry Rect="160,120,20,20" />
                  <EllipseGeometry Center="75,75" RadiusX="50"
                                   RadiusY="50" />
                  <LineGeometry StartPoint="75,75"
                                EndPoint="180,0" />
                </GeometryGroup>
              </GeometryDrawing.Geometry>
              <!-- Egyedi toll, amellyel megrajzoljuk a
                   határvonalakat -->
              <GeometryDrawing.Pen>
                <Pen Thickness="10" LineJoin="Round"
                     EndLineCap="Triangle" StartLineCap="Round">
                  <Pen.Brush>
                  <LinearGradientBrush>
                    <GradientStop Offset="0.0" Color="Red" />
                    <GradientStop Offset="1.0" Color="Green" />
                  </LinearGradientBrush>
                  </Pen.Brush>
                </Pen>
              </GeometryDrawing.Pen>
            </GeometryDrawing>
          </DrawingGroup>
      </DrawingImage.Drawing>
    </DrawingImage>
  </Image.Source>
</Image>
```

A `<DrawingImage>` egy `<GeometryGroup>` típust tartalmazó `<DrawingGroup>` típus, amely létrehozza a két téglalapból, egy ellipszisből és egy vonalból álló képet. A képek határvonalait egyedi tolltípussal jelenítjük meg, amely egyedi `LinearGradientBrush` típust foglal magában. A végeredményt a 30.7. ábrán láthatjuk.

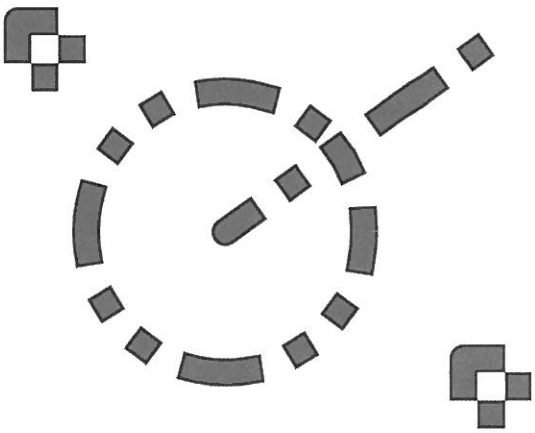


30.7. ábra: *DrawingGroup típust tartalmazó kép*

Ahhoz, hogy a `Pen` típus `DashStyle` típust – például a (korábban látott) `DashStyles.DashDotDot` típust – alkalmazzon, a kódot a következőképpen kell módosítani:

```
<Pen Thickness="10" LineJoin="Round"
  EndLineCap="Triangle" StartLineCap="Round"
  DashStyle = "{x:Static DashStyles.DashDotDot}" >
  <Pen.Brush>
    <LinearGradientBrush>
      <GradientStop Offset="0.0" Color="Red" />
      <GradientStop Offset="1.0" Color="Green" />
    </LinearGradientBrush>
  </Pen.Brush>
</Pen>
```

A végeredmény a 30.8. ábrán látható.



30.8. ábra: *Pen típus, amely vonás-pont-pont beállítással rendelkező DashStyle típussal dolgozik*

Forráskód A FunWithDrawingGeometries.xaml kódfájlt a forráskódkönyvtár 30. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A felhasználóifelület-transzformációk szerepe

Mielőtt rátérnénk az animációs szolgáltatások témakörére, zárjuk le a kétdimenziós grafikus renderelés tanulmányozását a *transzformációk* vizsgálatával. A WPF több olyan típust biztosít, amely kibővíti a `Transform` absztrakt ősosztályt. Az ősosztályt bármely `FrameworkElement` típuson (például a `Shape` típus leszármazottain, valamint olyan felhasználóifelület-elemeken, mint a Button, a TextBox stb.) alkalmazhatjuk. A típusok segítségével meghatározott szögben jeleníthetjük meg a `FrameworkElement` típust, a felületen elferdíthetjük, különböző módon növelhetjük vagy összenyomhatjuk a képet.

A Transform-leszármazott típusok

A 30.5. táblázatban a kulcsfontosságú `Transform` típusokat találjuk.

Típus	Jelentés
`MatrixTransform`	Tetszőleges mátrixtranszformációt hoz létre, amellyel kétdimenziós síkban objektumokat vagy koordináta-rendszereket kezelhetünk.
`RotateTransform`	Az óramutató járásával megegyező irányában elforgat egy objektumot egy kétdimenziós (x,y) koordináta-rendszer meghatározott pontja körül.
`ScaleTransform`	Egy objektumot kétdimenziós (x,y) koordináta-rendszerben méretez.
`SkewTransform`	Elferdít egy objektumot egy kétdimenziós (x,y) koordináta-rendszerben.
`TransformGroup`	Összetett `Transform` típust képvisel, amely több `Transform` objektumot foglal magában.

30.5. táblázat: *A System.Windows.Media.Transform típus kulcsfontosságú leszármazottai*

Ha létrehozunk egy `Transform` leszármazott objektumot, alkalmazhatjuk a `FrameworkElement` ősosztály két tulajdonságára. A `LayoutTransform` tulajdonság azért hasznos, mert a rendszer végrehajtja a transzformációt mielőtt az elemeket egy panelben megjelenítené. Másrészről a `RenderTransform` tulaj-

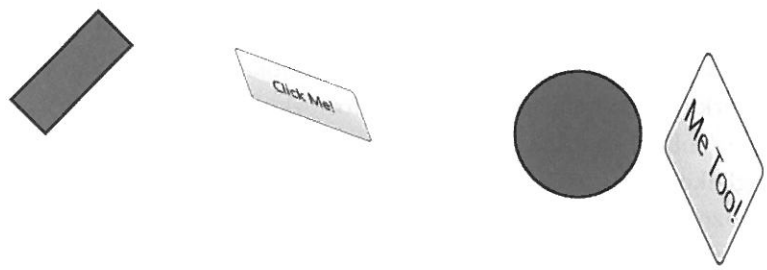
donságot akkor alkalmazza a rendszer, ha az elemek már a tárolóikban helyezkednek el, ezért előfordulhat, hogy az elemek a transzformációt követően fedik egymást. Azok a típusok, amelyek kibővítik a `UIElement` típust, értéket rendelhetnek a `RenderTransformOrigin` tulajdonsághoz. A tulajdonság a transzformáció kezdéséhez szükséges (x,y) pozíciót határozza meg.

Transzformációk alkalmazása

Tételezzük fel, hogy a `<Grid>` típusunk egyetlen sort tartalmaz négy oszloppal. Minden egyes cellában elforgatunk, ferdítünk és átméretezünk különböző `UIElement` típusokat, például a következőképpen:

```
<!-- Rectangle típus forgatási transzformációval -->
<Rectangle Height ="100" Width ="40" Fill ="Red" Grid.Row="0"
           Grid.Column="0">
  <Rectangle.LayoutTransform>
    <RotateTransform Angle ="45"/>
  </Rectangle.LayoutTransform>
</Rectangle>

<!-- Button típus ferdítési transzformációval -->
<Button Content ="Click Me!" Grid.Row="0" Grid.Column="1"
        Width="95" Height="40">
  <Button.RenderTransform>
    <SkewTransform AngleX ="20" AngleY ="20"/>
  </Button.RenderTransform>
</Button>

<!-- Ellipse típus, amelyet 20%-kal átméreteztünk -->
<Ellipse Fill ="Blue" Grid.Row="0" Grid.Column="2" Width="5"
         Height="5">
  <Ellipse.RenderTransform>
    <ScaleTransform ScaleX ="20" ScaleY ="20"/>
  </Ellipse.RenderTransform>
</Ellipse>

<!--  A Button típust ferdítettük, forgattuk, majd ismét
      ferdítettük -->
<Button Content ="Me Too!" Grid.Row="0" Grid.Column="3" Width="50"
        Height="40">
  <Button.RenderTransform>
    <TransformGroup>
    <SkewTransform AngleX ="20" AngleY ="20"/>
    <RotateTransform Angle ="45"/>
    <SkewTransform AngleX ="5" AngleY ="20"/>
    </TransformGroup>
  </Button.RenderTransform>
</Button>
```

Az első típusunk, a `<Rectangle>` a `RotateTransform` típust alkalmazza, amely az `Angle` tulajdonság segítségével 45 fokos szögben jeleníti meg a `UI` elemet. Az első `<Button>` típus a `SkewTransform` objektummal dolgozik, amely – (legalább) az `AngleX` és az `AngleY` tulajdonságok alapján – ferdére állítja a vezérlő megjelenített képét. A `<ScaleTransform>` típus – amelyet az `<Ellipse>` típus alkalmaz – jelentős mértékben növeli a kör magasságát és szélességét. Vegyük észre, hogy az `<Ellipse>` típus `Height` és `Width` tulajdonságainak értéke 5, miközben a megjelenített kimenet sokkal nagyobb. Végül, de nem utolsósorban a `<Button>` típus a `<TransformGroup>` típus segítségével alkalmazza a ferdítést és az elforgatást. A megjelenített kimenet a 30.9. ábrán látható.



30.9. ábra: *Transzformációk alkalmazása*

Forráskód A FunWithTransformations.xaml kódfájlt a forráskódkönyvtár 30. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A WPF animációs szolgáltatásai

A kétdimenziós (és háromdimenziós) grafikus renderelés teljes körű támogatását biztosító API-n kívül a WPF programozási interfészt is nyújt az animációs szolgáltatások támogatásához. Az „animáció" fogalma pörgő vállalati logóknak, forgó képi erőforrások sorozatának (amelyek a mozgás illúzióját keltik), a képernyőn keresztülpattogó szövegnek vagy meghatározott programoknak – például videojátékok vagy multimédia-alkalmazások – a képét is jelenti.

A WPF animációs API-kat valóban használhatjuk ilyen célra is, de az animációkat tulajdonképpen bármikor alkalmazhatjuk, ha az alkalmazásunknak szeretnénk további funkcionalitást kölcsönözni. Például egy képernyőn animációt készíthetünk egy gombhoz, amelyet a rendszer kissé felnagyít, ha az egérkurzort az objektum határain belülre mozgatjuk (majd vissza-

állítja az objektum eredeti méretét, ha az egérkurzort a határvonalon kívülre mozgatjuk). Egy ablak bezárásához animációt készíthetünk, amely speciális vizuális effektust alkalmaz (az ablak lassan elhalványul, és teljesen átlátszóvá válik). Röviden, ha szeretnénk lebilincselő élményeket nyújtani a felhasználóknak, bármely alkalmazással (üzleti alkalmazás, multimédia-programok, videojátékok stb.) használhatjuk a WPF animációtámogatását.

A WPF többi lehetőségéhez hasonlóan az animációk készítése sem újdonság. Azonban a korábban használt API-kkal ellentétben (mint például a GDI+) a fejlesztőknek a szükséges infrastruktúrát nem kell kézzel létrehozniuk. A WPF-alkalmazásokban nem kell háttérszálakat (vagy időzítőket) létrehoznunk az animációsorozat előreléptetéséhez, nem kell egyedi típusokat definiálnunk az animáció számára, és nem szükséges matematikai számításokkal bajlódnunk.

A WPF többi funkciójához hasonlóan az animációkat elkészíthetjük pusztán, XAML-markup alkalmazásával, csak C#-forráskóddal vagy a kettő kombinációjával. Azonban, ha a Microsoft Expression Blendet (amelyet a WPF-fel bővebben foglalkozó fejezetekben néhányszor már említettünk) használjuk, integrált eszközökkel és varázslókkal tervezhetünk meg egy animációt anélkül, hogy egyetlen sornyi C#-kódot vagy XAML-markupot kellene használnunk. Ez a megközelítés grafikusművészek számára ideális, akik nem tudnának mit kezdeni ezekkel a részletekkel.

Az Animation utótaggal rendelkező típusok szerepe

A WPF animációs támogatás felfedezéséhez kezdjük a vizsgálódást az alapvető animáció típusokkal, amelyeket a `PresentationCore.dll` szerelvény `System.Windows.Media.Animation` névtere foglal magában. A névtérben több osztálytípust találhatunk, amelyek utótagja az `Animation` (`ByteAnimation`, `ColorAnimation`, `DoubleAnimation`, `In32Animation` stb.). Nyilvánvaló, hogy ezekkel a típusokkal közvetlenül *nem* alkalmazhatunk meghatározott adattípusú változón animációs sorozatot (pontosan hogyan animálnánk a „9" értéket az `Int32Animation` segítségével?). Ehelyett az `Animation` utótaggal rendelkező típusokat a mögöttes típusokkal megegyező típusok *függőségi tulajdonságaihoz* kapcsolhatjuk.

