Windows Formsszal ellentétben – a WPF kezdeti kiadása nem biztosít léptető-gomb-vezérlőelemet, amely lehetővé teszi a felhasználó számára, hogy a fel- és lefelé nyilak segítségével állítson be numerikus értéket. A `RepeatButton` funkcionalitásának köszönhetően viszonylag könnyedén elkészíthetünk egy léptető-gomb-vezérlőt XAML-ben.

Ennek bemutatásához hozzunk létre új Visual Studio WPF-alkalmazás projektet CustomSpinButtonApp néven. Cseréljük le a kezdeti `<Grid>` definíciót olyan `<StackPanel>` típussal, amely az alábbi markupot tartalmazza:

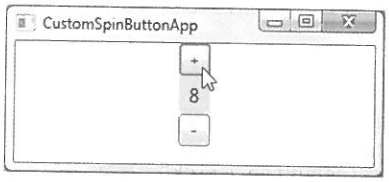
```
<Window x:Class="CustomSpinButtonApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="CustomSpinButtonApp" Height="300" Width="300">
  <StackPanel>
    <!-- A "Fel" gomb  -->
    <RepeatButton Height ="25" Width = "25"
                  Name ="repeatAddValueButton"
                  Delay ="200" Interval ="1"
                  Click ="repeatAddValueButton_Click"
                  Content = "+"/>

    <!-- Megjeleníti az aktuális értéket  -->
    <Label Name ="lblCurrentValue" Background ="LightGray"
           Height ="30" Width = "25"
           VerticalContentAlignment="Center"
           HorizontalContentAlignment="Center" FontSize="15"/>

    <!-- A "Le" gomb -->
    <RepeatButton Height ="25" Width = "25"
                  Name ="repeatRemoveValueButton"
                  Delay ="200" Interval ="1"
                  Click ="repeatRemoveValueButton_Click"
                  Content = "-"/>
  </StackPanel>
</Window>
```

Vegyük észre, hogy az egyes `RepeatButton` típusok egyedi eseménykezelővel hogyan kezelik a kattintási eseményt. Ezzel a következő C#-logikát írhatjuk le, hogy növeljük vagy csökkentsük a `<Label>` címkében megjelenített értéket (nyugodtan adjunk hozzá további logikát, ha szeretnénk rögzíteni a maximális és minimális értéket):

```
public partial class MainWindow : System.Windows.Window
{
  private int currValue = 0;
```

```
    public MainWindow()
    {
      InitializeComponent();
      lblCurrentValue.Content = currValue;
    }
    protected void repeatAddValueButton_Click(object sender,
                                              RoutedEventArgs e)
    {
      // Hozzáad egyet az aktuális értékhez, és megmutatja a címkében.
      currValue++;
      lblCurrentValue.Content = currValue;
    }

    protected void repeatRemoveValueButton_Click(object sender,
                                                 RoutedEventArgs e)
    {
      // Kivon egyet az aktuális értékből, és megmutatja a címkében.
      currValue--;
      lblCurrentValue.Content = currValue;
    }
  }
```

Ahogyan láthatjuk, amikor a felhasználó rákattint valamelyik RepeatButton gombra, annak megfelelően növeljük vagy csökkentjük a privát `currValue` értéket, és beállítjuk a `Label` típus `Content` tulajdonságát. A 29.6. ábra mutatja az egyedi léptetőgombunkat működés közben.



29.6. ábra: *Léptetőgomb készítése, kezdőpontként a RepeatButton használatával*

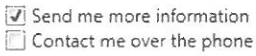
Forráskód A CustomSpinButtonApp kódfájlokat a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A jelölőnégyzetek és a rádiógombok használata

Ahogy korábban említettük, a CheckBox „az-egy" ToggleButton, „az-egy" ButtonBase, ami elég különösnek tűnhet, mivel egy gomb felhasználói felülete meglehetősen különbözik egy jelölőnégyzetétől. Azonban, ha a `CheckBox` tí-

pusra (mint a `Button` típusra) rákattinthatunk, reagál az egér- és billentyűzetbevitelre, valamint követi a WPF-tartalommodellt. A hasonlatosságoknak köszönhetően kiderül, hogy a `CheckBox` típus egyszerűen felülírja a `ToggleButton` különböző virtuális tagjait, hogy létrehozza a jelölőnégyzet megjelenését és működését (jusson eszünkbe, hogy a WPF fő mozgatórugója az, hogy elválasszuk egy vezérlőelem megjelenését annak funkcionalitásától). Tekintsük át a következő `<CheckBox>` deklarációkat, amelyek a 29.7. ábrán látható kimenetet eredményezik:

```
<StackPanel>
  <!-- CheckBox típusok -->
  <CheckBox Name ="checkInfo" >Send me more information</CheckBox>
  <CheckBox Name ="checkPhoneContact" >Contact me over the
                  phone</CheckBox>
</StackPanel>
```



29.7. ábra: *Egyszerű CheckBox típusok*

A RadioButton „az-egy" másik `ToggleButton` típus. A `CheckBox` típussal ellentétben azonban rendelkezik azzal a természetes képességgel, amely biztosítja, hogy az ugyanabban a tárolóban (`StackPanel`, `Grid` stb.) lévő összes `RadioButton` típus kölcsönösen kizárólagos, és nem igényel további munkát a részünkről. Vizsgáljuk meg a következőt:

```
<StackPanel>
  <!-- RadioButton típusok zeneválasztáshoz -->
  <Label FontSize = "15" Content = "Select Your Music Media"/>
  <RadioButton>CD Player</RadioButton>
  <RadioButton>MP3 Player</RadioButton>
  <RadioButton>8-Track</RadioButton>

  <!-- RadioButton típusok színválasztáshoz -->
  <Label FontSize = "15" Content = "Select Your Color Choice"/>
  <RadioButton>Red</RadioButton>
  <RadioButton>Green</RadioButton>
  <RadioButton>Blue</RadioButton>
</StackPanel>
```

Ha tesztelnénk ezt a XMAL-t, azt tapasztalnánk, hogy csak egyet választhatunk a hat lehetőség közül, ami valószínűleg nem áll szándékunkban, mivel két különböző csoport van (rádió és szín lehetőségek).

Logikai csoportosítások létrehozása

Ha egyetlen tárolót szeretnénk több `RadioButton` típussal, amelyek különálló fizikai csoportosításként viselkednek, ezt a `GroupName` tulajdonság beállításával tehetjük a `RadioButton` típus nyitó elemén:

```
<StackPanel>
  <!-- A Zene csoport  -->
  <Label FontSize = "15" Content = "Select Your Music Media"/>
  <RadioButton GroupName = "Music" >CD Player</RadioButton>
  <RadioButton GroupName = "Music" >MP3 Player</RadioButton>
  <RadioButton GroupName = "Music" >8-Track</RadioButton>

  <!--A Szín csoport (tetszőlegesen választható ebben a példában,
      lásd a megjegyzést lent) -->
  <Label FontSize = "15" Content = "Select Your Color Choice"/>
  <RadioButton GroupName = "Color">Red</RadioButton>
  <RadioButton GroupName = "Color">Green</RadioButton>
  <RadioButton GroupName = "Color">Blue</RadioButton>
</StackPanel>
```

Ezáltal egymástól függetlenül beállíthatjuk az egyes logikai csoportosításokat, még akkor is, ha azok ugyanabban a fizikai tárolóban vannak.

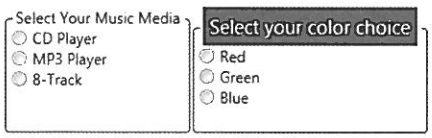
Megjegyzés Alapértelmezés szerint a `GroupName` értékkel nem rendelkező tárolóban lévő összes `RadioButton` egyetlen fizikai csoportként működik. Ezért, az előző példában a színközpontú gombok kölcsönösen kizáróak, még az elhagyott `GroupName` értékkel is, a `Zene` csoport jelenlétének köszönhetően.

A kapcsolódó elemek összeállítása GroupBox típusokba

Amikor rádiógombok vagy jelölőnégyzetek gyűjteményét tervezzük, megszokott dolog vizuális tárolóval körülvenni őket annak jelzésére, hogy csoportként viselkednek. Az általános módszer ehhez a `GroupBox` vezérlőelem használata. Mivel a `Header` tulajdonság alapvetően `System.Object` típussal dolgozik, hozzárendelhetünk bármilyen objektumot (egyszerű sztringet, színes téglalapot, gombot stb.), hogy fejlécként működjön. Vizsgáljuk meg a következő két `GroupBox` deklarációt, amelyek különféle módokon foglalják egységbe az előző `RadioButton` típusokat:

```
<StackPanel>
  <GroupBox Header = "Select Your Music Media" BorderBrush ="Black">
    <StackPanel>
      <RadioButton GroupName = "Music" >CD Player</RadioButton>
      <RadioButton GroupName = "Music" >MP3 Player</RadioButton>
      <RadioButton GroupName = "Music" >8-Track</RadioButton>
    </StackPanel>
  </GroupBox>

  <GroupBox BorderBrush ="Black">
    <GroupBox.Header>
      <Label Background = "Blue" Foreground = "White"
             FontSize = "15" Content = "Select your color choice"/>
    </GroupBox.Header>
    <StackPanel>
      <RadioButton>Red</RadioButton>
      <RadioButton>Green</RadioButton>
      <RadioButton>Blue</RadioButton>
    </StackPanel>
  </GroupBox>
</StackPanel>
```

A kimenet a 29.8. ábrán látható.



29.8. ábra: *RadioButton típusokat keretező GroupBox típusok*

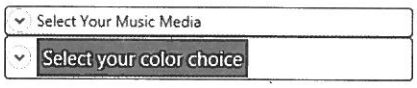
Kapcsolódó elemek bővítő típusokba csoportosítása

A szokásos `GroupBox` típus mellett a WPF rendelkezik olyan új felhasználói felület elemmel, amely csoportosítja olyan felhasználóifelület-elemek gyűjteményét, amelyek egy kapcsoló segítségével elrejthetők, illetve megmutathatók. Ez az elem, az Expander típus lehetővé teszi, hogy az `ExpandDirection` tulajdonság segítségével meghatározzuk az elemek megjelenítésének irányát (fel, le, balra vagy jobbra). Tekintsük meg az alábbi XAML-leírást (amely alapvetően `<GroupBox>` típusról `<Expander>` típusra módosul):

```
<StackPanel>
  <Expander Header = "Select Your Music Media" BorderBrush ="Black">
    <StackPanel>
      <RadioButton GroupName = "Music" >CD Player</RadioButton>
      <RadioButton GroupName = "Music" >MP3 Player</RadioButton>
```

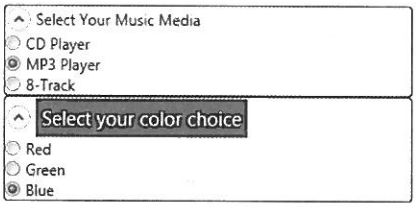
```
        <RadioButton GroupName = "Music" >8-Track</RadioButton>
      </StackPanel>
    </Expander>

    <Expander BorderBrush ="Black">
      <Expander.Header>
        <Label Background = "Blue" Foreground = "White"
               FontSize = "15" Content = "Select your color choice"/>
      </Expander.Header>
      <StackPanel>
        <RadioButton>Red</RadioButton>
        <RadioButton>Green</RadioButton>
        <RadioButton>Blue</RadioButton>
      </StackPanel>
    </Expander >
  </StackPanel>
```

A 29.9. ábra mutatja az egyes Expandereket összecsukott állapotban.



29.9. ábra: *Összecsukott Expanderek*

A 29.10. ábra mutatja az egyes Expandereket kinyitott állapotban.



29.10. ábra: *Kiterjesztett Expanderek*

Forráskód A CheckRadioGroup.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A ListBox és a ComboBox típusok használata

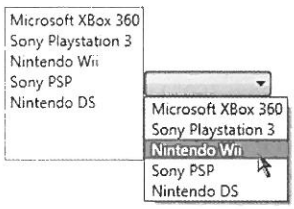
Ahogyan azt remélhettük, a WPF biztosít olyan típusokat, amelyek választható elemek csoportját tartalmazzák – ilyen a `ListBox` és a `ComboBox`, ezek mindegyike az `ItemsControl` absztrakt ősosztályból származik. A legfonto-

sabb, hogy ez a szülőosztály definiálja az `Items` nevű tulajdonságot, amely a részelemeket tároló, erősen típusos `ItemCollection` objektumot adja vissza. Az `ItemCollection` típus a `System.Object` típusokkal dolgozik, ezért gyakorlatilag bármit tartalmazhat. Ha markup révén szeretnénk egyszerű szöveges adatokkal feltölteni egy `ItemsControl`-leszármazott típust, akkor ezt `<ListBoxItem>` típusok készletének használatával tehetjük meg. Példaként nézzük a következő XAML-kódot:

```
<!-- Egyszerű listamező -->
<ListBox Name = "lstVideoGameConsoles">
  <ListBoxItem>Microsoft XBox 360</ListBoxItem>
  <ListBoxItem>Sony Playstation 3</ListBoxItem>
  <ListBoxItem>Nintendo Wii</ListBoxItem>
  <ListBoxItem>Sony PSP</ListBoxItem>
  <ListBoxItem>Nintendo DS</ListBoxItem>
  </ListBox>

<!-- Egyszerű legördülő listadoboz -->
<ComboBox Name = "comboVideoGameConsoles">
  <ListBoxItem>Microsoft XBox 360</ListBoxItem>
  <ListBoxItem>Sony Playstation 3</ListBoxItem>
  <ListBoxItem>Nintendo Wii</ListBoxItem>
  <ListBoxItem>Sony PSP</ListBoxItem>
  <ListBoxItem>Nintendo DS</ListBoxItem>
</ComboBox>
```

Megjegyzés A `ComboBox` típusokat feltölthetjük `<ListBoxItem>` elemek helyett `<ComboBokszItem>` elemek segítségével. Ezáltal hozzáférhetünk az `IsHighlighted` tulajdonsághoz, amelyet a `ListBoxItem` típus nem használ.

Nem meglepő módon a 29.11. ábrán látható renderelést látjuk.



29.11. ábra: *Egyszerű ListBox és ComboBox*

Lista-vezérlőelemek feltöltése programozott módon

A lista-vezérlőelemben lévő adatok gyakran csak futásidőben ismertek; lehet például, hogy a listamező elemeit egy adatbázis olvasásból visszakapott értékek, WFC-szolgáltatás meghívása vagy külső fájl beolvasása alapján kell feltöltenünk. Amikor programozott módon kell `ListBox` vagy `ComboBox` elemet feltöltenünk, egyszerűen használjuk az `ItemCollection` típus tagjait (`Add()`, `Remove()` stb.). Tételezzük fel, hogy új `ListControls` nevű Visual Studio 2008 WPF-alkalmazás projektünk van. A `lstVideoGameConsole` típus előző XAML-deklarációját a következők szerint lehet XAML-ben definiálni:

```
<Window x:Class="ListControls.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="ListControls" Height="300" Width="300" >
  <StackPanel>
    <!-- Ezt kód segítségével töltjük fel -->
    <ListBox Name = "lstVideoGameConsoles">
    </ListBox>
  </StackPanel>
</Window>
```

és a kapcsolódó kódfájlban az alábbiak szerint tölthetjük fel:

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

  private void FillListBox()
  {
    // Elemek hozzáadása a listamezőhöz.
    lstVideoGameConsoles.Items.Add("Microsoft XBox 360");
    lstVideoGameConsoles.Items.Add("Sony Playstation 3");
    lstVideoGameConsoles.Items.Add("Nintendo Wii");
    lstVideoGameConsoles.Items.Add("Sony PSP");
    lstVideoGameConsoles.Items.Add("Nintendo DS");
  }
}
```

Az egyik dolog, amely különösnek tűnhet a ListBox XAML-leírásában, hogy a `<ListBoxItem>` típusokat használtuk az elemek feltöltéséhez; itt azonban sztringtípusokat használtunk az `Add()` metódus meghívása során. Ennek rö-

vid magyarázata az, hogy a XMAL használatakor a `<ListBoxItem>` típusok kényelmesebbek abból a szempontból, hogy azokat a http://schemas.microsoft.com/winfx/2006/xaml/presentation XML-névtérben definiáltuk, ezért rendelkezünk a közvetlen referenciájukkal.

A háttérben a rendszer a `ToString()` metódust hívja minden egyes `<ListBoxItem>` típuson, így a végeredmény azonos. Ha `System.String` névtér használatával töltenénk fel a `ListBox` (vagy `ComboBox`) típust XAML-ben, akkor meg kellene határoznunk egy új XML-névteret, hogy behozzuk az `mscorlib.dll` fájlt (további részletekért lásd a 28. fejezetet):