Megjegyzés Fontos ismételten kiemelni, hogy az `Animation` utótaggal rendelkező típusok csak a függőségi tulajdonságokkal, és nem a CLR-tulajdonságokkal működnek együtt (lásd a 29. fejezetet). Ha animációs objektumokat próbálunk alkalmazni CLR-tulajdonságokon, fordítási idejű hibát kapunk.

Gondoljunk például a `Label` típus `Height` és `Width` tulajdonságaira, amelyek mindegyike `double` értéket magában foglaló függőségi tulajdonság. Ha olyan animációt szeretnénk definiálni, amely meghatározott időtartam alatt növeli a címke magasságát, a `DoubleAnimation` objektumot a `Height` tulajdonsághoz kapcsolhatjuk, és a WPF-re bízhatjuk a tényleges animáció végrehajtásának részleteit. Egy másik példa, ha egy ecsettípus színénél a zöld–sárga átmenetet szeretnénk megvalósítani, a `ColorAnimation` típust hívhatjuk segítségül.

Függetlenül attól, hogy melyik `Animation` utótaggal rendelkező típust szeretnénk használni, a típusok mindegyike több kulcsfontosságú tulajdonságot definiál, amelyek szabályozzák az animáció végrehajtásához szükséges kezdő- és záróértékeket.

- `To`: Ez a tulajdonság határozza meg az animáció záróértékét.
- `From`: Ez a tulajdonság határozza meg az animáció kezdőértékét.
- `By`: Ez a tulajdonság határozza meg azt mennyiséget, amellyel az animáció kezdőértéke változik.

Annak ellenére, hogy az `Animation` utótaggal rendelkező típusok támogatják a `To`, a `From` és a `By` tulajdonságokat, a típusok nem az ősosztály virtuális tagjain keresztül kapják ezeket az értékeket. Ez annak köszönhető, hogy a tulajdonságok által becsomagolt mögöttes típusok nagyon eltérőek (egész számok, színek, vastagság objektumok stb.), és az összes lehetőség megjelenítése a `System.Object` segítségével bedobozolási/kidobozolási többletet jelentene a veremalapú adatok esetén.

Felmerülhet bennünk a kérdés, hogy miért nem a .NET generikus típusok segítségével definiáltunk egyetlen animációs osztályt egyetlen paramétertípussal (például `Animate<T>`). Mivel az animált függőségi tulajdonságok több alap adattípust (színek, vektorok, egész számok, sztringek stb.) alkalmaznak, várakozásainkkal ellentétben ez nem a legtisztább megoldás (továbbá ne feledjük, hogy a XAML csak limitált támogatást biztosít a generikus típusok számára).

A Timeline ősosztály szerepe

Noha a virtuális `To`, `From` és `By` tulajdonságok definiálására nem készült külön ősosztály, az `Animation` utótaggal rendelkező típusok egy közös ősosztályon osztoznak: a `System.Windows.Media.Timeline` ősosztályon. Mint az a 30.6. táblázatban látható, a típus további tulajdonságokat biztosít, amelyek az animáció léptetését szabályozzák.

Tulajdonságok	Jelentés
`AccelerationRatio`, `DecelerationRatio`, `SpeedRatio`	Ezekkel a tulajdonságokkal az animációsorozat általános léptetését szabályozhatjuk.
`AutoReverse`	A tulajdonság lekérdezi vagy beállítja, hogy az idősor visszafelé haladjon-e miután a normál iterációt befejezte.
`BeginTime`	A tulajdonság lekérdezi vagy beállítja az időt, amikor az idősor elkezdi a lejátszást. Az alapértelmezett érték 0, amely azonnal indítja az animációt.
`Duration`	A tulajdonság segítségével beállíthatunk időtartamot, amíg a rendszer az idősort lejátssza.
`FillBehavior`, `RepeatBehavior`	A tulajdonságokkal szabályozhatjuk, hogy mi történjen, ha az idősor lejárt (például a rendszer megismételje az animációt, ne történjen semmi, stb.).

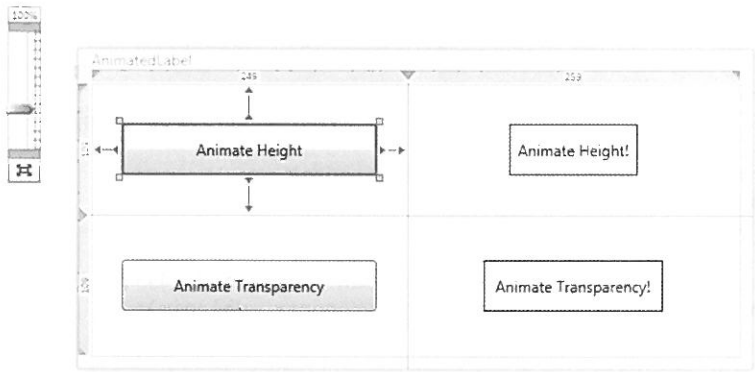
30.6. táblázat: *A Timeline ősosztály kulcsfontosságú tagjai*

Animáció készítése C#-forráskódból

A WPF animációs szolgáltatásainak vizsgálata során először a `DoubleAnimation` típust alkalmazzuk, amellyel a főablakban lévő címkék különböző tulajdonságait szabályozzuk. Készítsük el az `AnimatedLabel` WPF Windows alkalmazást, és tervezzünk `<Grid>` típust, amely két sort és két oszlopot foglal magában. Az első oszlopban minden cellában helyezzünk el egy-egy gombot, és kezeljük vezérlők kattintási eseményeit. A második oszlop celláiban helyezzünk el címkéket (`lblHeight` és `lblTransparency` névvel). A 30.10. ábrán egy lehetséges felhasználói felületet láthatunk.
Az alábbi kód alapján implementáljuk a kattintási eseménykezelőket:

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

    protected void btnAnimatelblMessage_Click(object sender,
                                              RoutedEventArgs args)
  {
    // Növeljük a címke magasságát.
    DoubleAnimation dblAnim = new DoubleAnimation();
```

```
    dblAnim.From = 40;
    dblAnim.To = 60;
    lblHeight.BeginAnimation(Label.HeightProperty, dblAnim);
  }

  protected void btnAnimatelblTransparency_Click(object sender,
                                                 RoutedEventArgs args)
  {
    // Módosítsuk a címke átlátszóságát.
    DoubleAnimation dblAnim = new DoubleAnimation();
    dblAnim.From = 1.0;
    dblAnim.To = 0.0;
    lblTransparency.BeginAnimation(Label.OpacityProperty, dblAnim);
  }
}
```



30.10. ábra: *Az AnimatedLabel alkalmazás kezdeti felhasználói felülete*

Vegyük észre, hogy minden kattintási esemény kezelőjében beállítjuk a `DoubleAnimation` típus `From` és `To` értékeit, amelyek a kezdő- és záróértéket képviselik. Ezt követően a megfelelő címkén hívjuk meg a `BeginAnimation()` metódust, majd adjuk át a kapcsolódó vezérlő (ismét a címke) megfelelő függőségi tulajdonságát, amelyet az animációt végrehajtó `Animation` utótaggal rendelkező objektum követ.

Megjegyzés Emlékezzünk a 29. fejezetre, ahol a függőségi tulajdonságok kapcsán megtanulhattuk, hogy a `DependencyObject` típus nyilvános, írásvédett statikus mezőinek segítségével jelenítjük meg az (opcionális) CLR-tulajdonságcsomagoló adott függőségi tulajdonságát.

Ha most futtatjuk az alkalmazást, és a gombokra kattintunk, láthatjuk, hogy a `lblHeight` címke mérete nagyobb, a `lblTransparency` gomb pedig lassan elhalványul, majd eltűnik.

Az animáció ütemezésének szabályozása

Alapértelmezés szerint az animációban a From és a To tulajdonságokhoz rendelt értékek közötti átmenet ideje egy másodperc. Ha például módosítani szeretnénk a kattintási eseménykezelőt, amely a címke Height tulajdonságának értékét 40-ről 200-ra növeli (pillanatnyilag kisebb mértékű növekedéssel dolgozunk), az átmenet továbbra is nagyjából egy másodpercet venne igénybe.

Ha az animáció átmenetei idejére egyedi időtartamot szeretnénk definiálni, akkor a Duration tulajdonság segítségével beállíthatjuk a Duration objektum egy példányát. Az időtartamot meghatározhatjuk, ha a Duration konstruktornak átadjunk egy TimeSpan objektumot. Vizsgáljuk meg az aktuális eseménykezelők következő módosítását, amely négy másodperc alatt növeli a címke magasságát, és a másik címkét tíz másodperc alatt teljesen átlátszóvá teszi:

```
protected void btnAnimatelblMessage_Click(object sender,
                                          RoutedEventArgs args)
{
  // Az animáció 4 másodperc alatt befejeződik.
   DoubleAnimation dblAnim = new DoubleAnimation();
  dblAnim.From = 40;
  dblAnim.To = 200;
  dblAnim.Duration = new Duration(TimeSpan.FromSeconds(4));
  lblHeight.BeginAnimation(Label.HeightProperty, dblAnim);
}

protected void btnAnimatelblTransparency_Click(object sender,
                                               RoutedEventArgs args)
{
  // Módosítjuk a címke átlátszóságát.
  DoubleAnimation dblAnim = new DoubleAnimation();
  dblAnim.From = 1.0;
  dblAnim.To = 0.0;
  dblAnim.Duration = new Duration(TimeSpan.FromSeconds(10));
  lblTransparency.BeginAnimation(Label.OpacityProperty, dblAnim);
}
```

Megjegyzés Az Animation utótaggal rendelkező típusok BeginTime tulajdonsága szintén TimeSpan objektumot vesz fel. Emlékezzünk rá, hogy ezzel a tulajdonsággal beállíthatjuk az animációsorozat elindítása előtti várakozási időt.