```
<StackPanel xmlns:CorLib = "clr-namespace:System;assembly=mscorlib">
  <ListBox Name = "lstVideoGameConsoles">
    <CorLib:String>Microsoft XBox 360</CorLib:String>
    <CorLib:String>Sony Playstation 3</CorLib:String>
    <CorLib:String>Nintendo Wii</CorLib:String>
    <CorLib:String>Sony PSP</CorLib:String>
    <CorLib:String>Nintendo DS</CorLib:String>
  </ListBox>
</StackPanel>
```

Ellenben, ha szeretnénk, programozott módon feltölthetünk egy `ItemsControl`-leszármazott típust erősen típusos `ListBoxItem` objektumok segítségével; a jelenlegi példában azonban ezzel nem nyerünk semmit, hanem valójában csak további munkát generálunk magunknak (mivel a `ListBoxItem` nem rendelkezik konstruktorral a `Content` tulajdonság beállításához!).

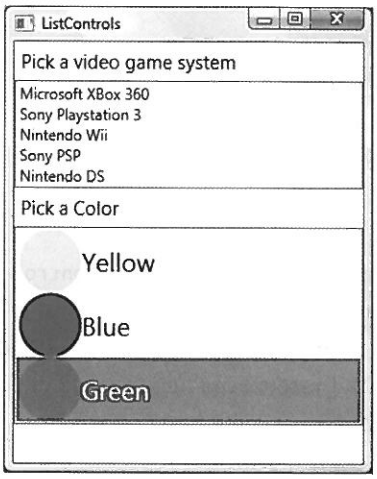
Tetszőleges tartalom hozzáadása

Mivel mind a `ListBox`, mind a `ComboBox` származtatási láncában megtalálható a `ContentControl` alaposztály, ezért azok egy egyszerű sztringnél jóval bonyolultabb adatokat is tartalmazhatnak. Tekintsük át a következő `ComboBox` típust, amely kétdimenziós grafikus objektumokat és egy leíró címkét tartalmazó `<StackPanel>` tárolókat foglal magában:

```
<StackPanel>
  <!-- Egy ListBox tartalommal -->
  <ListBox Name = "lstColors">
    <StackPanel Orientation ="Horizontal">
      <Ellipse Fill ="Yellow" Height ="50" Width ="50"/>
      <Label FontSize ="20" HorizontalAlignment="Center"
             VerticalAlignment="Center">Yellow</Label>
    </StackPanel>
    <StackPanel Orientation ="Horizontal">
```

```
        <Ellipse Fill ="Blue" Height ="50" Width ="50"/>
        <Label FontSize ="20" HorizontalAlignment="Center"
               VerticalAlignment="Center">Blue</Label>
      </StackPanel>
      <StackPanel Orientation ="Horizontal">
        <Ellipse Fill ="Green" Height ="50" Width ="50"/>
        <Label FontSize ="20" HorizontalAlignment="Center"
               VerticalAlignment="Center">Green</Label>
      </StackPanel>
    </ListBox>
  </StackPanel>
```

A 29.12. ábra mutatja az aktuális lista típusaink kimenetét.



29.12. ábra: *ItemsControl-leszármazott típusok tartalmazhatnak bármilyen jellegű tartalmat*

Az aktuális kiválasztás meghatározása

A `ListBox` vagy a `ComboBox` típus feltöltése után a következő nyilvánvaló kérdés az, hogy futásidőben hogyan lehet eldönteni, melyik elemet választotta ki a felhasználó. Mint látni fogjuk, erre három módszer létezik. Ha a kiválasztott elem numerikus indexének a kiderítése érdekel bennünket, akkor használhatjuk a `SelectedIndex` tulajdonságot (amely nulla alapú; a `-1` érték azt mutatja, hogy nincs kijelölés). Ha a listában a kiválasztott objektumot szeretnénk megszerezni, arra a `SelectedItem` tulajdonság való. Végül, a `SelectedValue` lehetővé teszi, hogy a kiválasztott objektum értékét megkapjuk (többnyire a `ToString()` metódus meghívásával).

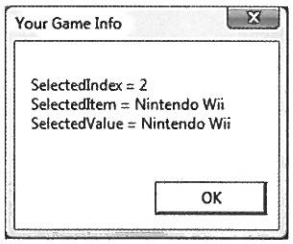
Nem hangzik bonyolultnak, igaz? Most, hogy teszteljük az egyes tulajdonságok viselkedését, tételezzük fel, hogy definiáltunk két új `Button` típust az aktuális ablak számára, amelyek egyaránt kezelik a kattintási eseményt:

```
<!-- Gombok a kiválasztott elem megszerzéséhez -->
<Button Name ="btnGetGameSystem" Click ="btnGetGameSystem_Click">
  Get Video Game System
</Button>
<Button Name ="btnGetColor" Click ="btnGetColor_Click">
  Get Color
</Button>
```

A `btnGetGameSystem` esetében a kattintási esemény kezelője megszerzi a `lstVideoGameConsoles` objektum `SelectedIndex`, `SelectedItem` és `SelectedValue` tulajdonságainak értékeit, és megjeleníti azokat egy üzenetdobozban:

```
protected void btnGetGameSystem_Click(object sender,
                                      RoutedEventArgs args)
{
  string data = string.Empty;
  data += string.Format("SelectedIndex = {0}\n",
          lstVideoGameConsoles.SelectedIndex);
  data += string.Format("SelectedItem = {0}\n",
          lstVideoGameConsoles.SelectedItem);
  data += string.Format("SelectedValue = {0}\n",
          lstVideoGameConsoles.SelectedValue);
  MessageBox.Show(data, "Your Game Info");
}
```

Ha a „Nintendo Wii"-t választanánk a játékkonzolok listájából, és a kapcsolódó gombra kattintanánk, akkor a 29.13. ábrán látható szövegdobozt kapnánk.

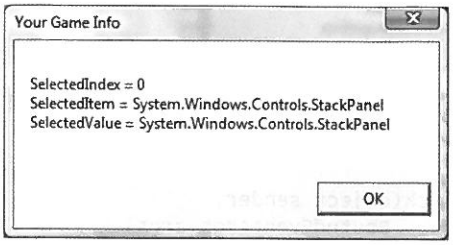


29.13. ábra: *Megtaláltuk a kiválasztott szringet*

Azonban, mi a helyzet a kiválasztott szín megszerzésével?

Az aktuális kiválasztás meghatározása a beágyazott tartalom esetében

Tételezzük fel, hogy a `btnGetColor` `Button` típus kattintási eseményének kezelője implementálja a `btnGetColor_Click()` metódust, hogy kiírja a `lstColors` `ListBox` objektum aktuális kijelölését, indexét és értékét. Ha most kiválasztanánk az első elemet a lstColors listamezőből (és a hozzátartozó gombra kattintanánk), meglepetésünkre a 29.14. ábrán látható kimenettel szembesülnénk.



29.14. ábra: *Megtaláltuk a kiválasztott... StackPanelt?*

Ennek a kimenetnek az oka az, hogy a `lstColors` objektum három `StackPanel` objektumot tart fenn, amelyek mindegyike rendelkezik beágyazott tartalommal. Ezért a `SelectedItem` és a `SelectedValue` tulajdonságok egyszerűen meghívják a `ToString()` metódust a `StackPanel` típuson, amely visszaadja teljesen meghatározott nevét.

Noha a `SelectedIndex` tulajdonságból visszaadott numerikus érték alapján könnyen kitalálhatnánk, hogy a felhasználó melyik elemet választotta, egy másik megközelítés az, hogy leásunk a `StackPanel` gyermekgyűjteményébe, hogy megkapjuk a `Label` típus `Content` értékét a `StackPanel` belsőleg fenntartott `Children` gyűjteményének segítségével:

```
protected void btnGetColor_Clicked(object sender,
                                   RoutedEventArgs args)
{
  // A Content érték megszerzése a StackPanel elemben
  // a kiválasztott Label típusban.
  StackPanel selectedStack =
             (StackPanel)lstColors.Items[lstColors.SelectedIndex];
  string color =
         ((Label)(selectedStack.Children[1])).Content.ToString();
  string data = string.Empty;
  data += string.Format("SelectedIndex = {0}\n",
          lstColors.SelectedIndex);
  data += string.Format("Color = {0}", color);
  MessageBox.Show(data, "Your Game Info");
}
```

Noha ez a megoldás megteszi, mégis elég kényes, mivel kódolt pozíciók vannak a `StackPanel` tárolóban (a második gyermek a `Label`), és számos kasztolási műveletet kell végrehajtanunk. Másik alternatíva minden `StackPanel` Tag tulajdonságának beállítása, amelyet a `FrameworkElement` alaposztályban definiálunk:

```
<ListBox Name = "lstColors">
  <StackPanel Orientation ="Horizontal" Tag ="Yellow">
  ...
  </StackPanel>
  <StackPanel Orientation ="Horizontal" Tag ="Blue">
  ...
  </StackPanel>
  <StackPanel Orientation ="Horizontal" Tag ="Green">
  ...
  </StackPanel>
</ListBox>
```

Ezen megközelítés használatával a kódunk jelentősen letisztul, mivel programozott módon kiszedhetjük a Tag tulajdonsághoz rendelt értéket, a következők szerint:

```
protected void btnGetColor_Clicked(object sender,
                                   RoutedEventArgs args)
{
  string data = string.Empty;
  data += string.Format("SelectedIndex = {0}\n",
          lstColors.SelectedIndex);
  data += string.Format("SelectedItem = {0}\n",
          lstColors.SelectedItem);
  data += string.Format("SelectedValue = {0}",
          (lstColors.Items[lstColors.SelectedIndex] as
          StackPanel).Tag);
  MessageBox.Show(data, "Your Color Info");
}
```

Noha ez a megközelítés egy kicsit tisztább, mint az első kísérletünk, vannak más módok is, ahol *adatsablonok* segítségével érhetjük el egy összetett vezérlőelem értékét. Ehhez meg kell értenünk a WPF adatkötési motor működését, amelyet a fejezet végén vizsgálunk meg.

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

A többsoros szövegbeviteli mezők használata

A WPF több olyan felhasználóifelület-elemet biztosít, amelyek lehetővé teszik a szövegalapú bevitelek összegyűjtését. A két legegyszerűbb típus a `TextBox` és a `PasswordBox`, amelyeket most a `TextControls` nevű, új Visual Studio 2008 WPF alkalmazás használatával vizsgálunk meg.

A TextBox típus használata

A múltban használt más `TextBox` típusokhoz hasonlóan a WPF `TextBox` típusát beállíthatjuk, hogy egy sornyi szöveget tartalmazzon (ez az alapértelmezett beállítás), vagy több sort, ha az `AcceptReturn` tulajdonság értéke `true`. A `TextBox`on belül az adatokat mindig karakteradatokként kezeljük, ezért a „tartalom" mindig olyan sztringtípus, amelyet a Text tulajdonság segítségével beállíthatunk és lekérdezhetünk:

```
<TextBox Name ="txtData" Text = "Hello!" BorderBrush ="Blue"
         Width ="100"/>
```

A WPF `TextBox` típusának egyik nagyon egyedi jellemzője, hogy beépített képessége révén ellenőrizheti a bevitt adatok helyesírását, ha a `SpellCheck.IsEnabled` tulajdonság értéke `true`. Ilyen esetekben látni fogjuk, hogy - a Microsoft Office-hoz hasonlóan - a helytelenül írt szavakat az alkalmazás vörös hullámos vonallal aláhúzza. Ennél még jobb, hogy az alapul szolgáló programozási modell segítségével hozzáférhetünk a helyesírásellenőrző-motorhoz, ezáltal megkaphatjuk a helytelenül írt szavakra vonatkozó javaslatok listáját.

Az alábbiak szerint módosítsuk az aktuális ablak XAML-definícióját, hogy használhassuk a `Label`, a `TextBox` és a `Button` típusokat (figyeljük meg, hogy a `TextBox` többsoros szövegeket támogat, és engedélyezi a helyesírás-ellenőrzést):

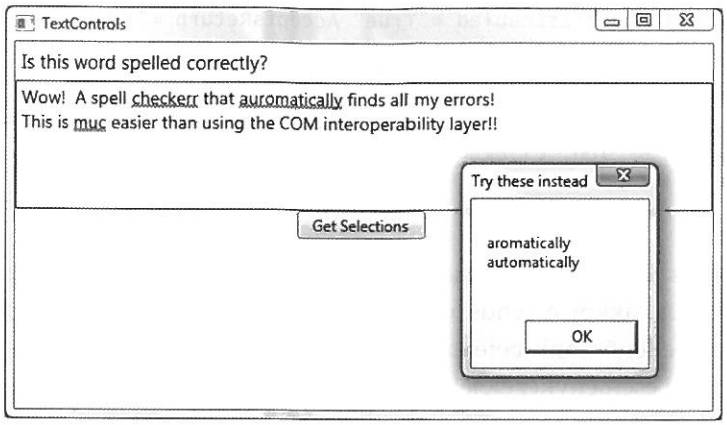
```
<Window x:Class="TextControls.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="TextControls" Height="204" Width="292" >
  <StackPanel>
    <Label FontSize ="15">Is this word spelled correctly?</Label>
```

```
    <TextBox SpellCheck.IsEnabled ="True" AcceptsReturn ="True"
             Name ="txtData" FontSize ="12"
             BorderBrush ="Blue" Height ="100">
    </TextBox>
    <Button Name ="btnOK" Content ="Get Selections"
            Width = "100" Click ="btnOK_Click"/>
  </StackPanel>
</Window>
```

Ilyen funkcionalitással már látjuk majd, hogy amikor helytelenül írunk be szavakat a `TextBox`-ba, akkor a rendszer megjelöli a helyesírási hibákat. Az egyszerű helyesírás-ellenőrzőnk befejezéséhez a következőképpen módosítsuk a gomb kattintási eseménykezelőjét:

```
protected void btnOK_Click(object sender, RoutedEventArgs args)
{
  string spellingHints = string.Empty;

  // Próbáljunk helyesírási hibát keresni a kurzor aktuális
  // helyénél.
  SpellingError error =
               txtData.GetSpellingError(txtData.CaretIndex);
  if (error != null)
  {
    // Készítsük el a helyesírási javaslatok sztringjét.
    foreach (string s in error.Suggestions)
    {
      spellingHints += string.Format("{0}\n", s);
    }

    // Mutassuk meg a javaslatokat!
    MessageBox.Show(spellingHints, "Try these instead");
  }
}
```

A kód meglehetősen egyszerű. Csak megkeressük a kurzor jelenlegi helyét a szövegdobozban a `CaretIndex` tulajdonság használatával, hogy kinyerjünk egy `SpellingError` objektumot. Ha valami hiba van a mondott helyen (vagyis az érték nem `null`), akkor végiglépkedünk a javaslatok listáján a stílszerűen elnevezett `Suggestions` tulajdonság révén. Végül megjelenítjük a lehetőségeket egy egyszerű `MessageBox.Show()` kérés használatával. A 29.15. ábra lehetséges tesztfutást mutat, amikor a kurzor a helytelenül írt „auromatically" szón áll.



29.15. ábra: *Egy egyedi helyesírás-ellenőrző!*

A PasswordBox típus használata

A `PasswordBox` típus – nem meglepően – egy olyan biztonságos helyet kínál, ahova érzékeny szöveges adatokat írhatunk be. Alapértelmezés szerint a jelszó karakterkör típusú, ez azonban megváltoztatható a `PasswordChar` tulajdonság használatával. A végfelhasználó által beírt érték megszerzéséhez egyszerűen ellenőrizzük a `Password` tulajdonságot. Módosítsuk a jelenlegi helyesírás-ellenőrző alkalmazásunkat a helyes jelszó kérésével, hogy láthassuk a helyesírási javaslatok listáját. Először módosítsuk a már meglévő `<StackPanel>` tárolót egy beágyazott `<StackPanel>` típussal, amely vízszintesen elhelyezi a meglévő `<Button>` típus mellé a `PasswordBox` mezőt:

```
<StackPanel>
  <Label FontSize ="15">Is this word spelled correctly?</Label>
  <TextBox SpellCheck.IsEnabled ="True" AcceptsReturn ="True"
           Name ="txtFavoriteColor" FontSize ="14"
           BorderBrush ="Blue" Height ="100">
  </TextBox>
  <StackPanel Orientation ="Horizontal">
    <PasswordBox Name ="pwdText" BorderBrush ="Black"
                 Width ="100"></PasswordBox>
    <Button Name ="btnOK" Content ="Get Selections"
            Width = "100" Click ="btnOK_Click"/>
  </StackPanel>
</StackPanel>
```

Most módosítsuk úgy a jelenlegi `Button` kattintási eseménykezelőjét, hogy meghívja a `CheckPassword()` segédfüggvényt, amely teszteli a kódolt sztringet. Győződjünk meg arról, hogy a javaslatok csak akkor jelenhessenek meg, ha a jelszó ellenőrzése sikeres volt. Íme a releváns módosítások:

```
public partial class MainWindow : System.Windows.Window
{
  ...
  protected void btnOK_Click(object sender, RoutedEventArgs args)
  {
    if (CheckPassword())
    {
      // Ugyanaz a helyesírás-ellenőrző logika, mint korábban...
    }
    else
      MessageBox.Show("Security error!!");
  }

  private bool CheckPassword()
 {
    if (pwdText.Password == "Chucky")
    return true;
  else
    return false;
  }
}
```

A `TextBox` és a `PasswordBox` mellett ne feledjük, hogy ha egy olyan alkalmazást készítünk, amely rendelkezik olyan szövegmezővel, amely bármilyen típusú tartalmat (grafikus renderelést, szöveget stb.) magában foglalhat, a WPF biztosítja számunkra a `RichTextBox` típust. Emellett, ha kell a teljesítmény egy különösen szövegintenzív alkalmazás készítéséhez, a WPF biztosít egy teljes dokumentumprezentációs API-t, amelyet elsősorban a `System.Windows.Documents` névtér képvisel.

Itt találunk olyan típusokat, amelyek lehetővé teszik *folyószöveg-dokumentumok* (flow document) készítését, és ezek segítségével programozott módon ábrázolhatunk (XAML- vagy C#-kódban) bekezdéseket, kapcsolódó szövegblokk-részeket, feljegyzéseket, jegyzeteket, táblázatokat és egyéb gazdag dokumentumcentrikus típusokat. A könyv jelen kiadása nem tárgyalja a `RichTextBox` típust, illetve a folyószöveg-dokumentum API-t; ha érdekesnek bizonyul a téma, további részletekért lapozzuk fel a .NET Framework 3.5 SDK dokumentációját.

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

Ezzel befejeztük a WPF vezérlőelemkészlet áttekintését. A fejezet későbbi részében látni fogjuk, hogyan készíthetünk menürendszereket, állapotsávokat és eszköztárakat. A következő feladatunk azonban annak áttekintése, hogyan rendezhetjük a felhasználói felület elemeit `Window` típuson belül tetszőleges számú paneltípus használatával.

A tartalomelrendezés kezelése panelek használatával

Egy valós WPF-alkalmazás mindenképpen tartalmaz jó néhány felhasználóifelület-elemet (beviteli vezérlőelemeket, grafikus tartalmat, menürendszereket, állapotsávokat stb.), amelyeket jól kell elrendezni a tároló ablakban. Emellett, ha egyszer elhelyeztük a felhasználói felület vezérlőit az új helyükre, biztosak akarunk lenni abban, hogy az elvártak szerint viselkednek, amikor a végfelhasználó átméretezi az ablakot vagy akár az ablak egy részét (osztott ablak esetében). Ahhoz, hogy biztosítsuk, a WPF-vezérlőelemek megtartsák pozíciójukat a hosztoló ablakban, több *paneltípus* áll rendelkezésünkre.

Ahogy az előző fejezetből emlékezhetünk rá, hogy amikor tartalmat helyezünk el olyan ablakban, amely nem használ paneleket, az hajszálpontosan a tároló közepére kerül. Tekintsük át az alábbi egyszerű ablakdeklarációt, amely egyetlen `Button` típust tartalmaz. Függetlenül attól, hogyan méretezzük át az ablakot, a felhasználóifelület-vezérlő mindig egyenlő távolságra jelenik meg az adott terület négy oldalától.