Animáció lejátszása visszafelé és folyamatos ismétlése

Beállíthatjuk az `AutoReverse` tulajdonságot `true` értékre, és az `Animation` utótaggal rendelkező típusokat utasíthatjuk, hogy az animációt a sorozat befejezését követően visszafelé játsszák le. A kattintási eseménykezelőjének következő módosításával a címke magassága 40 képpontról 100 képpontra növekszik, majd visszazsugorodik 100 képpontról 40-re (mindezt nyolc másodperc alatt, négy másodpercig egy-egy irányban):

```
// Ha befejeződött a sorozat, az animációt visszafelé játsszuk le.
dblAnim.AutoReverse = true;
```

Ha szeretnénk az animációt néhányszor megismételni (vagy az aktiválást követően folyamatosan lejátszani), beállíthatjuk a `RepeatBehavior` tulajdonságot, amely minden `Animation` utótaggal rendelkező típusban rendelkezésünkre áll. A `RepeatBehavior` tulajdonság értéke a megegyező névvel rendelkező objektum egy példánya. Ha egyszerű numerikus értéket adunk át a konstruktornak, meghatározhatjuk az ismétlések számát. Másrészről, ha konstruktornak átadjunk egy `TimeSpan` objektumot, beállíthatjuk az időtartamot, ameddig a rendszer ismétli az animációt. Ha végtelenített animációt szeretnénk készíteni, megadhatjuk a `RepeatBehavior.Forever` értéket. Vizsgáljuk meg a különböző módszereket, amelyekkel módosíthatjuk a címke magasságát:

```
// Végtelen ciklus.
dblAnim.RepeatBehavior = RepeatBehavior.Forever;

// Ismétlés háromszor.
dblAnim.RepeatBehavior = new RepeatBehavior(3);

// Ismétlés 30 másodpercig.
dblAnim.RepeatBehavior =
        new RepeatBehavior(TimeSpan.FromSeconds(30));
```

Forráskód Az AnimatedLabel kódfájlokat a forráskódkönyvtár 30. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Animáció készítése XAML-leírással

Ha dinamikusan kell az animáció állapotát kezelnünk, ennek legjobb módja a forráskód alkalmazása. Az előző példában pontosan ezt tettük. Ha azonban előre definiált, „fix" animációval dolgozunk, amely nem igényel futási idejű beavatkozást, a teljes animációsorozatot elkészíthetjük XAML-leírással. Ez a folyamat nagyrészt megegyezik azzal, amelyet már megvizsgáltunk; azonban a különböző `Animation` utótaggal rendelkező típusokat *forgatókönyv* típusokba csomagoljuk. A forgatókönyv típusokat a rendszer *eseménytriggerekhez* kapcsolja.

Vizsgáljunk meg egy teljes animációs példát, amelyet XAML-markuppal definiálunk. A példa célja, hogy a XAML segítségével megjelenítse az előző példában alkalmazott, lankadatlanul növekedő, majd zsugorodó címkét. Tanulmányozzuk az alábbi markupot, amelyet a következő részekben alaposan megvizsgálunk:

```
<Label Content = "Interesting...">
  <Label.Triggers>
    <EventTrigger RoutedEvent = "Label.Loaded">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard TargetProperty = "Height">
            <DoubleAnimation From ="40" To = "200"
                             Duration = "0:0:4"
                             RepeatBehavior = "Forever"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Label.Triggers>
</Label>
```

A Storyboard típus szerepe

Ha a legbelső elemtől indulunk kifelé, először a `<DoubleAnimation>` típussal találkozhatunk. A típus ugyanazokkal a tulajdonságokkal dolgozik, amelyeket a forráskódban beállítottunk (`To`, `From`, `Duration` és `RepeatBehavior`). Mint említettük, az `Animation` utótaggal rendelkező típusokat a `<Storyboard>` típusban helyezzük el. A típus segítségével leképezhetjük az animációt a szülőtípus meghatározott tulajdonságára a `TargetProperty` tulajdonsággal (amely ebben az esetben a `Height` tulajdonság).

Az indirekció ezen szintjét az indokolja, hogy a XAML nem támogat olyan szintaxist, amellyel objektumokon hívhatunk metódusokat, mint például a `BeginAnimation()` metódust a címkén. Tulajdonképpen a `<Storyboard>` és a `<BeginStoryboard>` szülőtípusok képviselik a következő forráskód XAML-központú verzióját:

```
// Az animációs logikát a XAML-ben a forgatókönyvek képviselik.
lblHeight.BeginAnimation(Label.HeightProperty, dblAnim);
```

Az <EventTrigger> használata

Miután a `<Storyboard>` típust definiáltuk, a következő feladat a típust magában foglaló eseménytrigger meghatározása. A WPF a triggerek három különböző típusát támogatja, amelyek segítségével műveletkészletet definiálhatunk. A rendszer ezeket hajtja végre, ha adott esemény bekövetkezik.

A triggerek első típusa megvizsgálja a függőségi tulajdonságok feltételeit a típuson. Ha függőségi tulajdonságokhoz definiálunk triggereket, a `<Trigger>` elemet hívjuk segítségül a trigger meghatározásakor. A triggerek második típusa a normál .NET-tulajdonságtípusokról gondoskodik, és a `<DataTrigger>` elemben definiálhatjuk. A triggerek ezen típusa adatkötési műveletek végrehajtásához lehet hasznos. A stílusok és témák tanulmányozásakor, a fejezet későbbi részében megvizsgáljuk a `<Trigger>` elemet.

Az aktuális példa szempontjából utolsó fontos triggertípus az (`<EventTrigger>` elemben definiált) *eseménytrigger*, amelyet WPF-animációk készítése során használunk. Itt az `<EventTrigger>` elem a címke `Loaded` eseményéhez kapcsolódik. Ha az esemény bekövetkezik, a rendszer végrehajtja a `<DoubleAnimation>` sorozatot a címke `Height` tulajdonságán.