```
<!-- Ez a gomb mindig az ablak közepén van -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">
  <Button Name="btnOK" Height = "100" Width="80">OK</Button>
</Window>
```

Arra is emlékezzünk, hogy ha több elemet próbálunk elhelyezni közvetlenül egy `<Window>` típus hatókörén belül, akkor címkézési és/vagy fordítási idejű hibát kapunk. Ezeknek a hibáknak az az oka, hogy egy ablak (vagy ami azt illeti, a `ContentControl` bármelyik leszármazottja) csak egyetlen objektumot rendelhet a `Content` tulajdonságához:

```
<!-- Hiba! A Content tulajdonságot implicit módon többször
     állítottuk be! -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">

  <!-- Nyugtázva! A <Window> két közvetlen gyermekeleme! -->
  <Label Name="lblInstructions"
         Width="328" Height="27" FontSize="15">Enter
         Car Information</Label>
  <Button Name="btnOK" Height = "100" Width="80">OK</Button>
</Window>
```

Nyilvánvaló, hogy nem sok haszna van egy olyan ablaknak, amely csak egy elemet tartalmazhat. Ha arra van szükség, hogy egy ablak több elemet tartalmazzon, akkor azokat tetszőleges számú panelbe kell rendeznünk. A panel tartalmazza az ablakot képviselő felhasználóifelület-elemeket, amely után a panelt magát csak a `Content` tulajdonsághoz rendelt objektumként használjuk.

A WPF alapvető paneltípusai

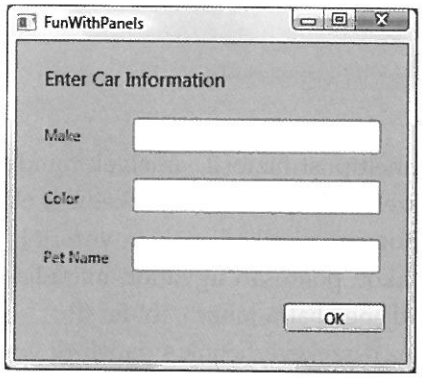
A `System.Windows.Controls` névtér több paneltípust biztosít, amelyek mindegyike azt szabályozza, hogy az egyes részelemek hogyan helyezkednek el. A panelek használatával megadhatjuk, hogyan viselkedjenek a vezérlők, amikor a végfelhasználó átméretezi az ablakot: pontosan ugyanott maradjanak, ahova tervezéskor kerültek; átrendeződjenek balra-jobbra, föl-le; stb.

Bonyolult felhasználói felület készítéséhez összekeverhetjük a panelbeli vezérlőelemeket (pl. egy `StackPanelt` tartalmazó `DockPanel`), ezáltal nagyfokú rugalmasságot és kezelhetőséget biztosíthatunk. Ezenfelül a paneltípusok együtt tudnak dolgozni más dokumentumközpontú vezérlőelemekkel (mint a `ViewBox`, a `TextBlock`, a `TextFlow` és a `Paragraph` típusok), hogy tovább lehessen finomítani a tartalom elrendezését egy adott panelen belül. A 29.3. táblázat bemutatja néhány gyakran használt WPF panelbeli vezérlőelem szerepét.

Panelbeli vezérlőelem	Jelentés
`Canvas`	„Klasszikus" mód a tartalom elhelyezésére. Az elemek pontosan ugyanott maradnak, ahová a tervezéskor tettük őket.
`DockPanel`	Zárolja a tartalmat a panel megadott oldalára (`Top`, `Bottom`, `Left` vagy `Right`).
`Grid`	Cellák sorozatába rendezi a tartalmat, táblázatos rácsformában.
`StackPanel`	Függőlegesen vagy vízszintesen egymás után helyezi a tartalmat, ahogy azt az `Orientation` tulajdonság megkívánja.
`WrapPanel`	Balról jobbra pozícionálja a tartalmat, új sorba tördelve azt a tároló doboz szélénél. A további rendezés folyamatosan történik felülről lefelé vagy jobbról balra, az `Orientation` tulajdonság értékétől függően.

29.3. táblázat: *Alapvető WPF panelbeli vezérlőelemek*

A következő részben bemutatjuk ezeknek a gyakran használt paneltípusoknak a használatát, elkészítjük a 29.16. ábrán látható grafikus felületet különböző paneleken belül, és megvizsgáljuk, hogyan változik az elhelyezés, ha az ablakot átméretezzük.



29.16. ábra: *A felhasználói felület megcélzott elrendezése*

Tartalom elhelyezése a Canvas paneleken belül

Vitathatatlanul a legegyszerűbb panel a Canvas. A Canvas az a panel, amelyben leginkább otthon érezzük magunkat, ugyanis egy Windows Forms alkalmazás alapértelmezett elrendezését utánozza. A Canvas panel lehetővé teszi a

Windows Formsszal ellentétben – a WPF kezdeti kiadása nem biztosít léptetőgomb-vezérlőelemet, amely lehetővé teszi a felhasználó számára, hogy a fel- és lefelé nyilak segítségével állítson be numerikus értéket. A `RepeatButton` funkcionalitásának köszönhetően viszonylag könnyedén elkészíthetünk egy léptetőgomb-vezérlőt XAML-ben.

Ennek bemutatásához hozzunk létre új Visual Studio WPF-alkalmazás projektet CustomSpinButtonApp néven. Cseréljük le a kezdeti `<Grid>` definíciót olyan `<StackPanel>` típussal, amely az alábbi markupot tartalmazza:

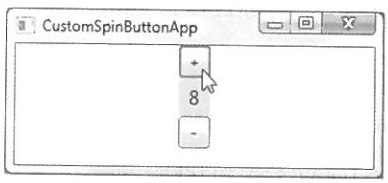
```
<Window x:Class="CustomSpinButtonApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="CustomSpinButtonApp" Height="300" Width="300">
  <StackPanel>
    <!-- A "Fel" gomb  -->
    <RepeatButton Height ="25" Width = "25"
                  Name ="repeatAddValueButton"
                  Delay ="200" Interval ="1"
                  Click ="repeatAddValueButton_Click"
                  Content = "+"/>

    <!-- Megjeleníti az aktuális értéket  -->
    <Label Name ="lblCurrentValue" Background ="LightGray"
           Height ="30" Width = "25"
           VerticalContentAlignment="Center"
           HorizontalContentAlignment="Center" FontSize="15"/>

    <!-- A "Le" gomb -->
    <RepeatButton Height ="25" Width = "25"
                  Name ="repeatRemoveValueButton"
                  Delay ="200" Interval ="1"
                  Click ="repeatRemoveValueButton_Click"
                  Content = "-"/>
  </StackPanel>
</Window>
```

Vegyük észre, hogy az egyes `RepeatButton` típusok egyedi eseménykezelővel hogyan kezelik a kattintási eseményt. Ezzel a következő C#-logikát írhatjuk le, hogy növeljük vagy csökkentsük a `<Label>` címkében megjelenített értéket (nyugodtan adjunk hozzá további logikát, ha szeretnénk rögzíteni a maximális és minimális értéket):

```
public partial class MainWindow : System.Windows.Window
{
  private int currValue = 0;
```

```
    public MainWindow()
    {
      InitializeComponent();
      lblCurrentValue.Content = currValue;
    }
    protected void repeatAddValueButton_Click(object sender,
                                              RoutedEventArgs e)
    {
      // Hozzáad egyet az aktuális értékhez, és megmutatja a címkében.
      currValue++;
      lblCurrentValue.Content = currValue;
    }

    protected void repeatRemoveValueButton_Click(object sender,
                                                 RoutedEventArgs e)
    {
      // Kivon egyet az aktuális értékből, és megmutatja a címkében.
      currValue--;
      lblCurrentValue.Content = currValue;
    }
}
```

Ahogyan láthatjuk, amikor a felhasználó rákattint valamelyik RepeatButton gombra, annak megfelelően növeljük vagy csökkentjük a privát `currValue` értéket, és beállítjuk a `Label` típus `Content` tulajdonságát. A 29.6. ábra mutatja az egyedi léptetőgombunkat működés közben.



29.6. ábra: *Léptetőgomb készítése, kezdőpontként a RepeatButton használatával*

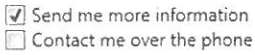
Forráskód A CustomSpinButtonApp kódfájlokat a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A jelölőnégyzetek és a rádiógombok használata

Ahogy korábban említettük, a CheckBox „az-egy" ToggleButton, „az-egy" ButtonBase, ami elég különösnek tűnhet, mivel egy gomb felhasználói felülete meglehetősen különbözik egy jelölőnégyzetétől. Azonban, ha a `CheckBox` tí-

pusra (mint a `Button` típusra) rákattinthatunk, reagál az egér- és billentyűzetbevitelre, valamint követi a WPF-tartalommodellt. A hasonlatosságoknak köszönhetően kiderül, hogy a `CheckBox` típus egyszerűen felülírja a `ToggleButton` különböző virtuális tagjait, hogy létrehozza a jelölőnégyzet megjelenését és működését (jusson eszünkbe, hogy a WPF fő mozgatórugója az, hogy elválasszuk egy vezérlőelem megjelenését annak funkcionalitásától). Tekintsük át a következő `<CheckBox>` deklarációkat, amelyek a 29.7. ábrán látható kimenetet eredményezik:

```
<StackPanel>
  <!-- CheckBox típusok -->
  <CheckBox Name ="checkInfo" >Send me more information</CheckBox>
  <CheckBox Name ="checkPhoneContact" >Contact me over the
                  phone</CheckBox>
</StackPanel>
```



29.7. ábra: *Egyszerű CheckBox típusok*

A RadioButton „az-egy" másik `ToggleButton` típus. A `CheckBox` típussal ellentétben azonban rendelkezik azzal a természetes képességgel, amely biztosítja, hogy az ugyanabban a tárolóban (`StackPanel`, `Grid` stb.) lévő összes `RadioButton` típus kölcsönösen kizárólagos, és nem igényel további munkát a részünkről. Vizsgáljuk meg a következőt:

```
<StackPanel>
  <!-- RadioButton típusok zeneválasztáshoz -->
  <Label FontSize = "15" Content = "Select Your Music Media"/>
  <RadioButton>CD Player</RadioButton>
  <RadioButton>MP3 Player</RadioButton>
  <RadioButton>8-Track</RadioButton>

  <!-- RadioButton típusok színválasztáshoz -->
  <Label FontSize = "15" Content = "Select Your Color Choice"/>
  <RadioButton>Red</RadioButton>
  <RadioButton>Green</RadioButton>
  <RadioButton>Blue</RadioButton>
</StackPanel>
```

Ha tesztelnénk ezt a XMAL-t, azt tapasztalnánk, hogy csak egyet választhatunk a hat lehetőség közül, ami valószínűleg nem áll szándékunkban, mivel két különböző csoport van (rádió és szín lehetőségek).

Logikai csoportosítások létrehozása

Ha egyetlen tárolót szeretnénk több `RadioButton` típussal, amelyek különálló fizikai csoportosításként viselkednek, ezt a `GroupName` tulajdonság beállításával tehetjük a `RadioButton` típus nyitó elemén:

```
<StackPanel>
  <!-- A Zene csoport  -->
  <Label FontSize = "15" Content = "Select Your Music Media"/>
  <RadioButton GroupName = "Music" >CD Player</RadioButton>
  <RadioButton GroupName = "Music" >MP3 Player</RadioButton>
  <RadioButton GroupName = "Music" >8-Track</RadioButton>

  <!--A Szín csoport (tetszőlegesen választható ebben a példában,
      lásd a megjegyzést lent) -->
  <Label FontSize = "15" Content = "Select Your Color Choice"/>
  <RadioButton GroupName = "Color">Red</RadioButton>
  <RadioButton GroupName = "Color">Green</RadioButton>
  <RadioButton GroupName = "Color">Blue</RadioButton>
</StackPanel>
```

Ezáltal egymástól függetlenül beállíthatjuk az egyes logikai csoportosításokat, még akkor is, ha azok ugyanabban a fizikai tárolóban vannak.

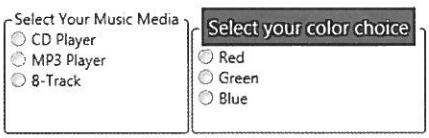
Megjegyzés Alapértelmezés szerint a `GroupName` értékkel nem rendelkező tárolóban lévő összes `RadioButton` egyetlen fizikai csoportként működik. Ezért, az előző példában a színközpontú gombok kölcsönösen kizáróak, még az elhagyott `GroupName` értékkel is, a `Zene` csoport jelenlétének köszönhetően.

A kapcsolódó elemek összeállítása GroupBox típusokba

Amikor rádiógombok vagy jelölőnégyzetek gyűjteményét tervezzük, megszokott dolog vizuális tárolóval körülvenni őket annak jelzésére, hogy csoportként viselkednek. Az általános módszer ehhez a `GroupBox` vezérlőelem használata. Mivel a `Header` tulajdonság alapvetően `System.Object` típussal dolgozik, hozzárendelhetünk bármilyen objektumot (egyszerű sztringet, színes téglalapot, gombot stb.), hogy fejlécként működjön. Vizsgáljuk meg a következő két `GroupBox` deklarációt, amelyek különféle módokon foglalják egységbe az előző `RadioButton` típusokat:

```
<StackPanel>
  <GroupBox Header = "Select Your Music Media" BorderBrush ="Black">
    <StackPanel>
      <RadioButton GroupName = "Music" >CD Player</RadioButton>
      <RadioButton GroupName = "Music" >MP3 Player</RadioButton>
      <RadioButton GroupName = "Music" >8-Track</RadioButton>
    </StackPanel>
  </GroupBox>

  <GroupBox BorderBrush ="Black">
    <GroupBox.Header>
      <Label Background = "Blue" Foreground = "White"
             FontSize = "15" Content = "Select your color choice"/>
    </GroupBox.Header>
    <StackPanel>
      <RadioButton>Red</RadioButton>
      <RadioButton>Green</RadioButton>
      <RadioButton>Blue</RadioButton>
    </StackPanel>
  </GroupBox>
</StackPanel>
```

A kimenet a 29.8. ábrán látható.



29.8. ábra: *RadioButton típusokat keretező GroupBox típusok*

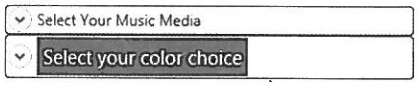
Kapcsolódó elemek bővítő típusokba csoportosítása

A szokásos `GroupBox` típus mellett a WPF rendelkezik olyan új felhasználói felület elemmel, amely csoportosítja olyan felhasználóifelület-elemek gyűjteményét, amelyek egy kapcsoló segítségével elrejthetők, illetve megmutathatók. Ez az elem, az Expander típus lehetővé teszi, hogy az `ExpandDirection` tulajdonság segítségével meghatározzuk az elemek megjelenítésének irányát (fel, le, balra vagy jobbra). Tekintsük meg az alábbi XAML-leírást (amely alapvetően `<GroupBox>` típusról `<Expander>` típusra módosul):

```
<StackPanel>
  <Expander Header = "Select Your Music Media" BorderBrush ="Black">
    <StackPanel>
      <RadioButton GroupName = "Music" >CD Player</RadioButton>
      <RadioButton GroupName = "Music" >MP3 Player</RadioButton>
```

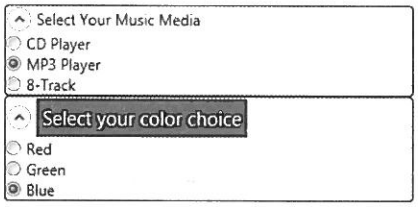
```
        <RadioButton GroupName = "Music" >8-Track</RadioButton>
      </StackPanel>
    </Expander>

    <Expander BorderBrush ="Black">
      <Expander.Header>
        <Label Background = "Blue" Foreground = "White"
               FontSize = "15" Content = "Select your color choice"/>
      </Expander.Header>
      <StackPanel>
        <RadioButton>Red</RadioButton>
        <RadioButton>Green</RadioButton>
        <RadioButton>Blue</RadioButton>
      </StackPanel>
    </Expander >
  </StackPanel>
```

A 29.9. ábra mutatja az egyes Expandereket összecsukott állapotban.



29.9. ábra: *Összecsukott Expanderek*

A 29.10. ábra mutatja az egyes Expandereket kinyitott állapotban.



29.10. ábra: *Kiterjesztett Expanderek*

Forráskód A CheckRadioGroup.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A ListBox és a ComboBox típusok használata

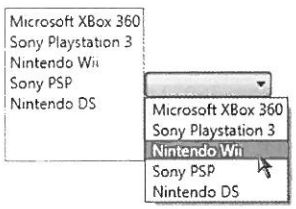
Ahogyan azt remélhettük, a WPF biztosít olyan típusokat, amelyek választható elemek csoportját tartalmazzák – ilyen a `ListBox` és a `ComboBox`, ezek mindegyike az `ItemsControl` absztrakt ősosztályból származik. A legfonto-

sabb, hogy ez a szülőosztály definiálja az `Items` nevű tulajdonságot, amely a részelemeket tároló, erősen típusos `ItemCollection` objektumot adja vissza. Az `ItemCollection` típus a `System.Object` típusokkal dolgozik, ezért gyakorlatilag bármit tartalmazhat. Ha markup révén szeretnénk egyszerű szöveges adatokkal feltölteni egy `ItemsControl`-leszármazott típust, akkor ezt `<ListBoxItem>` típusok készletének használatával tehetjük meg. Példaként nézzük a következő XAML-kódot:

```
<!-- Egyszerű listamező -->
<ListBox Name = "lstVideoGameConsoles">
  <ListBoxItem>Microsoft XBox 360</ListBoxItem>
  <ListBoxItem>Sony Playstation 3</ListBoxItem>
  <ListBoxItem>Nintendo Wii</ListBoxItem>
  <ListBoxItem>Sony PSP</ListBoxItem>
  <ListBoxItem>Nintendo DS</ListBoxItem>
  </ListBox>

<!-- Egyszerű legördülő listadoboz -->
<ComboBox Name = "comboVideoGameConsoles">
  <ListBoxItem>Microsoft XBox 360</ListBoxItem>
  <ListBoxItem>Sony Playstation 3</ListBoxItem>
  <ListBoxItem>Nintendo Wii</ListBoxItem>
  <ListBoxItem>Sony PSP</ListBoxItem>
  <ListBoxItem>Nintendo DS</ListBoxItem>
</ComboBox>
```

Megjegyzés A `ComboBox` típusokat feltölthetjük `<ListBoxItem>` elemek helyett `<ComboBokszItem>` elemek segítségével. Ezáltal hozzáférhetünk az `IsHighlighted` tulajdonsághoz, amelyet a `ListBoxItem` típus nem használ.

Nem meglepő módon a 29.11. ábrán látható renderelést látjuk.



29.11. ábra: *Egyszerű ListBox és ComboBox*

Lista-vezérlőelemek feltöltése programozott módon

A lista-vezérlőelemben lévő adatok gyakran csak futásidőben ismertek; lehet például, hogy a listamező elemeit egy adatbázis olvasásból visszakapott értékek, WFC-szolgáltatás meghívása vagy külső fájl beolvasása alapján kell feltöltenünk. Amikor programozott módon kell `ListBox` vagy `ComboBox` elemet feltöltenünk, egyszerűen használjuk az `ItemCollection` típus tagjait (`Add()`, `Remove()` stb.). Tételezzük fel, hogy új `ListControls` nevű Visual Studio 2008 WPF-alkalmazás projektünk van. A `lstVideoGameConsole` típus előző XAML-deklarációját a következők szerint lehet XAML-ben definiálni:

```
<Window x:Class="ListControls.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="ListControls" Height="300" Width="300" >
  <StackPanel>
    <!-- Ezt kód segítségével töltjük fel -->
    <ListBox Name = "lstVideoGameConsoles">
    </ListBox>
  </StackPanel>
</Window>
```

és a kapcsolódó kódfájlban az alábbiak szerint tölthetjük fel:

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

  private void FillListBox()
  {
    // Elemek hozzáadása a listamezőhöz.
    lstVideoGameConsoles.Items.Add("Microsoft XBox 360");
    lstVideoGameConsoles.Items.Add("Sony Playstation 3");
    lstVideoGameConsoles.Items.Add("Nintendo Wii");
    lstVideoGameConsoles.Items.Add("Sony PSP");
    lstVideoGameConsoles.Items.Add("Nintendo DS");
  }
}
```

Az egyik dolog, amely különösnek tűnhet a ListBox XAML-leírásában, hogy a `<ListBoxItem>` típusokat használtuk az elemek feltöltéséhez; itt azonban sztringtípusokat használtunk az `Add()` metódus meghívása során. Ennek rö-

vid magyarázata az, hogy a XMAL használatakor a <ListBoxItem> típusok kényelmesebbek abból a szempontból, hogy azokat a http://schemas.microsoft.com/winfx/2006/xaml/presentation XML-névtérben definiáltuk, ezért rendelkezünk a közvetlen referenciájukkal.

A háttérben a rendszer a ToString() metódust hívja minden egyes <ListBoxItem> típuson, így a végeredmény azonos. Ha System.String névtér használatával töltenénk fel a ListBox (vagy ComboBox) típust XAML-ben, akkor meg kellene határoznunk egy új XML-névteret, hogy behozzuk az mscorlib.dll fájlt (további részletekért lásd a 28. fejezetet):

```
<StackPanel xmlns:CorLib = "clr-namespace:System;assembly=mscorlib">
  <ListBox Name = "lstVideoGameConsoles">
    <CorLib:String>Microsoft XBox 360</CorLib:String>
    <CorLib:String>Sony Playstation 3</CorLib:String>
    <CorLib:String>Nintendo Wii</CorLib:String>
    <CorLib:String>Sony PSP</CorLib:String>
    <CorLib:String>Nintendo DS</CorLib:String>
  </ListBox>
</StackPanel>
```

Ellenben, ha szeretnénk, programozott módon feltölthetünk egy ItemsControl-leszármazott típust erősen típusos ListBoxItem objektumok segítségével; a jelenlegi példában azonban ezzel nem nyerünk semmit, hanem valójában csak további munkát generálunk magunknak (mivel a ListBoxItem nem rendelkezik konstruktorral a Content tulajdonság beállításához!).

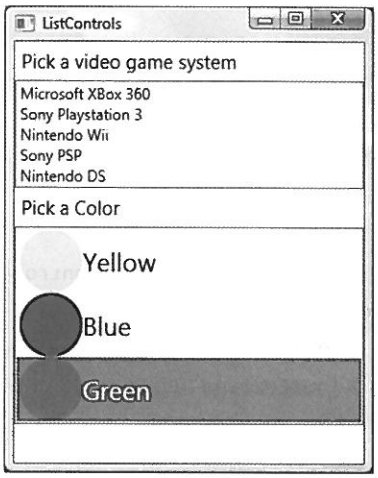
Tetszőleges tartalom hozzáadása

Mivel mind a ListBox, mind a ComboBox származtatási láncában megtalálható a ContentControl alaposztály, ezért azok egy egyszerű sztringnél jóval bonyolultabb adatokat is tartalmazhatnak. Tekintsük át a következő ComboBox típust, amely kétdimenziós grafikus objektumokat és egy leíró címkét tartalmazó <StackPanel> tárolókat foglal magában:

```
<StackPanel>
  <!-- Egy ListBox tartalommal -->
  <ListBox Name = "lstColors">
    <StackPanel Orientation ="Horizontal">
      <Ellipse Fill ="Yellow" Height ="50" Width ="50"/>
      <Label FontSize ="20" HorizontalAlignment="Center"
             VerticalAlignment="Center">Yellow</Label>
    </StackPanel>
    <StackPanel Orientation ="Horizontal">
```

```
        <Ellipse Fill ="Blue" Height ="50" Width ="50"/>
        <Label FontSize ="20" HorizontalAlignment="Center"
               VerticalAlignment="Center">Blue</Label>
      </StackPanel>
      <StackPanel Orientation ="Horizontal">
        <Ellipse Fill ="Green" Height ="50" Width ="50"/>
        <Label FontSize ="20" HorizontalAlignment="Center"
               VerticalAlignment="Center">Green</Label>
      </StackPanel>
    </ListBox>
  </StackPanel>
```

A 29.12. ábra mutatja az aktuális lista típusaink kimenetét.



29.12. ábra: *ItemsControl-leszármazott típusok tartalmazhatnak bármilyen jellegű tartalmat*

Az aktuális kiválasztás meghatározása

A `ListBox` vagy a `ComboBox` típus feltöltése után a következő nyilvánvaló kérdés az, hogy futásidőben hogyan lehet eldönteni, melyik elemet választotta ki a felhasználó. Mint látni fogjuk, erre három módszer létezik. Ha a kiválasztott elem numerikus indexének a kiderítése érdekel bennünket, akkor használhatjuk a `SelectedIndex` tulajdonságot (amely nulla alapú; a `-1` érték azt mutatja, hogy nincs kijelölés). Ha a listában a kiválasztott objektumot szeretnénk megszerezni, arra a `SelectedItem` tulajdonság való. Végül, a `SelectedValue` lehetővé teszi, hogy a kiválasztott objektum értékét megkapjuk (többnyire a `ToString()` metódus meghívásával).

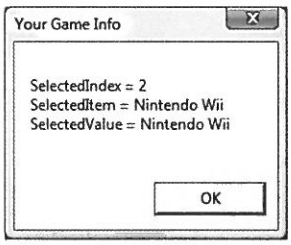
Nem hangzik bonyolultnak, igaz? Most, hogy teszteljük az egyes tulajdonságok viselkedését, tételezzük fel, hogy definiáltunk két új `Button` típust az aktuális ablak számára, amelyek egyaránt kezelik a kattintási eseményt:

```
<!-- Gombok a kiválasztott elem megszerzéséhez -->
<Button Name ="btnGetGameSystem" Click ="btnGetGameSystem_Click">
  Get Video Game System
</Button>
<Button Name ="btnGetColor" Click ="btnGetColor_Click">
  Get Color
</Button>
```

A `btnGetGameSystem` esetében a kattintási esemény kezelője megszerzi a `lstVideoGameConsoles` objektum `SelectedIndex`, `SelectedItem` és `SelectedValue` tulajdonságainak értékeit, és megjeleníti azokat egy üzenetdobozban:

```
protected void btnGetGameSystem_Click(object sender,
                                      RoutedEventArgs args)
{
  string data = string.Empty;
  data += string.Format("SelectedIndex = {0}\n",
          lstVideoGameConsoles.SelectedIndex);
  data += string.Format("SelectedItem = {0}\n",
          lstVideoGameConsoles.SelectedItem);
  data += string.Format("SelectedValue = {0}\n",
          lstVideoGameConsoles.SelectedValue);
  MessageBox.Show(data, "Your Game Info");
}
```

Ha a „Nintendo Wii"-t választanánk a játékkonzolok listájából, és a kapcsolódó gombra kattintanánk, akkor a 29.13. ábrán látható szövegdobozt kapnánk.

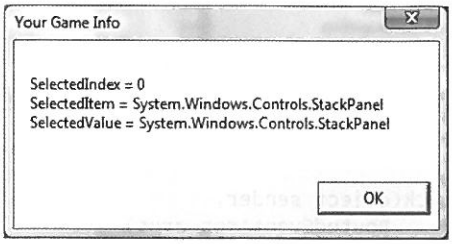


29.13. ábra: *Megtaláltuk a kiválasztott szringet*

Azonban, mi a helyzet a kiválasztott szín megszerzésével?

Az aktuális kiválasztás meghatározása a beágyazott tartalom esetében

Tételezzük fel, hogy a `btnGetColor` `Button` típus kattintási eseményének kezelője implementálja a `btnGetColor_Click()` metódust, hogy kiírja a `lstColors` `ListBox` objektum aktuális kijelölését, indexét és értékét. Ha most kiválasztanánk az első elemet a lstColors listamezőből (és a hozzátartozó gombra kattintanánk), meglepetésünkre a 29.14. ábrán látható kimenettel szembesülnénk.



29.14. ábra: *Megtaláltuk a kiválasztott... StackPanelt?*

Ennek a kimenetnek az oka az, hogy a `lstColors` objektum három `StackPanel` objektumot tart fenn, amelyek mindegyike rendelkezik beágyazott tartalommal. Ezért a `SelectedItem` és a `SelectedValue` tulajdonságok egyszerűen meghívják a `ToString()` metódust a `StackPanel` típuson, amely visszaadja teljesen meghatározott nevét.

Noha a `SelectedIndex` tulajdonságból visszaadott numerikus érték alapján könnyen kitalálhatnánk, hogy a felhasználó melyik elemet választotta, egy másik megközelítés az, hogy leásunk a `StackPanel` gyermekgyűjteményébe, hogy megkapjuk a `Label` típus `Content` értékét a `StackPanel` belsőleg fenntartott `Children` gyűjteményének segítségével:

```
protected void btnGetColor_Clicked(object sender,
                                   RoutedEventArgs args)
{
  // A Content érték megszerzése a StackPanel elemben
  // a kiválasztott Label típusban.
  StackPanel selectedStack =
             (StackPanel)lstColors.Items[lstColors.SelectedIndex];
  string color =
         ((Label)(selectedStack.Children[1])).Content.ToString();
  string data = string.Empty;
  data += string.Format("SelectedIndex = {0}\n",
          lstColors.SelectedIndex);
  data += string.Format("Color = {0}", color);
  MessageBox.Show(data, "Your Game Info");
}
```

Noha ez a megoldás megteszi, mégis elég kényes, mivel kódolt pozíciók vannak a `StackPanel` tárolóban (a második gyermek a `Label`), és számos kasztolási műveletet kell végrehajtanunk. Másik alternatíva minden `StackPanel` Tag tulajdonságának beállítása, amelyet a `FrameworkElement` alaposztályban definiálunk:

```
<ListBox Name = "lstColors">
  <StackPanel Orientation ="Horizontal" Tag ="Yellow">
  ...
  </StackPanel>
  <StackPanel Orientation ="Horizontal" Tag ="Blue">
  ...
  </StackPanel>
  <StackPanel Orientation ="Horizontal" Tag ="Green">
  ...
  </StackPanel>
</ListBox>
```

Ezen megközelítés használatával a kódunk jelentősen letisztul, mivel programozott módon kiszedhetjük a Tag tulajdonsághoz rendelt értéket, a következők szerint:

```
protected void btnGetColor_Clicked(object sender,
                                   RoutedEventArgs args)
{
  string data = string.Empty;
  data += string.Format("SelectedIndex = {0}\n",
          lstColors.SelectedIndex);
  data += string.Format("SelectedItem = {0}\n",
          lstColors.SelectedItem);
  data += string.Format("SelectedValue = {0}",
          (lstColors.Items[lstColors.SelectedIndex] as
          StackPanel).Tag);
  MessageBox.Show(data, "Your Color Info");
}
```

Noha ez a megközelítés egy kicsit tisztább, mint az első kísérletünk, vannak más módok is, ahol *adatsablonok* segítségével érhetjük el egy összetett vezérlőelem értékét. Ehhez meg kell értenünk a WPF adatkötési motor működését, amelyet a fejezet végén vizsgálunk meg.

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

A többsoros szövegbeviteli mezők használata

A WPF több olyan felhasználóifelület-elemet biztosít, amelyek lehetővé teszik a szövegalapú bevitelek összegyűjtését. A két legegyszerűbb típus a `TextBox` és a `PasswordBox`, amelyeket most a `TextControls` nevű, új Visual Studio 2008 WPF alkalmazás használatával vizsgálunk meg.

A TextBox típus használata

A múltban használt más `TextBox` típusokhoz hasonlóan a WPF `TextBox` típusát beállíthatjuk, hogy egy sornyi szöveget tartalmazzon (ez az alapértelmezett beállítás), vagy több sort, ha az `AcceptReturn` tulajdonság értéke `true`. A `TextBox`on belül az adatokat mindig karakteradatokként kezeljük, ezért a „tartalom" mindig olyan sztringtípus, amelyet a Text tulajdonság segítségével beállíthatunk és lekérdezhetünk:

```
<TextBox Name ="txtData" Text = "Hello!" BorderBrush ="Blue"
         Width ="100"/>
```

A WPF `TextBox` típusának egyik nagyon egyedi jellemzője, hogy beépített képessége révén ellenőrizheti a bevitt adatok helyesírását, ha a `SpellCheck.IsEnabled` tulajdonság értéke `true`. Ilyen esetekben látni fogjuk, hogy – a Microsoft Office-hoz hasonlóan – a helytelenül írt szavakat az alkalmazás vörös hullámos vonallal aláhúzza. Ennél még jobb, hogy az alapul szolgáló programozási modell segítségével hozzáférhetünk a helyesírásellenőrző-motorhoz, ezáltal megkaphatjuk a helytelenül írt szavakra vonatkozó javaslatok listáját.