Forráskód Az AnimationInXaml.xaml kódfájlt a forráskódkönyvtár 30. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A kulcsképkocka-animáció szerepe

A WPF animációs rendszer utolsó jellemzője, amelyet megvizsgálunk, a *kulcsképkockák* alkalmazása. A `System.Windows.Media.Animation` névtér több tagot is tartalmaz, amelyek az `AnimationUsingKeyFrames` utótaggal rendelkeznek (`ByteAnimationUsingKeyFrames`, `ColorAnimationUsingKeyFrames`, `DoubleAnimationUsingKeyFrames`, `In32AnimationUsingKeyFrames` stb.).

A típusok mindegyikének van `Duration` tulajdonsága, amely szabályozza az animációsorozat teljes időtartalmát. Azonban a kulcsképkockák feladata tájékoztatni az animációs rendszert arról, hogy mikor kell őket használni.

Az `Animation` utótaggal rendelkező típusokkal ellentétben – amelyek csak a kezdő- és a végpont között mozoghatnak – a *kulcsképkocka* lehetővé teszi, hogy a meghatározott időpontokban végrehajtott animáció számára speciális értékek gyűjteményét hozzuk létre. Például az `AnimationUsingKeyFrames` utótaggal rendelkező típusok segítségével elkészíthetünk egy animációt, amelyben egy kör pattog az ablak körül, egyedi kép mozog egy geometriai alakzat határvonala mentén vagy időszeletek szerint cseréli egy szövegdobozban a színeket.

Az `AnimationUsingKeyFrames` utótaggal rendelkező elem hatókörében három különböző kulcsképkocka-típusgyűjteményt helyezhetünk el, amelyek mindegyike az animált elem mozgáskeretét szabályozza:

- `LinearXXXKey Frame`: A lineáris kulcsképkocka típusok segítségével egyenes vonal pontjai mentén mozgathatjuk az elemet.
- `SplineXXXKey Frame`: A `spline` kulcsképkocka típusokkal Bezier-görbe ívén mozgathatunk egy elemet a `KeySpline` tulajdonság segítségével.
- `DiscreteXXXKey Frame`: A diszkrét kulcsképkocka típusok nem biztosítanak átmenetet a kulcsképkockák között. Ez például akkor lehet hasznos, ha sztringadatokat „animálunk", amelyek méretét növeljük, vagy határvonallal dolgozunk, amelynek színe változik.

Hasonló minta szerint a részelemek pontos neve az animált elem típusán (double, szín, logikai érték stb.) alapul. Az XXX nyilvánvalóan helyőrző szerepet tölt be. A három kulcsképkocka típus valódi neve a `LinearDoubleKeyFrame`, a `SplineDoubleKeyFrame` és a `DiscreteDoubleKeyFrame` neveken alapul.

Animáció diszkrét kulcsképkockákkal

A diszkrét kulcsképkocka típusok bemutatásához tételezzük fel, hogy olyan gombot szeretnénk készíteni, amely animálja a tartalmát: három másodperc alatt megjeleníti az „OK!" feliratot, másodpercenként egy-egy karaktert. Tételezzük fel továbbá, hogy ezt a rendszer rögtön azután megteszi, hogy a gombot betöltötte, és az animációt folyamatosan ismétli. Ezt a viselkedést megfigyelhetjük, ha a 28. fejezetben elkészített `SimpleXamlApp.exe` programban elhelyezzük a következő XAML-kódot:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Grid>
    <Button Name="myButton" Height="40"
            FontSize="16pt" FontFamily="Verdana" Width = "100">
    <Button.Triggers>
      <EventTrigger RoutedEvent="Button.Loaded">
        <BeginStoryboard>
          <Storyboard>
            <StringAnimationUsingKeyFrames
                   RepeatBehavior = "Forever"
                   Storyboard.TargetName="myButton"
                   Storyboard.TargetProperty="Content"
                   Duration="0:0:3" FillBehavior="HoldEnd">
              <DiscreteStringKeyFrame Value="" KeyTime="0:0:0" />
              <DiscreteStringKeyFrame Value="O" KeyTime="0:0:1" />
              <DiscreteStringKeyFrame Value="OK"
                                      KeyTime="0:0:1.5" />
              <DiscreteStringKeyFrame Value="OK!"
                                      KeyTime="0:0:2" />
            </StringAnimationUsingKeyFrames>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger>
    </Button.Triggers>
    </Button>
  </Grid>
</Window>
```

Vegyük észre, hogy először a gombunk számára definiáltunk egy eseménytriggert, amely biztosítja, hogy a gomb betöltése után a rendszer végrehajtja a forgatókönyvet. A `Storyboard.TargetName` és a `Storyboard.TargetProperty` értékek segítségével a `StringAnimationUsingKeyFrames` típus feladata a gomb tartalmának módosítása. A `<StringAnimationUsingKeyFrames>` elemünk hatókörében három diszkrét `DiscreteStringKeyFrame` típust definiáltunk, amelyek két másodperc alatt módosítják a gomb tartalmát (vegyük észre, hogy az időtartam, amelyet a `StringAnimationUsingKeyFrames` beállít, összesen három másodperc, tehát a `"!"` és az újrakezdődő `"O"` megjelenése között rövid szünetet láthatunk).

Forráskód Az AnimatedButtonWithDiscreteKeyFrames.xaml kódfájlt a forráskódkönyvtár 30. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Animáció lineáris kulcsképkockákkal

A lineáris kulcsképkockákat működés közben láthatjuk, ha megvizsgáljuk a következő XAML-kódot, amely kezdőpontként a gomb középpontját használja, és teljesen körbeforgatja a gombot. Ha a 360 fokos fordulat befejeződött, a gomb fejjel lefelé fordul (majd ismét visszaáll). Tételezzük fel, hogy a következő XAML-címkézést definiáltuk a `<Grid>` típusban:

```
<!-- A gomb körbeforog, majd megfordul, ha rákattintunk -->
<Button Name="myAnimatedButton" Width="120" Height = "40"
  RenderTransformOrigin="0.5,0.5" Content = "OK">

<Button.RenderTransform>
  <RotateTransform Angle="0"/>
</Button.RenderTransform>

<!-- Az animáció akkor indul, ha a gombra kattintunk -->
<Button.Triggers>
  <EventTrigger RoutedEvent="Button.Click">
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimationUsingKeyFrames
            Storyboard.TargetName="myAnimatedButton"
            Storyboard.TargetProperty=
                 "(Button.RenderTransform).(RotateTransform.Angle)"
            Duration="0:0:2" FillBehavior="Stop">
          <DoubleAnimationUsingKeyFrames.KeyFrames>
            <LinearDoubleKeyFrame Value="360" KeyTime="0:0:1" />
            <DiscreteDoubleKeyFrame Value="180" KeyTime="0:0:1.5" />
            </DoubleAnimationUsingKeyFrames.KeyFrames>
          </DoubleAnimationUsingKeyFrames>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Button.Triggers>
</Button>
```

A gomb definíciójának elején értéket adunk a `RenderTransformOrigin` tulajdonságnak, amely biztosítja, hogy a forgatás során pontosan a gomb középpontját használjuk az elforgatás középpontjaként. Ezt követően beállítjuk a megjelenítési transzformáció kezdőértékét a beágyazott `<Button.RenderTransform>` tag segítségével (vegyük észre, hogy a kezdeti szög nulla). Végül eseménytriggert definiálunk, amely biztosítja, hogy a rendszer végrehajtsa a forgatókönyvet, ha a felhasználó a gombra kattint.

Miután elkészültünk a kezdeti beállításokkal, hozzuk létre a `<Storyboard>` taget, amely a `DoubleAnimationUsingKeyFrames` típust használja. Vegyük észre, hogy a forgatókönyv célpontja a gomb példányunk (`myAnimatedButton`), és az objektumon a tulajdonság, amellyel végül dolgozunk, az `Angle` tulajdonság. Tanulmányozzuk az új XAML-szintaxist, amelyet akkor kell alkalmaznunk, amikor a tulajdonság értékéhez függőségi tulajdonságot rendelünk:

```
<DoubleAnimationUsingKeyFrames
  Storyboard.TargetName="myAnimatedButton"
  Storyboard.TargetProperty=
            "(Button.RenderTransform).(RotateTransform.Angle)"
  Duration="0:0:2" FillBehavior="Stop">
```

Mint látjuk, a függőségi tulajdonságokat zárójelek közé kell helyeznünk; ennek köszönhetően az egyetlen - a félkövér betűkkel kiemelt - kódsorban leírhatjuk, hogy „a gombhoz tartozó `RotateTransform` objektum `Angle` tulajdonságához készítünk animációt". Az animáció végrehajtásához engedélyezett időt két másodpercre állítsuk be.

A `DoubleAnimationUsingKeyFrames` típusunk hatókörében két kulcsképkocka-típussal bővítjük a kódot. Az első (`LinearDoubleKeyFrame`) típus egy másodperc alatt 360 fokkal elforgatja a gombot. Nagyjából fél másodperccel később a `DiscreteDoubleKeyFrame` típus 180 fokkal elforgatja (és fejtetőre állítja) a gombot. Ha az animáció (a `DoubleAnimationUsingKeyFrames` típus `Duration` tulajdonságának köszönhetően fél másodperccel később) véget ér, a gomb visszafordul a megfelelő irányba, mivel a `DoubleAnimationUsingKeyFrames` típus `FillBehavior` tulajdonságának értéke Stop (amely a kiinduló állapotba állítja vissza az elemet).

Forráskód A SpinButtonWithLinearKeyFrames.xaml kódfájlt a forráskódkönyvtár 30. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Ezzel a WPF révén rendelkezésünkre álló alapvető animációs szolgáltatások (és a kétdimenziós renderelési módszerek) vizsgálatának végére értünk. A témakörök mindegyikéből készülhetne önálló könyv is, ha az összes újdonságot és érdekességet ismertetni szeretnénk. Mostanra már biztosan megfelelő alapokkal rendelkezünk a folytatáshoz, a következő témakörben figyelmünket az alkalmazás-erőforrások beágyazása felé fordítjuk.

A WPF erőforrásrendszere

A következő feladatunk, hogy megvizsgáljuk az alkalmazás-erőforrások beágyazásának és hozzáférésének látszólag nem kapcsolódó témakörét. A WPF az erőforrások két típusát támogatja. Az első típushoz a *bináris erőforrások* tartoznak, amelyek a legtöbb programozó által a hagyományos értelemben „erőforrásnak" tekintett elemeket (bittérkép fájlok, ikonok, sztringtáblázatok stb.) képviselik.

Az erőforrások másik típusa, az *objektum erőforrások* vagy *logikai erőforrások* bármilyen .NET-objektumot képviselhetnek, amelyet névvel látunk el és szerelvénybe ágyazzuk. Látjuk majd, hogy a logikai erőforrások akkor bizonyulnak hasznosnak, ha grafikus adatokkal kell dolgoznunk, mivel az erőforráskönyvtárban rendszeresen alkalmazott grafikai primitíveket (ecseteket, tollakat, animációkat stb.) definiálhatunk.

A bináris erőforrások használata

Mint említettük, a bináris erőforrások a .NET-alkalmazás kiegészítő darabjai, mint például sztringtáblák, ikonok és képfájlok (például vállalati logók, animált képek stb.). Ha a Visual Studio segítségével készítünk WPF-alkalmazást, a számítógépet utasíthatjuk arra, hogy `Resource` fordítóopció meghatározásakor külső erőforrást ágyazzon a szerelvénybe. Ennek bemutatására készítsünk új WPF Windows alkalmazást FunWithResources néven. Egészítsük ki a főablak kezdeti XAML-definícióját három oszloppal rendelkező `<Grid>` elemmel, amelynek első cellájában egy Image vezérlő van:

```
<Window x:Class="FunWithResources.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="FunWithResources" Height="207" Width="612"
  WindowStartupLocation="CenterScreen">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition/>
      <ColumnDefinition/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Image Grid.Column ="0" Name ="companyLogo"/>
  </Grid>
</Window>
```

Az első cél, hogy bináris erőforrásként beágyazzuk a képfájlt az alkalmazásunkba. Az erőforrás segítségével beállítjuk a companyLogo `Image` vezérlőelem `Source` tulajdonságát. A választott képfájl segítségével a Visual Studio Project > Add Existing Item menüpontjában hozzáadjuk az elemet az aktuális projekthez (a példában feltételezzük, hogy a fájl neve `IntertechBlue.gif`).

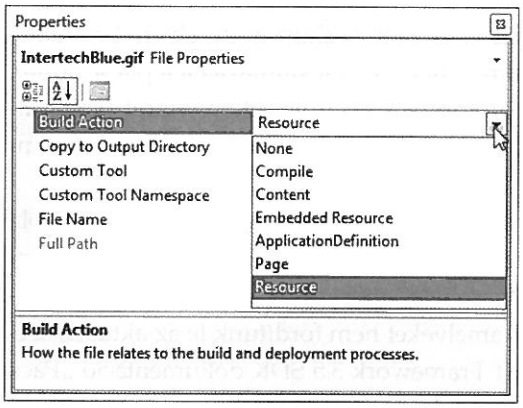
A Resource Build Action

Ha külső képfájlt adtunk az alkalmazásunkhoz, akkor a fájl megjelenik a Solution Explorer listájában. Ha kiválasztjuk, a Properties segítségével utasíthatjuk a fordítót, hogyan dolgozza fel a külső fájlokat a Build Action alkalmazásával (lásd a 30.11. ábrát).

Ha az alapértelmezés szerinti Resource beállítást választjuk, a fordító a .NET-szerelvénybe ágyazza az adatokat, ezért ezeket a külső fájlokat nem kell az alkalmazással együtt szállítanunk. Tételezzük fel, hogy a képfájlunk Build Action elemét Resources értékre állítottuk. Ekkor a következőképpen módosíthatjuk az Image vezérlőelem XAML-definícióját (vegyük észre, hogy a bináris erőforrás nevére név szerint hivatkozunk):

```
<Image Grid.Column ="0" Name ="companyLogo"
           Source ="IntertechBlue.gif"/>
```

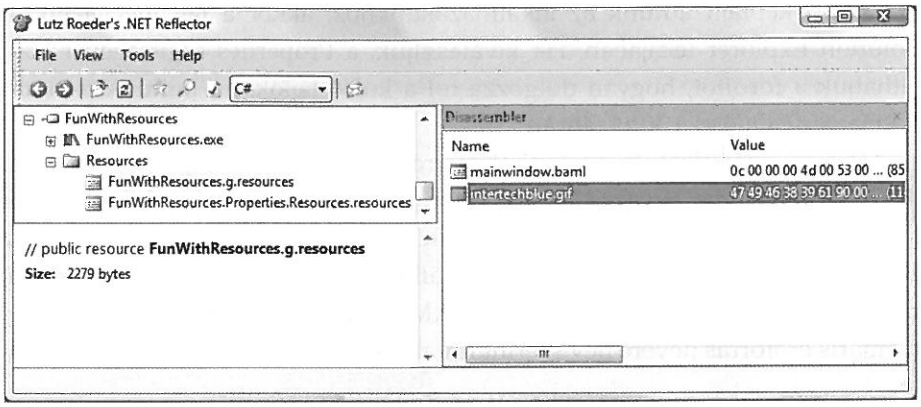
Ha lefordítjuk és futtatjuk a programunkat, láthatjuk, hogy a képünk kitölti a `<Grid>` elemünk első celláját.



30.11. ábra: *A bináris erőforrások csomagolásának lehetőségei*

Megjegyzés Ne feledjük, hogy a WPF-alkalmazások a Resource opció segítségével ágyaznak be erőforrásokat, amely WPF-alkalmazások esetén választható opció. A csábítónak tűnő Embedded Resource beállítást a Windows Forms alkalmazások használják.

Ha a lefordított alkalmazást a `reflector.exe` (lásd a 2. fejezetet) eszközbe töltjük, a 30.12. ábrán látható módon közvetlenül megtekinthetjük a beágyazott erőforrást.



30.12. ábra: *Beágyazott bináris erőforrás*

A Content Build Action

A kapcsolódó külső fájl Build Action elemét Resource helyett Content értékre is állíthatjuk. Ebben az esetben a rendszer a fordítás során értesíti a szerelvényt a külső fájl relatív elérési útjáról, de nem foglalja a szerelvénybe a bináris adatokat. A beállítás segítségünkre lehet olyan alkalmazás telepítése során, amely időről időre lecserélt külső erőforrásokat tároló alkönyvtárat (vagy alkönyvtárakat) tartalmaz, illetve ha a külső erőforrás hálózati megosztási ponton helyezkedik el.

Ilyenkor a WPF-erőforráskezelőrendszer az egyszerű fájlnéven kívül további URI-kat (a külső szerelvénybe ágyazott erőforrások betöltéséhez szükséges szintaxist is beleértve) definiál. Ha az olyan helyi erőforrásokkal alkalmazható URI-formátumokat szeretnénk vizsgálni, amelyeket nem fordítunk le az aktuális szerelvénybe, ezzel kapcsolatban a .NET Framework 3.5 SDK dokumentáció „Pack URIs in WPF" című részében találunk részletes információkat.