Az alábbiak szerint módosítsuk az aktuális ablak XAML-definícióját, hogy használhassuk a `Label`, a `TextBox` és a `Button` típusokat (figyeljük meg, hogy a `TextBox` többsoros szövegeket támogat, és engedélyezi a helyesírás-ellenőrzést):

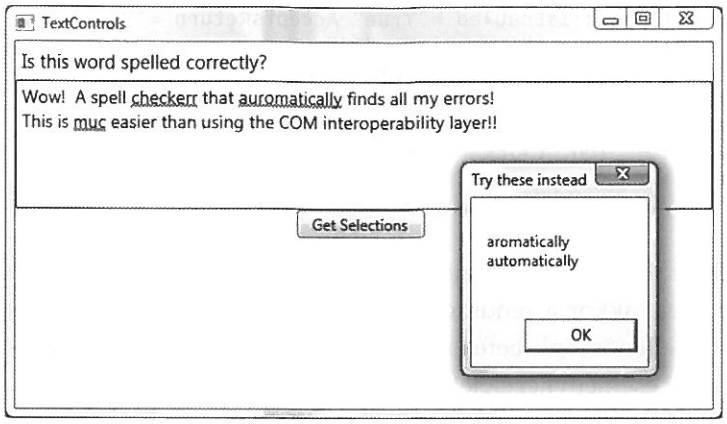
```
<Window x:Class="TextControls.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="TextControls" Height="204" Width="292" >
  <StackPanel>
    <Label FontSize ="15">Is this word spelled correctly?</Label>
```

```
    <TextBox SpellCheck.IsEnabled ="True" AcceptsReturn ="True"
             Name ="txtData" FontSize ="12"
             BorderBrush ="Blue" Height ="100">
    </TextBox>
    <Button Name ="btnOK" Content ="Get Selections"
            Width = "100" Click ="btnOK_Click"/>
  </StackPanel>
</Window>
```

Ilyen funkcionalitással már látjuk majd, hogy amikor helytelenül írunk be szavakat a `TextBox`-ba, akkor a rendszer megjelöli a helyesírási hibákat. Az egyszerű helyesírás-ellenőrzőnk befejezéséhez a következőképpen módosítsuk a gomb kattintási eseménykezelőjét:

```
protected void btnOK_Click(object sender, RoutedEventArgs args)
{
  string spellingHints = string.Empty;

  // Próbáljunk helyesírási hibát keresni a kurzor aktuális
  // helyénél.
  SpellingError error =
               txtData.GetSpellingError(txtData.CaretIndex);
  if (error != null)
  {
    // Készítsük el a helyesírási javaslatok sztringjét.
    foreach (string s in error.Suggestions)
    {
      spellingHints += string.Format("{0}\n", s);
    }

    // Mutassuk meg a javaslatokat!
    MessageBox.Show(spellingHints, "Try these instead");
  }
}
```

A kód meglehetősen egyszerű. Csak megkeressük a kurzor jelenlegi helyét a szövegdobozban a `CaretIndex` tulajdonság használatával, hogy kinyerjünk egy `SpellingError` objektumot. Ha valami hiba van a mondott helyen (vagyis az érték nem `null`), akkor végiglépkedünk a javaslatok listáján a stílszerűen elnevezett `Suggestions` tulajdonság révén. Végül megjelenítjük a lehetőségeket egy egyszerű `MessageBox.Show()` kérés használatával. A 29.15. ábra lehetséges tesztfutást mutat, amikor a kurzor a helytelenül írt „auromatically" szón áll.



29.15. ábra: *Egy egyedi helyesírás-ellenőrző!*

A PasswordBox típus használata

A `PasswordBox` típus - nem meglepően - egy olyan biztonságos helyet kínál, ahova érzékeny szöveges adatokat írhatunk be. Alapértelmezés szerint a jelszó karakterkör típusú, ez azonban megváltoztatható a `PasswordChar` tulajdonság használatával. A végfelhasználó által beírt érték megszerzéséhez egyszerűen ellenőrizzük a `Password` tulajdonságot. Módosítsuk a jelenlegi helyesírás-ellenőrző alkalmazásunkat a helyes jelszó kérésével, hogy láthassuk a helyesírási javaslatok listáját. Először módosítsuk a már meglévő `<StackPanel>` tárolót egy beágyazott `<StackPanel>` típussal, amely vízszintesen elhelyezi a meglévő `<Button>` típus mellé a `PasswordBox` mezőt:

```
<StackPanel>
  <Label FontSize ="15">Is this word spelled correctly?</Label>
  <TextBox SpellCheck.IsEnabled ="True" AcceptsReturn ="True"
           Name ="txtFavoriteColor" FontSize ="14"
           BorderBrush ="Blue" Height ="100">
  </TextBox>
  <StackPanel Orientation ="Horizontal">
    <PasswordBox Name ="pwdText" BorderBrush ="Black"
                 Width ="100"></PasswordBox>
    <Button Name ="btnOK" Content ="Get Selections"
            Width = "100" Click ="btnOK_Click"/>
  </StackPanel>
</StackPanel>
```

Most módosítsuk úgy a jelenlegi `Button` kattintási eseménykezelőjét, hogy meghívja a `CheckPassword()` segédfüggvényt, amely teszteli a kódolt sztringet. Győződjünk meg arról, hogy a javaslatok csak akkor jelenhessenek meg, ha a jelszó ellenőrzése sikeres volt. Íme a releváns módosítások:

```
public partial class MainWindow : System.Windows.Window
{
  ...
  protected void btnOK_Click(object sender, RoutedEventArgs args)
  {
    if (CheckPassword())
    {
      // Ugyanaz a helyesírás-ellenőrző logika, mint korábban...
    }
    else
      MessageBox.Show("Security error!!");
  }

  private bool CheckPassword()
 {
    if (pwdText.Password == "Chucky")
    return true;
  else
    return false;
  }
}
```

A `TextBox` és a `PasswordBox` mellett ne feledjük, hogy ha egy olyan alkalmazást készítünk, amely rendelkezik olyan szövegmezővel, amely bármilyen típusú tartalmat (grafikus renderelést, szöveget stb.) magában foglalhat, a WPF biztosítja számunkra a `RichTextBox` típust. Emellett, ha kell a teljesítmény egy különösen szövegintenzív alkalmazás készítéséhez, a WPF biztosít egy teljes dokumentumprezentációs API-t, amelyet elsősorban a `System.Windows.Documents` névtér képvisel.

Itt találunk olyan típusokat, amelyek lehetővé teszik *folyószöveg-dokumentumok* (flow document) készítését, és ezek segítségével programozott módon ábrázolhatunk (XAML- vagy C#-kódban) bekezdéseket, kapcsolódó szövegblokk-részeket, feljegyzéseket, jegyzeteket, táblázatokat és egyéb gazdag dokumentumcentrikus típusokat. A könyv jelen kiadása nem tárgyalja a `RichTextBox` típust, illetve a folyószöveg-dokumentum API-t; ha érdekesnek bizonyul a téma, további részletekért lapozzuk fel a .NET Framework 3.5 SDK dokumentációját.

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

Ezzel befejeztük a WPF vezérlőelemkészlet áttekintését. A fejezet későbbi részében látni fogjuk, hogyan készíthetünk menürendszereket, állapotsávokat és eszköztárakat. A következő feladatunk azonban annak áttekintése, hogyan rendezhetjük a felhasználói felület elemeit `Window` típuson belül tetszőleges számú paneltípus használatával.

A tartalomelrendezés kezelése panelek használatával

Egy valós WPF-alkalmazás mindenképpen tartalmaz jó néhány felhasználóifelület-elemet (beviteli vezérlőelemeket, grafikus tartalmat, menürendszereket, állapotsávokat stb.), amelyeket jól kell elrendezni a tároló ablakban. Emellett, ha egyszer elhelyeztük a felhasználói felület vezérlőit az új helyükre, biztosak akarunk lenni abban, hogy az elvártak szerint viselkednek, amikor a végfelhasználó átméretezi az ablakot vagy akár az ablak egy részét (osztott ablak esetében). Ahhoz, hogy biztosítsuk, a WPF-vezérlőelemek megtartsák pozíciójukat a hosztoló ablakban, több *paneltípus* áll rendelkezésünkre.

Ahogy az előző fejezetből emlékezhetünk rá, hogy amikor tartalmat helyezünk el olyan ablakban, amely nem használ paneleket, az hajszálpontosan a tároló közepére kerül. Tekintsük át az alábbi egyszerű ablakdeklarációt, amely egyetlen `Button` típust tartalmaz. Függetlenül attól, hogyan méretezzük át az ablakot, a felhasználóifelület-vezérlő mindig egyenlő távolságra jelenik meg az adott terület négy oldalától.

```
<!-- Ez a gomb mindig az ablak közepén van -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">
  <Button Name="btnOK" Height = "100" Width="80">OK</Button>
</Window>
```

Arra is emlékezzünk, hogy ha több elemet próbálunk elhelyezni közvetlenül egy `<Window>` típus hatókörén belül, akkor címkézési és/vagy fordítási idejű hibát kapunk. Ezeknek a hibáknak az az oka, hogy egy ablak (vagy ami azt illeti, a `ContentControl` bármelyik leszármazottja) csak egyetlen objektumot rendelhet a `Content` tulajdonságához:

```
<!-- Hiba! A Content tulajdonságot implicit módon többször
     állítottuk be! -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">

  <!-- Nyugtázva! A <Window> két közvetlen gyermekeleme! -->
  <Label Name="lblInstructions"
         Width="328" Height="27" FontSize="15">Enter
         Car Information</Label>
  <Button Name="btnOK" Height = "100" Width="80">OK</Button>
</Window>
```

Nyilvánvaló, hogy nem sok haszna van egy olyan ablaknak, amely csak egy elemet tartalmazhat. Ha arra van szükség, hogy egy ablak több elemet tartalmazzon, akkor azokat tetszőleges számú panelbe kell rendeznünk. A panel tartalmazza az ablakot képviselő felhasználóifelület-elemeket, amely után a panelt magát csak a `Content` tulajdonsághoz rendelt objektumként használjuk.

A WPF alapvető paneltípusai

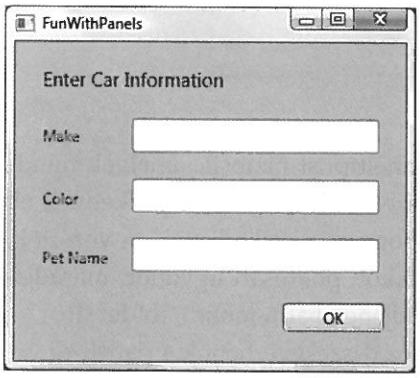
A `System.Windows.Controls` névtér több paneltípust biztosít, amelyek mindegyike azt szabályozza, hogy az egyes részelemek hogyan helyezkednek el. A panelek használatával megadhatjuk, hogyan viselkedjenek a vezérlők, amikor a végfelhasználó átméretezi az ablakot: pontosan ugyanott maradjanak, ahova tervezéskor kerültek; átrendeződjenek balra-jobbra, föl-le; stb.

Bonyolult felhasználói felület készítéséhez összekeverhetjük a panelbeli vezérlőelemeket (pl. egy `StackPanelt` tartalmazó `DockPanel`), ezáltal nagyfokú rugalmasságot és kezelhetőséget biztosíthatunk. Ezenfelül a paneltípusok együtt tudnak dolgozni más dokumentumközpontú vezérlőelemekkel (mint a `ViewBox`, a `TextBlock`, a `TextFlow` és a `Paragraph` típusok), hogy tovább lehessen finomítani a tartalom elrendezését egy adott panelen belül. A 29.3. táblázat bemutatja néhány gyakran használt WPF panelbeli vezérlőelem szerepét.

Panelbeli vezérlőelem	Jelentés
`Canvas`	„Klasszikus" mód a tartalom elhelyezésére. Az elemek pontosan ugyanott maradnak, ahová a tervezéskor tettük őket.
`DockPanel`	Zárolja a tartalmat a panel megadott oldalára (`Top`, `Bottom`, `Left` vagy `Right`).
`Grid`	Cellák sorozatába rendezi a tartalmat, táblázatos rácsformában.
`StackPanel`	Függőlegesen vagy vízszintesen egymás után helyezi a tartalmat, ahogy azt az `Orientation` tulajdonság megkívánja.
`WrapPanel`	Balról jobbra pozícionálja a tartalmat, új sorba tördelve azt a tároló doboz szélénél. A további rendezés folyamatosan történik felülről lefelé vagy jobbról balra, az `Orientation` tulajdonság értékétől függően.

29.3. táblázat: *Alapvető WPF panelbeli vezérlőelemek*

A következő részben bemutatjuk ezeknek a gyakran használt paneltípusoknak a használatát, elkészítjük a 29.16. ábrán látható grafikus felületet különböző paneleken belül, és megvizsgáljuk, hogyan változik az elhelyezés, ha az ablakot átméretezzük.



29.16. ábra: *A felhasználói felület megcélzott elrendezése*

Tartalom elhelyezése a Canvas paneleken belül

Vitathatatlanul a legegyszerűbb panel a Canvas. A Canvas az a panel, amelyben leginkább otthon érezzük magunkat, ugyanis egy Windows Forms alkalmazás alapértelmezett elrendezését utánozza. A Canvas panel lehetővé teszi a

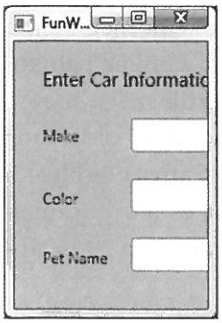
felhasználói felület tartalmának abszolút elhelyezését. Ha a végfelhasználó a Canvas panelen fenntartott elrendezésnél kisebb területűre méretezi az ablakot, a belső tartalom addig nem lesz látható, amíg a tárolót ugyanakkorára vagy nagyobbra nem terjeszti ki, mint a Canvas területe.

Ahhoz, hogy a `Canvas` típushoz tartalmat adjunk, határozzuk meg meg a szükséges részelemeket a nyitó `<Canvas>` és a záró `</Canvas>` címkék hatókörén belül, valamint adjuk meg a renderelés helyét (jegyezzük meg, hogy a tartalom pozíciója relatív lehet a Canvas jobb/bal vagy alsó/felső oldalához képest, de nem mindkettőhöz). Ha azt szeretnénk, hogy a Canvas területe a tároló egész területét fedje, egyszerűen hagyjuk el a `Height` és a `Width` tulajdonságokat. Tekintsük meg a következő XAML-markupot, amely a 29.16. ábrán látható elrendezést definiálja:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">

  <Canvas Background="LightSteelBlue">
    <Button Canvas.Left="212" Canvas.Top="203" Name="btnOK"
            Width="80">OK</Button>
    <Label Canvas.Left="17" Canvas.Top="14" Name="lblInstructions"
           Width="328" Height="27"
           FontSize="15">Enter Car Information</Label>
    <Label Canvas.Left="17" Canvas.Top="60"
           Name="lblMake">Make</Label>
    <TextBox Canvas.Left="94" Canvas.Top="60" Name="txtMake"
             Width="193" Height="25"/>
    <Label Canvas.Left="17" Canvas.Top="109"
           Name="lblColor">Color</Label>
    <TextBox Canvas.Left="94" Canvas.Top="107" Name="txtColor"
             Width="193" Height="25"/>
    <Label Canvas.Left="17" Canvas.Top="155"
           Name="lblPetName">Pet Name</Label>
    <TextBox Canvas.Left="94" Canvas.Top="153" Name="txtPetName"
             Width="193" Height="25"/>

  </Canvas>

</Window>
```

Ebben a példában a `<Canvas>` hatókörén belül lévő összes elemet egy `Canvas.Left` és egy `Canvas.Top` érték minősít, amelyek a tartalom felső és bal oldali elhelyezkedését vezérlik a panelen belül, a csatolt tulajdonságszintaxis használatával (lásd a 28. fejezetet). Ahogy kitalálhattuk, a függőleges elhelyezkedést a Top, illetve a `Bottom` tulajdonság, míg a vízszintes elhelyezkedést a `Left`, illetve a `Right` tulajdonság vezérli.

Mivel minden vezérlőt a <Canvas> elemen belül helyeztünk el, látni fogjuk, hogy az ablak átméretezésekor a rendszer eltakarja a vezérlőket, ha a tároló területe kisebb lesz, mint a tartalom (lásd a 29.17. ábrát).



29.17. ábra: *A Canvas panel tartalma lehetővé teszi az abszolút pozícionálást*

Annak a sorrendje, hogyan deklarálunk tartalmat egy Canvas típuson belül, befolyásolja az elhelyezkedés kiszámítását, ugyanis ez a vezérlőelem méretén, valamint a Canvas.Top, a Canvas.Bottom, a Canvas.Left és a Canvas.Right tulajdonságokon alapul. Ezek alapján az alábbi markup (amely csoportosítja a hasonló vezérlőelemeket) ugyanolyan renderelést eredményez:

```
<Canvas Background="LightSteelBlue">
  <TextBox Canvas.Left="94" Canvas.Top="153" Name="txtColor"
           Width="193" Height="25"/>
  <TextBox Canvas.Left="94" Canvas.Top="60" Name="txtPetName"
           Width="193" Height="25"/>
  <TextBox Canvas.Left="94" Canvas.Top="107" Name="txtMake"
           Width="193" Height="25"/>
  <Label Canvas.Left="17" Canvas.Top="14" Name="lblInstructions"
         Width="328" Height="27"
         FontSize="15">Enter Car Information</Label>
  <Label Canvas.Left="17" Canvas.Top="109"
         Name="lblColor">Color</Label>
  <Label Canvas.Left="17" Canvas.Top="155"
         Name="lblMake">Pet Name</Label>
  <Label Canvas.Left="17" Canvas.Top="60"
         Name="lblPetName">Make</Label>

  <Button Canvas.Left="212" Canvas.Top="203" Name="btnOK"
          Width="80">OK</Button>
</Canvas>
```

Megjegyzés Ha egy Canvas típuson belüli részelemek nem határoznak meg adott helyet a csatolt tulajdonságszintaxis révén, akkor azok automatikusan a bal felső sarokba kerülnek.

Noha a Canvas típus használata kívánatosnak tűnhet a tartalom elrendezéséhez (mivel olyan ismerősnek tűnik), mégis vannak korlátai. Először is, egy Canvas típuson belüli elemek nem méretezik át magukat dinamikusan stílusok vagy sablonok alkalmazásakor (pl. a betűk mérete nem változik). A másik nyilvánvaló korlát az, hogy a Canvas nem próbálja meg láthatóan megtartani az elemeket, amikor a végfelhasználó kisebbre méretezi át az ablakot.

A Canvas típus legjobb használata talán a grafikus tartalom elhelyezése. Például, ha egyedi képet készítenénk XAML használatával, akkor bizonyára azt akarnánk, hogy a vonalak, a formák és a szöveg ugyanazon a helyen maradjon ahelyett, hogy dinamikusan áthelyeződjenek, ha a felhasználó átméretezi az ablakot. Újra megvizsgáljuk a Canvas típust a következő fejezetben, amikor a WPF grafikus renderelési szolgáltatásairól lesz szó.

Forráskód A SimpleCanvas.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

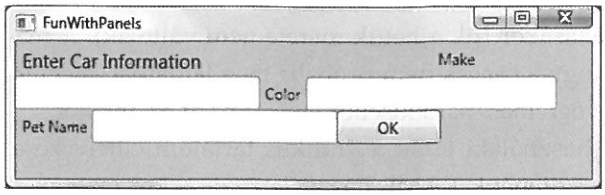
Tartalom elhelyezése a WrapPanel paneleken belül

A WrapPanel lehetővé teszi olyan tartalom definiálását, amely átnyúlik a panelen, ha az ablakot átméretezik. Amikor elhelyezünk elemeket egy WrapPanel panelben, nem kell megadnunk felső, alsó, bal, illetve jobb oldali elhelyezési értékeket, mint általában a Canvas esetében tesszük. Ehelyett minden részelem szabadon meghatározhat egy Height és egy Width értéket (egyéb tulajdonságértékek mellett), hogy szabályozza az általános méretét a tárolóban.

Mivel a WrapPanelen belül a tartalom nem „rögzül" a panel egy adott oldalához, ezért fontos a sorrend, amelyben az elemeket deklaráljuk (a tartalom renderelése az első elemtől az utolsóig történik). Nézzük meg a következő XAML-kódrészletet:

```
<WrapPanel Background="LightSteelBlue">
  <Label Name="lblInstruction" Width="328"
         Height="27" FontSize="15">Enter Car Information</Label>
  <Label Name="lblMake">Make</Label>
  <TextBox Name="txtMake" Width="193" Height="25"/>
  <Label Name="lblColor">Color</Label>
  <TextBox Name="txtColor" Width="193" Height="25"/>
  <Label Name="lblPetName">Pet Name</Label>
  <TextBox Name="txtPetName" Width="193" Height="25"/>
  <Button Name="btnOK" Width="80">OK</Button>
</WrapPanel>
```

Amikor megtekintjük ezt a markupot, a tartalom változtatja a formáját a szélesség átméretezésekor, mivel balról jobbra halad az ablakban (lásd a 29.18. ábrát).



29.18. ábra: *Egy WrapPanel panelben a tartalom úgy viselkedik, mint egy közönséges HTML-oldal*

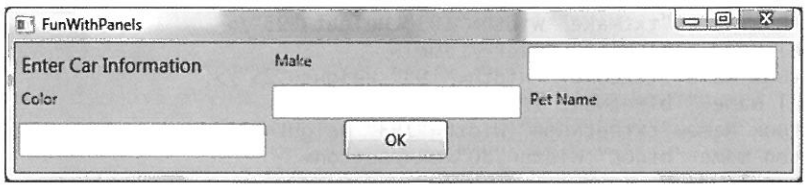
Alapértelmezés szerint a WrapPanel tartalma balról jobbra halad. Ha azonban megváltoztatjuk az `Orientation` tulajdonság értékét `Vertical`-ra, a tartalom fentről lefelé halad:

```
<WrapPanel Background="LightSteelBlue" Orientation ="Vertical">
```

A WrapPanel (valamint néhány más paneltípus) deklarálható az `ItemWidth` és az `ItemHeight` értékek meghatározásával, amelyek az egyes elemek alapértelmezett méretét szabályozzák. Ha egy részelem megadja a saját `Height` és/vagy `Width` értékét, akkor az a panel által megállapított mérethez viszonyítva helyezzük el. Tekintsük meg a következő markupot:

```
<WrapPanel Background="LightSteelBlue" ItemWidth ="200"
           ItemHeight ="30">
  <Label Name="lblInstruction"
         FontSize="15">Enter Car Information</Label>
  <Label Name="lblMake">Make</Label>
  <TextBox Name="txtMake"/>
  <Label Name="lblColor">Color</Label>
  <TextBox Name="txtColor"/>
  <Label Name="lblPetName">Pet Name</Label>
  <TextBox Name="txtPetName"/>
  <Button Name="btnOK" Width ="80">OK</Button>
</WrapPanel>
```

Renderelés után a 29.19. ábrán látható kimenetet kapjuk (figyeljük meg a Button vezérlő méretét és pozícióját).



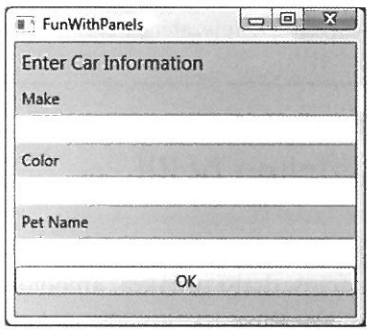
29.19. ábra: *A WrapPanel megállapíthatja egy adott elem szélességét és magasságát*

A 29.19. ábra megtekintése után egyetérthetünk abban, hogy a WrapPanel általában nem a legjobb választás tartalom közvetlen elrendezésére egy ablakban, mivel az elemek összekeveredhetnek, ha a felhasználó átméretezi az ablakot. A WrapPanel a legtöbb esetben egy másik paneltípus részeleme lesz, és lehetővé teszi az ablak egy kis területének, hogy átméretezéskor becsomagolja a tartalmát.

Forráskód A SimpleWrapPanel.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Tartalom elhelyezése a StackPanel paneleken belül

A WrapPanelhez hasonlóan a StackPanel vezérlőelem egy sorba rendezi a tartalmat, amely vízszintesen vagy függőlegesen igazítható (ez az alapértelmezett) az `Orientation` tulajdonsághoz rendelt érték alapján. A különbség az, hogy a StackPanel *nem* kísérli meg a tartalom becsomagolását, amikor a felhasználó átméretezi az ablakot. Ehelyett, a StackPanel panelben lévő elemek egyszerűen megnyúlnak (a tájolásuk alapján), hogy illeszkedjenek a StackPanel méretéhez.



29.20. ábra: *A tartalom függőleges egymás után helyezése*

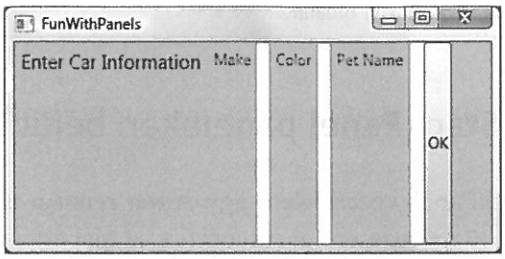
A következő markup például a 29.20. ábrán látható kimenetet eredményezi:

```
<StackPanel Background="LightSteelBlue">
  <Label Name="lblInstruction"
         FontSize="15">Enter Car Information</Label>
  <Label Name="lblMake">Make</Label>
  <TextBox Name="txtMake"/>
  <Label Name="lblColor">Color</Label>
  <TextBox Name="txtColor"/>
```

```
    <Label Name="lblPetName">Pet Name</Label>
    <TextBox Name="txtPetName"/>
    <Button Name="btnOK">OK</Button>
</StackPanel>
```

Ha az alábbiak szerint `Horizontal` értékre változtatjuk az `Orientation` tulajdonságot, akkor a renderelt kimenet a 29.21. ábrán láthatóval egyezik:

```
<StackPanel Background="LightSteelBlue" Orientation ="Horizontal">
```



29.21. ábra: *A tartalom vízszintes egymás mellé helyezése*

Ugyanúgy, mint a WrapPanel esetében, ritkán használjuk majd a `StackPanel` típust arra, hogy közvetlenül rendezzük el a tartalmat egy ablakon belül. A StackPanel ehelyett jobban alkalmazható egy főpanel részpaneleként.

Forráskód A SimpleStackPanel.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Tartalom elhelyezése a Grid paneleken belül

A WPF API-kon belül kínált panelek közül a Grid messze a legrugalmasabb. A HTML-táblázatokhoz hasonlóan a `Grid` típus feloszthatő cellákra, amelyek mindegyikében lehet tartalom. Egy `Grid` definiálásához három lépést kell végrehajtanunk:

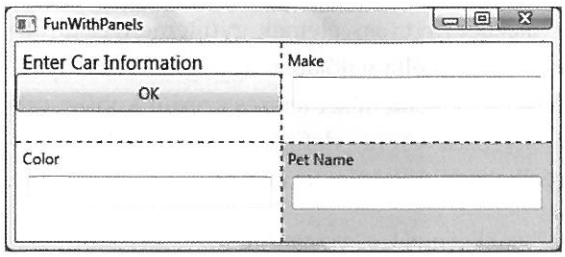
1. Az egyes oszlopok meghatározása és konfigurálása.
2. Az egyes sorok meghatározása és konfigurálása.
3. Tartalom hozzárendelése a rács minden cellájához a csatolt tulajdonságszintaxis használatával.

Megjegyzés Ha nem határozunk meg sorokat vagy oszlopokat, akkor a <Grid> alapértelmezés szerint egyetlen cella, amely kitölti az ablak teljes felületét. Továbbá, ha nem rendelünk hozzá cellaértéket egy részelemhez a <Grid> típuson belül, akkor az automatikusan a 0. oszlop 0. sorába kerül.

Az első két lépést (az oszlopok és a sorok definiálását) a <Grid.ColumnDefinitions> és a <Grid.RowDefinitions> elemek használatával érhetjük el, amelyek a <ColumnDefinition>, illetve a <RowDefinition> elemek gyűjteményét tartalmazzák. Mivel egy rácson belül minden cella valójában egy igazi .NET-típus, ezért az egyes elemek megjelenését és viselkedését tetszés szerint konfigurálhatjuk. Íme, egy meglehetősen egyszerű <Grid> definíció, amely a 29.22. ábrán látható módon rendezi el a felhasználói felület tartalmát:

```
<Grid ShowGridLines ="True" Background ="AliceBlue">
  <!-- A sorok/oszlopok meghatározása -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>

  <!-- Most hozzáadjuk az elemeket a rács celláihoz -->
  <Label Name="lblInstruction" Grid.Column ="0" Grid.Row ="0"
         FontSize="15">Enter Car Information</Label>
  <Button Name="btnOK" Height ="30" Grid.Column ="0"
          Grid.Row ="0" >OK</Button>
  <Label Name="lblMake" Grid.Column ="1" Grid.Row ="0">Make</Label>
  <TextBox Name="txtMake" Grid.Column ="1" Grid.Row ="0"
           Width="193" Height="25"/>
  <Label Name="lblColor" Grid.Column ="0"
         Grid.Row ="1" >Color</Label>
  <TextBox Name="txtColor" Width="193" Height="25" Grid.Column ="0"
           Grid.Row ="1" />

  <!-- Hogy érdekesebb legyen, adjunk valamilyen színt a PetName
       cellának -->
  <Rectangle Fill ="LightGreen" Grid.Column ="1" Grid.Row ="1" />
  <Label Name="lblPetName" Grid.Column ="1"
         Grid.Row ="1" >Pet Name</Label>
  <TextBox Name="txtPetName" Grid.Column ="1" Grid.Row ="1"
           Width="193" Height="25"/>
</Grid>
```

Figyeljük meg, hogy minden elem (beleértve a ráadásként hozzáadott világoszöld `Rectangle` elemet is) a rács egy cellájához kapcsolja magát a `Grid.Row` és a `Grid.Column` csatolt tulajdonságok segítségével. Alapértelmezés szerint a rácsban lévő cellák rendezése a bal felső sarokban kezdődik, amelyet a `Grid.Column="0"` `Grid.Row="0"` ad meg. Mivel a mi rácsunk összesen négy cellát definiál, a jobb alsó cellát a `Grid.Column="1"` `Grid.Row="1"` azonosítja.



29.22. ábra: *A Grid panel működés közben*

Forráskód A SimpleGrid.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

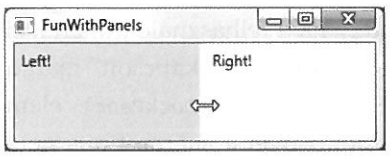
Rácsok GridSplitter típusokkal

A `Grid` típusok az úgynevezett választóvonalakat támogatják. Ahogy bizonyára tudjuk, a választóvonalak lehetővé teszik, hogy a végfelhasználó átméretezze egy rácstípus sorait vagy oszlopait. Ha ez készen van, az egyes átméretezhető cellákban lévő tartalom átalakítja magát a befoglalt elemek alapján. Nagyon könnyű választóvonalat hozzáadni egy `Grid` típushoz; egyszerűen definiáljuk a `<GridSplitter>` típust a csatolt tulajdonságszintaxis használatával, és határozzuk meg, mely sorra vagy oszlopra érvényes. Figyeljünk arra, hogy hozzá kell rendelni egy `Width` vagy egy `Height` értéket (attól függően, hogy függőleges vagy vízszintes osztásról van szó) annak érdekében, hogy látható legyen a képernyőn. Vizsgáljuk meg a következő `Grid` típust, amelyben választóvonal van az első oszlopon (`Grid.Column = "0"`):

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="FunWithPanels" Height="191" Width="436">
  <Grid Background ="AliceBlue">
    <!-- Az oszlopok meghatározása -->
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width ="Auto"/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
```

```
    <!-- Címke hozzáadása a 0 cellához -->
    <Label Name="lblLeft" Background ="GreenYellow"
           Grid.Column="0" Content ="Left!"/>

    <!-- A választóvonal meghatározása -->
    <GridSplitter Grid.Column ="0" Width ="5"/>

    <!-- Címke hozzáadása az 1 cellához -->
    <Label Name="lblRight" Grid.Column ="1" Content ="Right!"/>
  </Grid>
</Window>
```

Mindenekelőtt vegyük észre, hogy az oszlop, amely támogatja a választóvonalat, rendelkezik az `Auto.Next Width` tulajdonságával, míg a `<GridSplitter>` a csatolt tulajdonságszintaxis segítségével állapítja meg, melyik oszloppal dolgozik. Ha megnéznénk ezt a kimenetet, látnánk egy öt képpontos választóvonalat, amely lehetővé teszi, hogy átméretezzük az egyes Label típusokat (mivel nem adtunk meg `Height` vagy `Width` tulajdonságot a `Label` elemekhez, ezért azok betöltik az egész cellát). Nézzük meg a 29.23. ábrát.



29.23. ábra: *Választóvonalakat tartalmazó Grid típusok*

Forráskód A GridWithSplitter.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Tartalom elhelyezése a DockPanel paneleken belül

A `DockPanel` típust általában olyan főpanelként használjuk, amely tetszőleges számú további panelt foglalhat magában a kapcsolódó tartalom csoportosítására. Ahogyan a `Canvas` típusnál láttuk, a `DockPanel` panelek a csatolt tulajdonságszintaxis segítségével szabályozzák, hogy a bal felső sarkuk (az alapértelmezett) hova csatolja magát a panelen belül. Íme, egy nagyon egyszerű `DockPanel` definíció, amely a 29.24. ábrán látható kimenetet eredményezi:

```
<DockPanel LastChildFill ="True">
  <!-- Elemek dokkolása a panelhez -->
  <Label DockPanel.Dock ="Top" Name="lblInstruction"
         FontSize="15">Enter Car Information</Label>
```

```
    <Label DockPanel.Dock ="Left" Name="lblMake">Make</Label>
    <Label DockPanel.Dock ="Right" Name="lblColor">Color</Label>
    <Label DockPanel.Dock ="Bottom" Name="lblPetName">Pet Name</Label>
    <Button Name="btnOK">OK</Button>
</DockPanel>
```

29.24. ábra: *Egy egyszerű DockPanel*

Megjegyzés Ha több elemet adunk hozzá egy DockPanel ugyanazon oldalához, azokat a megadott oldal mellett a deklaráció sorrendjében halmozza fel a rendszer.

A `DockPanel` típusok használatának előnye, hogy ha a felhasználó átméretezi az ablakot, akkor minden elem a panel megadott oldalához „kapcsolt" marad (a `DockPanel.Dock` révén). Azt is vegyük észre, hogy a nyitó `<DockPanel>` elem `true` értékre állítja a `LastChildFill` attribútumot. Mivel a `Button` típus nem adott meg semmilyen `DockPanel.Dock` értéket, ezért kitölti megmaradt helyet.

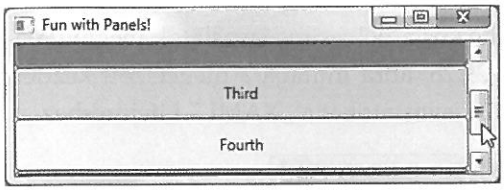
Forráskód A SimpleDockPanel.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A lapozás engedélyezése a paneltípusoknál

Érdemes rámutatni, hogy a WPF biztosítja a `<ScrollViewer>` típust, amely automatikus lapozási viselkedést kínál a beágyazott paneltípusok számára:

```
<ScrollViewer>
  <StackPanel>
    <Button Content ="First" Background = "Green" Height ="40"/>
    <Button Content ="Second" Background = "Red" Height ="40"/>
    <Button Content ="Third" Background = "Pink" Height ="40"/>
    <Button Content ="Fourth" Background = "Yellow" Height ="40"/>
    <Button Content ="Fifth" Background = "Blue" Height ="40"/>
  </StackPanel>
</ScrollViewer>
```

Az előző XAML-definíció eredménye a 29.25. ábrán látható.



29.25. ábra: *A ScrollViewer típus használata*

Forráskód A ScrollViewer.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Ahogy várhatjuk, minden panel több tagot biztosít, amelyek lehetővé teszik a tartalom elhelyezésének finomhangolását. Ehhez kapcsolódó megjegyzés, hogy a WPF-vezérlőelemek mindegyike két érdekes tulajdonságot (`Padding` és `Margin`) támogat, amelyek segítségével maga a vezérlőelem tájékoztathatja a panelt arról, hogy milyen bánásmódot igényel. Pontosabban a `Padding` tulajdonság szabályozza, hogy mekkora plusz hely vegye körül a belső vezérlőelemet, míg a `Margin` szabályozza a vezérlőelem külső részét körülvevő extra helyet.

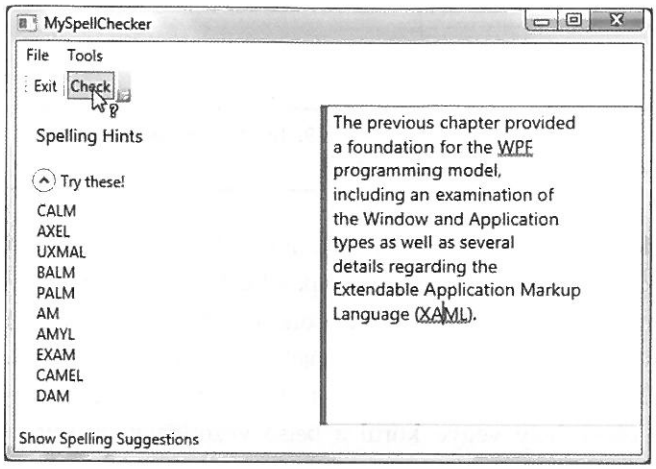
Ezzel befejeztük a WPF főbb paneltípusainak, valamint a tartalmuk pozícionálását szolgáló különböző módszerek áttekintését. Ezután megnézzük a beágyazott panelek használatának egy példáját, hogy létrehozzunk egy elrendezés-rendszert a főablak számára. Ehhez bővítjük a TextControls projekt (vagyis a helyesírásellenőrző-alkalmazás) funkcionalitását, hogy támogasson egy főmenüt, egy állapotsávot és egy eszköztárat.

Ablak kereteinek készítése beágyazott panelek használatával

Az alkalmazás frissített verzióját (amely az új, MySpellChecker nevű Visual Studio 2008 WPF alkalmazás projekt) kibővítjük és véglegesítjük a következő oldalakon, tehát pillanatnyilag az alapelrendezést és működést fogjuk összeállítani.

Célunk olyan elrendezés kialakítása, ahol a főablak rendelkezik egy felső menürendszerrel, egy eszköztárral, valamint az ablak alján elhelyezett állapotsávval. Az állapotsáv magában foglal egy panelt, amely megjelenít egy

szöveges sort akkor, ha a felhasználó kiválaszt egy menüelemet (vagy eszköztárgombot), míg a menürendszer és az eszköztár grafikusfelület-kapcsolókat kínál fel az alkalmazás bezárásához és a helyesírás-javaslatok megmutatásához egy Expander vezérlőben. A 29.26. ábra mutatja a megcélzott kezdeti elrendezést, megjelenítve a helyesírási javaslatokat a „XAML" kifejezéshez.



29.26. ábra: *Beágyazott panelek használata egy ablak felhasználói felületének létrehozásához*

Figyeljük meg, hogy a két eszköztárgombunk nem egy várt képet, hanem egy egyszerű szöveges értéket jelenít meg. Noha ez nem lenne elegendő egy termékszintű alkalmazáshoz, képek hozzárendelése az eszköztár gombjaihoz általában beágyazott erőforrások használatát igényli, amelynek témakörét a 30. fejezetben fogjuk megvizsgálni (tehát most megteszi a szöveges adat). Azt is vegyük észre, hogy amikor az egér a Check button fölé kerül, a kurzor megváltozik, az állapotsáv egyetlen panele hasznos felhasználóifelület-üzenetet jelenít meg.

A felület készítésének megkezdéséhez módosítsuk a `Window` típusunk kezdeti XAML-definícióját, hogy `<DockPanel>` gyermekelemet használjon `<Grid>` helyett:

```
<Window x:Class="MySpellChecker.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MySpellChecker" Height="331" Width="508"
  WindowStartupLocation ="CenterScreen" >

  <!-- Ez a panel alakítja ki az ablak tartalmát -->
  <DockPanel>
  </DockPanel>
</Window>
```

Windows Formsszal ellentétben - a WPF kezdeti kiadása nem biztosít léptetőgomb-vezérlőelemet, amely lehetővé teszi a felhasználó számára, hogy a fel- és lefelé nyilak segítségével állítson be numerikus értéket. A `RepeatButton` funkcionalitásának köszönhetően viszonylag könnyedén elkészíthetünk egy léptetőgomb-vezérlőt XAML-ben.

Ennek bemutatásához hozzunk létre új Visual Studio WPF-alkalmazás projektet CustomSpinButtonApp néven. Cseréljük le a kezdeti `<Grid>` definíciót olyan `<StackPanel>` típussal, amely az alábbi markupot tartalmazza:

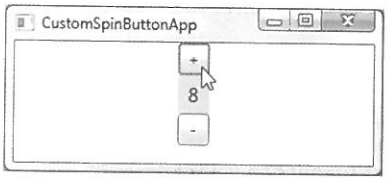
```
<Window x:Class="CustomSpinButtonApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="CustomSpinButtonApp" Height="300" Width="300">
  <StackPanel>
    <!-- A "Fel" gomb  -->
    <RepeatButton Height ="25" Width = "25"
                  Name ="repeatAddValueButton"
                  Delay ="200" Interval ="1"
                  Click ="repeatAddValueButton_Click"
                  Content = "+"/>

    <!-- Megjeleníti az aktuális értéket  -->
    <Label Name ="lblCurrentValue" Background ="LightGray"
           Height ="30" Width = "25"
           VerticalContentAlignment="Center"
           HorizontalContentAlignment="Center" FontSize="15"/>

    <!-- A "Le" gomb -->
    <RepeatButton Height ="25" Width = "25"
                  Name ="repeatRemoveValueButton"
                  Delay ="200" Interval ="1"
                  Click ="repeatRemoveValueButton_Click"
                  Content = "-"/>
  </StackPanel>
</Window>
```

Vegyük észre, hogy az egyes `RepeatButton` típusok egyedi eseménykezelővel hogyan kezelik a kattintási eseményt. Ezzel a következő C#-logikát írhatjuk le, hogy növeljük vagy csökkentsük a `<Label>` címkében megjelenített értéket (nyugodtan adjunk hozzá további logikát, ha szeretnénk rögzíteni a maximális és minimális értéket):

```
public partial class MainWindow : System.Windows.Window
{
  private int currValue = 0;
```

```
    public MainWindow()
    {
      InitializeComponent();
      lblCurrentValue.Content = currValue;
    }
    protected void repeatAddValueButton_Click(object sender,
                                              RoutedEventArgs e)
    {
      // Hozzáad egyet az aktuális értékhez, és megmutatja a címkében.
      currValue++;
      lblCurrentValue.Content = currValue;
    }

    protected void repeatRemoveValueButton_Click(object sender,
                                                 RoutedEventArgs e)
    {
      // Kivon egyet az aktuális értékből, és megmutatja a címkében.
      currValue--;
      lblCurrentValue.Content = currValue;
    }
}
```

Ahogyan láthatjuk, amikor a felhasználó rákattint valamelyik RepeatButton gombra, annak megfelelően növeljük vagy csökkentjük a privát `currValue` értéket, és beállítjuk a `Label` típus `Content` tulajdonságát. A 29.6. ábra mutatja az egyedi léptetőgombunkat működés közben.



29.6. ábra: *Léptetőgomb készítése, kezdőpontként a RepeatButton használatával*

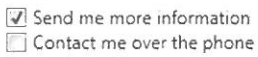
Forráskód A CustomSpinButtonApp kódfájlokat a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A jelölőnégyzetek és a rádiógombok használata

Ahogy korábban említettük, a CheckBox „az-egy" ToggleButton, „az-egy" ButtonBase, ami elég különösnek tűnhet, mivel egy gomb felhasználói felülete meglehetősen különbözik egy jelölőnégyzetétől. Azonban, ha a `CheckBox` tí-

pusra (mint a `Button` típusra) rákattinthatunk, reagál az egér- és billentyűzetbevitelre, valamint követi a WPF-tartalommodellt. A hasonlatosságoknak köszönhetően kiderül, hogy a `CheckBox` típus egyszerűen felülírja a `ToggleButton` különböző virtuális tagjait, hogy létrehozza a jelölőnégyzet megjelenését és működését (jusson eszünkbe, hogy a WPF fő mozgatórugója az, hogy elválasszuk egy vezérlőelem megjelenését annak funkcionalitásától). Tekintsük át a következő `<CheckBox>` deklarációkat, amelyek a 29.7. ábrán látható kimenetet eredményezik:

```
<StackPanel>
  <!-- CheckBox típusok -->
  <CheckBox Name ="checkInfo" >Send me more information</CheckBox>
  <CheckBox Name ="checkPhoneContact" >Contact me over the
                phone</CheckBox>
</StackPanel>
```



29.7. ábra: *Egyszerű CheckBox típusok*

A RadioButton „az-egy" másik `ToggleButton` típus. A `CheckBox` típussal ellentétben azonban rendelkezik azzal a természetes képességgel, amely biztosítja, hogy az ugyanabban a tárolóban (`StackPanel`, `Grid` stb.) lévő összes `RadioButton` típus kölcsönösen kizárólagos, és nem igényel további munkát a részünkről. Vizsgáljuk meg a következőt:

```
<StackPanel>
  <!-- RadioButton típusok zeneválasztáshoz -->
  <Label FontSize = "15" Content = "Select Your Music Media"/>
  <RadioButton>CD Player</RadioButton>
  <RadioButton>MP3 Player</RadioButton>
  <RadioButton>8-Track</RadioButton>

  <!-- RadioButton típusok színválasztáshoz -->
  <Label FontSize = "15" Content = "Select Your Color Choice"/>
  <RadioButton>Red</RadioButton>
  <RadioButton>Green</RadioButton>
  <RadioButton>Blue</RadioButton>
</StackPanel>
```

Ha tesztelnénk ezt a XMAL-t, azt tapasztalnánk, hogy csak egyet választhatunk a hat lehetőség közül, ami valószínűleg nem áll szándékunkban, mivel két különböző csoport van (rádió és szín lehetőségek).

Logikai csoportosítások létrehozása

Ha egyetlen tárolót szeretnénk több `RadioButton` típussal, amelyek különálló fizikai csoportosításként viselkednek, ezt a `GroupName` tulajdonság beállításával tehetjük a `RadioButton` típus nyitó elemén:

```
<StackPanel>
  <!-- A Zene csoport  -->
  <Label FontSize = "15" Content = "Select Your Music Media"/>
  <RadioButton GroupName = "Music" >CD Player</RadioButton>
  <RadioButton GroupName = "Music" >MP3 Player</RadioButton>
  <RadioButton GroupName = "Music" >8-Track</RadioButton>

  <!--A Szín csoport (tetszőlegesen választható ebben a példában,
      lásd a megjegyzést lent) -->
  <Label FontSize = "15" Content = "Select Your Color Choice"/>
  <RadioButton GroupName = "Color">Red</RadioButton>
  <RadioButton GroupName = "Color">Green</RadioButton>
  <RadioButton GroupName = "Color">Blue</RadioButton>
</StackPanel>
```

Ezáltal egymástól függetlenül beállíthatjuk az egyes logikai csoportosításokat, még akkor is, ha azok ugyanabban a fizikai tárolóban vannak.

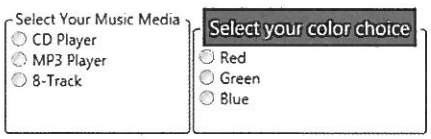
Megjegyzés Alapértelmezés szerint a `GroupName` értékkel nem rendelkező tárolóban lévő összes `RadioButton` egyetlen fizikai csoportként működik. Ezért, az előző példában a színközpontú gombok kölcsönösen kizáróak, még az elhagyott `GroupName` értékkel is, a `Zene` csoport jelenlétének köszönhetően.

A kapcsolódó elemek összeállítása GroupBox típusokba

Amikor rádiógombok vagy jelölőnégyzetek gyűjteményét tervezzük, megszokott dolog vizuális tárolóval körülvenni őket annak jelzésére, hogy csoportként viselkednek. Az általános módszer ehhez a `GroupBox` vezérlőelem használata. Mivel a `Header` tulajdonság alapvetően `System.Object` típussal dolgozik, hozzárendelhetünk bármilyen objektumot (egyszerű sztringet, színes téglalapot, gombot stb.), hogy fejlécként működjön. Vizsgáljuk meg a következő két `GroupBox` deklarációt, amelyek különféle módokon foglalják egységbe az előző `RadioButton` típusokat:

```
<StackPanel>
  <GroupBox Header = "Select Your Music Media" BorderBrush ="Black">
    <StackPanel>
      <RadioButton GroupName = "Music" >CD Player</RadioButton>
      <RadioButton GroupName = "Music" >MP3 Player</RadioButton>
      <RadioButton GroupName = "Music" >8-Track</RadioButton>
    </StackPanel>
  </GroupBox>

  <GroupBox BorderBrush ="Black">
    <GroupBox.Header>
      <Label Background = "Blue" Foreground = "White"
             FontSize = "15" Content = "Select your color choice"/>
    </GroupBox.Header>
    <StackPanel>
      <RadioButton>Red</RadioButton>
      <RadioButton>Green</RadioButton>
      <RadioButton>Blue</RadioButton>
    </StackPanel>
  </GroupBox>
</StackPanel>
```

A kimenet a 29.8. ábrán látható.



29.8. ábra: *RadioButton típusokat keretező GroupBox típusok*

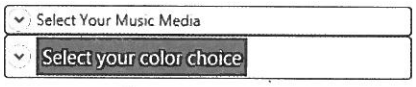
Kapcsolódó elemek bővítő típusokba csoportosítása

A szokásos `GroupBox` típus mellett a WPF rendelkezik olyan új felhasználói felület elemmel, amely csoportosítja olyan felhasználóifelület-elemek gyűjteményét, amelyek egy kapcsoló segítségével elrejthetők, illetve megmutathatók. Ez az elem, az Expander típus lehetővé teszi, hogy az `ExpandDirection` tulajdonság segítségével meghatározzuk az elemek megjelenítésének irányát (fel, le, balra vagy jobbra). Tekintsük meg az alábbi XAML-leírást (amely alapvetően `<GroupBox>` típusról `<Expander>` típusra módosul):

```
<StackPanel>
  <Expander Header = "Select Your Music Media" BorderBrush ="Black">
    <StackPanel>
      <RadioButton GroupName = "Music" >CD Player</RadioButton>
      <RadioButton GroupName = "Music" >MP3 Player</RadioButton>
```

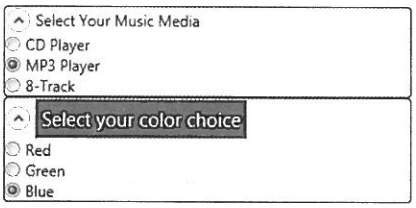
```
        <RadioButton GroupName = "Music" >8-Track</RadioButton>
      </StackPanel>
    </Expander>

    <Expander BorderBrush ="Black">
      <Expander.Header>
        <Label Background = "Blue" Foreground = "White"
               FontSize = "15" Content = "Select your color choice"/>
      </Expander.Header>
      <StackPanel>
        <RadioButton>Red</RadioButton>
        <RadioButton>Green</RadioButton>
        <RadioButton>Blue</RadioButton>
      </StackPanel>
    </Expander >
  </StackPanel>
```

A 29.9. ábra mutatja az egyes Expandereket összecsukott állapotban.



29.9. ábra: *Összecsukott Expanderek*

A 29.10. ábra mutatja az egyes Expandereket kinyitott állapotban.



29.10. ábra: *Kiterjesztett Expanderek*

Forráskód A CheckRadioGroup.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A ListBox és a ComboBox típusok használata

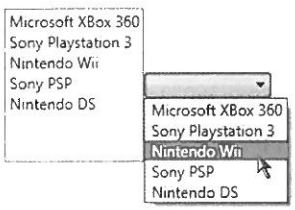
Ahogyan azt remélhettük, a WPF biztosít olyan típusokat, amelyek választható elemek csoportját tartalmazzák – ilyen a `ListBox` és a `ComboBox`, ezek mindegyike az `ItemsControl` absztrakt ősosztályból származik. A legfonto-

sabb, hogy ez a szülőosztály definiálja az `Items` nevű tulajdonságot, amely a részelemeket tároló, erősen típusos `ItemCollection` objektumot adja vissza. Az `ItemCollection` típus a `System.Object` típusokkal dolgozik, ezért gyakorlatilag bármit tartalmazhat. Ha markup révén szeretnénk egyszerű szöveges adatokkal feltölteni egy `ItemsControl`-leszármazott típust, akkor ezt `<ListBoxItem>` típusok készletének használatával tehetjük meg. Példaként nézzük a következő XAML-kódot:

```
<!-- Egyszerű listamező -->
<ListBox Name = "lstVideoGameConsoles">
  <ListBoxItem>Microsoft XBox 360</ListBoxItem>
  <ListBoxItem>Sony Playstation 3</ListBoxItem>
  <ListBoxItem>Nintendo Wii</ListBoxItem>
  <ListBoxItem>Sony PSP</ListBoxItem>
  <ListBoxItem>Nintendo DS</ListBoxItem>
  </ListBox>

<!-- Egyszerű legördülő listadoboz -->
<ComboBox Name = "comboVideoGameConsoles">
  <ListBoxItem>Microsoft XBox 360</ListBoxItem>
  <ListBoxItem>Sony Playstation 3</ListBoxItem>
  <ListBoxItem>Nintendo Wii</ListBoxItem>
  <ListBoxItem>Sony PSP</ListBoxItem>
  <ListBoxItem>Nintendo DS</ListBoxItem>
</ComboBox>
```

Megjegyzés A `ComboBox` típusokat feltölthetjük `<ListBoxItem>` elemek helyett `<ComboBokszItem>` elemek segítségével. Ezáltal hozzáférhetünk az `IsHighlighted` tulajdonsághoz, amelyet a `ListBoxItem` típus nem használ.

Nem meglepő módon a 29.11. ábrán látható renderelést látjuk.



29.11. ábra: *Egyszerű ListBox és ComboBox*

Lista-vezérlőelemek feltöltése programozott módon

A lista-vezérlőelemben lévő adatok gyakran csak futásidőben ismertek; lehet például, hogy a listamező elemeit egy adatbázis olvasásból visszakapott értékek, WFC-szolgáltatás meghívása vagy külső fájl beolvasása alapján kell feltöltenünk. Amikor programozott módon kell `ListBox` vagy `ComboBox` elemet feltöltenünk, egyszerűen használjuk az `ItemCollection` típus tagjait (`Add()`, `Remove()` stb.). Tételezzük fel, hogy új `ListControls` nevű Visual Studio 2008 WPF-alkalmazás projektünk van. A `lstVideoGameConsole` típus előző XAML-deklarációját a következők szerint lehet XAML-ben definiálni:

```
<Window x:Class="ListControls.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="ListControls" Height="300" Width="300" >
  <StackPanel>
    <!-- Ezt kód segítségével töltjük fel -->
    <ListBox Name = "lstVideoGameConsoles">
    </ListBox>
  </StackPanel>
</Window>
```

és a kapcsolódó kódfájlban az alábbiak szerint tölthetjük fel:

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

  private void FillListBox()
  {
    // Elemek hozzáadása a listamezőhöz.
    lstVideoGameConsoles.Items.Add("Microsoft XBox 360");
    lstVideoGameConsoles.Items.Add("Sony Playstation 3");
    lstVideoGameConsoles.Items.Add("Nintendo Wii");
    lstVideoGameConsoles.Items.Add("Sony PSP");
    lstVideoGameConsoles.Items.Add("Nintendo DS");
  }
}
```

Az egyik dolog, amely különösnek tűnhet a ListBox XAML-leírásában, hogy a `<ListBoxItem>` típusokat használtuk az elemek feltöltéséhez; itt azonban sztringtípusokat használtunk az `Add()` metódus meghívása során. Ennek rö-

vid magyarázata az, hogy a XMAL használatakor a `<ListBoxItem>` típusok kényelmesebbek abból a szempontból, hogy azokat a http://schemas.microsoft.com/winfx/2006/xaml/presentation XML-névtérben definiáltuk, ezért rendelkezünk a közvetlen referenciájukkal.

A háttérben a rendszer a `ToString()` metódust hívja minden egyes `<ListBoxItem>` típuson, így a végeredmény azonos. Ha `System.String` névtér használatával töltenénk fel a `ListBox` (vagy `ComboBox`) típust XAML-ben, akkor meg kellene határoznunk egy új XML-névteret, hogy behozzuk az `mscorlib.dll` fájlt (további részletekért lásd a 28. fejezetet):

```
<StackPanel xmlns:CorLib = "clr-namespace:System;assembly=mscorlib">
  <ListBox Name = "lstVideoGameConsoles">
    <CorLib:String>Microsoft XBox 360</CorLib:String>
    <CorLib:String>Sony Playstation 3</CorLib:String>
    <CorLib:String>Nintendo Wii</CorLib:String>
    <CorLib:String>Sony PSP</CorLib:String>
    <CorLib:String>Nintendo DS</CorLib:String>
  </ListBox>
</StackPanel>
```

Ellenben, ha szeretnénk, programozott módon feltölthetünk egy `ItemsControl`-leszármazott típust erősen típusos `ListBoxItem` objektumok segítségével; a jelenlegi példában azonban ezzel nem nyerünk semmit, hanem valójában csak további munkát generálunk magunknak (mivel a `ListBoxItem` nem rendelkezik konstruktorral a `Content` tulajdonság beállításához!).

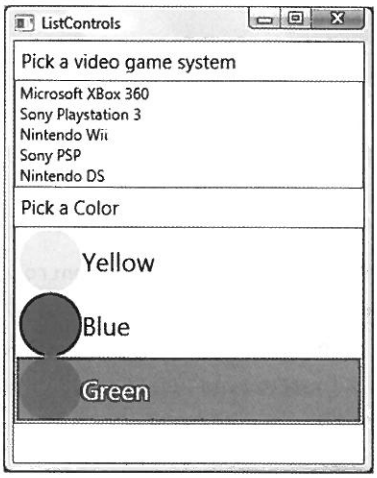
Tetszőleges tartalom hozzáadása

Mivel mind a `ListBox`, mind a `ComboBox` származtatási láncában megtalálható a `ContentControl` alaposztály, ezért azok egy egyszerű sztringnél jóval bonyolultabb adatokat is tartalmazhatnak. Tekintsük át a következő `ComboBox` típust, amely kétdimenziós grafikus objektumokat és egy leíró címkét tartalmazó `<StackPanel>` tárolókat foglal magában:

```
<StackPanel>
  <!-- Egy ListBox tartalommal -->
  <ListBox Name = "lstColors">
    <StackPanel Orientation ="Horizontal">
      <Ellipse Fill ="Yellow" Height ="50" Width ="50"/>
      <Label FontSize ="20" HorizontalAlignment="Center"
             VerticalAlignment="Center">Yellow</Label>
    </StackPanel>
    <StackPanel Orientation ="Horizontal">
```

```
        <Ellipse Fill ="Blue" Height ="50" Width ="50"/>
        <Label FontSize ="20" HorizontalAlignment="Center"
               VerticalAlignment="Center">Blue</Label>
      </StackPanel>
      <StackPanel Orientation ="Horizontal">
        <Ellipse Fill ="Green" Height ="50" Width ="50"/>
        <Label FontSize ="20" HorizontalAlignment="Center"
               VerticalAlignment="Center">Green</Label>
      </StackPanel>
    </ListBox>
  </StackPanel>
```

A 29.12. ábra mutatja az aktuális lista típusaink kimenetét.



29.12. ábra: *ItemsControl-leszármazott típusok tartalmazhatnak bármilyen jellegű tartalmat*

Az aktuális kiválasztás meghatározása

A `ListBox` vagy a `ComboBox` típus feltöltése után a következő nyilvánvaló kérdés az, hogy futásidőben hogyan lehet eldönteni, melyik elemet választotta ki a felhasználó. Mint látni fogjuk, erre három módszer létezik. Ha a kiválasztott elem numerikus indexének a kiderítése érdekel bennünket, akkor használhatjuk a `SelectedIndex` tulajdonságot (amely nulla alapú; a `-1` érték azt mutatja, hogy nincs kijelölés). Ha a listában a kiválasztott objektumot szeretnénk megszerezni, arra a `SelectedItem` tulajdonság való. Végül, a `SelectedValue` lehetővé teszi, hogy a kiválasztott objektum értékét megkapjuk (többnyire a `ToString()` metódus meghívásával).

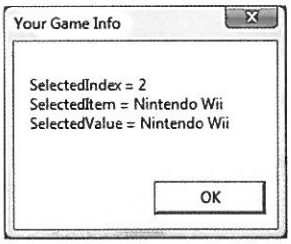
Nem hangzik bonyolultnak, igaz? Most, hogy teszteljük az egyes tulajdonságok viselkedését, tételezzük fel, hogy definiáltunk két új `Button` típust az aktuális ablak számára, amelyek egyaránt kezelik a kattintási eseményt:

```
<!-- Gombok a kiválasztott elem megszerzéséhez -->
<Button Name ="btnGetGameSystem" Click ="btnGetGameSystem_Click">
  Get Video Game System
</Button>
<Button Name ="btnGetColor" Click ="btnGetColor_Click">
  Get Color
</Button>
```

A `btnGetGameSystem` esetében a kattintási esemény kezelője megszerzi a `lstVideoGameConsoles` objektum `SelectedIndex`, `SelectedItem` és `SelectedValue` tulajdonságainak értékeit, és megjeleníti azokat egy üzenetdobozban:

```
protected void btnGetGameSystem_Click(object sender,
                                      RoutedEventArgs args)
{
  string data = string.Empty;
  data += string.Format("SelectedIndex = {0}\n",
          lstVideoGameConsoles.SelectedIndex);
  data += string.Format("SelectedItem = {0}\n",
          lstVideoGameConsoles.SelectedItem);
  data += string.Format("SelectedValue = {0}\n",
          lstVideoGameConsoles.SelectedValue);
  MessageBox.Show(data, "Your Game Info");
}
```

Ha a „Nintendo Wii"-t választanánk a játékkonzolok listájából, és a kapcsolódó gombra kattintanánk, akkor a 29.13. ábrán látható szövegdobozt kapnánk.

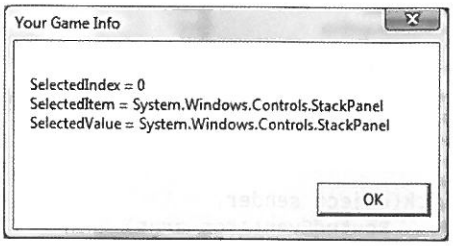


29.13. ábra: *Megtaláltuk a kiválasztott szringet*

Azonban, mi a helyzet a kiválasztott szín megszerzésével?

Az aktuális kiválasztás meghatározása a beágyazott tartalom esetében

Tételezzük fel, hogy a `btnGetColor` `Button` típus kattintási eseményének kezelője implementálja a `btnGetColor_Click()` metódust, hogy kiírja a `lstColors` `ListBox` objektum aktuális kijelölését, indexét és értékét. Ha most kiválasztanánk az első elemet a lstColors listamezőből (és a hozzátartozó gombra kattintanánk), meglepetésünkre a 29.14. ábrán látható kimenettel szembesülnénk.



29.14. ábra: *Megtaláltuk a kiválasztott... StackPanelt?*

Ennek a kimenetnek az oka az, hogy a `lstColors` objektum három `StackPanel` objektumot tart fenn, amelyek mindegyike rendelkezik beágyazott tartalommal. Ezért a `SelectedItem` és a `SelectedValue` tulajdonságok egyszerűen meghívják a `ToString()` metódust a `StackPanel` típuson, amely visszaadja teljesen meghatározott nevét.

Noha a `SelectedIndex` tulajdonságból visszaadott numerikus érték alapján könnyen kitalálhatnánk, hogy a felhasználó melyik elemet választotta, egy másik megközelítés az, hogy leásunk a `StackPanel` gyermekgyűjteményébe, hogy megkapjuk a `Label` típus `Content` értékét a `StackPanel` belsőleg fenntartott `Children` gyűjteményének segítségével:

```
protected void btnGetColor_Clicked(object sender,
                                   RoutedEventArgs args)
{
  // A Content érték megszerzése a StackPanel elemben
  // a kiválasztott Label típusban.
  StackPanel selectedStack =
             (StackPanel)lstColors.Items[lstColors.SelectedIndex];
  string color =
         ((Label)(selectedStack.Children[1])).Content.ToString();
  string data = string.Empty;
  data += string.Format("SelectedIndex = {0}\n",
          lstColors.SelectedIndex);
  data += string.Format("Color = {0}", color);
  MessageBox.Show(data, "Your Game Info");
}
```

Noha ez a megoldás megteszi, mégis elég kényes, mivel kódolt pozíciók vannak a `StackPanel` tárolóban (a második gyermek a `Label`), és számos kasztolási műveletet kell végrehajtanunk. Másik alternatíva minden `StackPanel` Tag tulajdonságának beállítása, amelyet a `FrameworkElement` alaposztályban definiálunk:

```
<ListBox Name = "lstColors">
  <StackPanel Orientation ="Horizontal" Tag ="Yellow">
  ...
  </StackPanel>
  <StackPanel Orientation ="Horizontal" Tag ="Blue">
  ...
  </StackPanel>
  <StackPanel Orientation ="Horizontal" Tag ="Green">
  ...
  </StackPanel>
</ListBox>
```

Ezen megközelítés használatával a kódunk jelentősen letisztul, mivel programozott módon kiszedhetjük a Tag tulajdonsághoz rendelt értéket, a következők szerint:

```
protected void btnGetColor_Clicked(object sender,
                                   RoutedEventArgs args)
{
  string data = string.Empty;
  data += string.Format("SelectedIndex = {0}\n",
          lstColors.SelectedIndex);
  data += string.Format("SelectedItem = {0}\n",
          lstColors.SelectedItem);
  data += string.Format("SelectedValue = {0}",
          (lstColors.Items[lstColors.SelectedIndex] as
          StackPanel).Tag);
  MessageBox.Show(data, "Your Color Info");
}
```

Noha ez a megközelítés egy kicsit tisztább, mint az első kísérletünk, vannak más módok is, ahol *adatsablonok* segítségével érhetjük el egy összetett vezérlőelem értékét. Ehhez meg kell értenünk a WPF adatkötési motor működését, amelyet a fejezet végén vizsgálunk meg.

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

A többsoros szövegbeviteli mezők használata

A WPF több olyan felhasználóifelület-elemet biztosít, amelyek lehetővé teszik a szövegalapú bevitelek összegyűjtését. A két legegyszerűbb típus a `TextBox` és a `PasswordBox`, amelyeket most a `TextControls` nevű, új Visual Studio 2008 WPF alkalmazás használatával vizsgálunk meg.

A TextBox típus használata

A múltban használt más `TextBox` típusokhoz hasonlóan a WPF `TextBox` típusát beállíthatjuk, hogy egy sornyi szöveget tartalmazzon (ez az alapértelmezett beállítás), vagy több sort, ha az `AcceptReturn` tulajdonság értéke `true`. A `TextBox`on belül az adatokat mindig karakteradatokként kezeljük, ezért a „tartalom" mindig olyan sztringtípus, amelyet a Text tulajdonság segítségével beállíthatunk és lekérdezhetünk:

```
<TextBox Name ="txtData" Text = "Hello!" BorderBrush ="Blue"
         Width ="100"/>
```

A WPF `TextBox` típusának egyik nagyon egyedi jellemzője, hogy beépített képessége révén ellenőrizheti a bevitt adatok helyesírását, ha a `SpellCheck.IsEnabled` tulajdonság értéke `true`. Ilyen esetekben látni fogjuk, hogy – a Microsoft Office-hoz hasonlóan – a helytelenül írt szavakat az alkalmazás vörös hullámos vonallal aláhúzza. Ennél még jobb, hogy az alapul szolgáló programozási modell segítségével hozzáférhetünk a helyesírásellenőrző-motorhoz, ezáltal megkaphatjuk a helytelenül írt szavakra vonatkozó javaslatok listáját.

Az alábbiak szerint módosítsuk az aktuális ablak XAML-definícióját, hogy használhassuk a `Label`, a `TextBox` és a `Button` típusokat (figyeljük meg, hogy a `TextBox` többsoros szövegeket támogat, és engedélyezi a helyesírás-ellenőrzést):

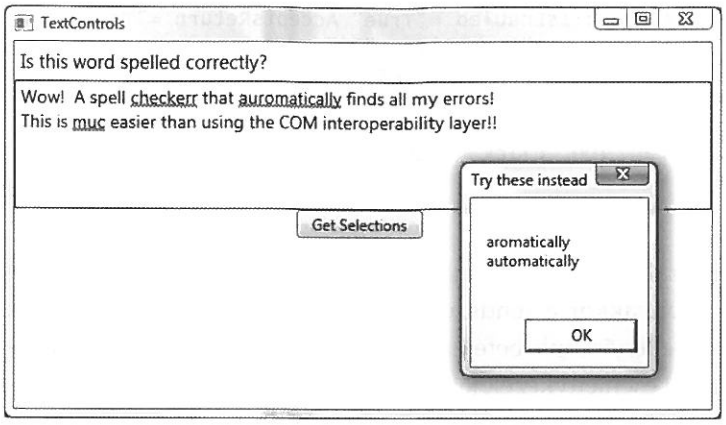
```
<Window x:Class="TextControls.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="TextControls" Height="204" Width="292" >
  <StackPanel>
    <Label FontSize ="15">Is this word spelled correctly?</Label>
```

```
    <TextBox SpellCheck.IsEnabled ="True" AcceptsReturn ="True"
             Name ="txtData" FontSize ="12"
             BorderBrush ="Blue" Height ="100">
    </TextBox>
    <Button Name ="btnOK" Content ="Get Selections"
            Width = "100" Click ="btnOK_Click"/>
  </StackPanel>
</Window>
```

Ilyen funkcionalitással már látjuk majd, hogy amikor helytelenül írunk be szavakat a `TextBox`-ba, akkor a rendszer megjelöli a helyesírási hibákat. Az egyszerű helyesírás-ellenőrzőnk befejezéséhez a következőképpen módosítsuk a gomb kattintási eseménykezelőjét:

```
protected void btnOK_Click(object sender, RoutedEventArgs args)
{
  string spellingHints = string.Empty;

  // Próbáljunk helyesírási hibát keresni a kurzor aktuális
  // helyénél.
  SpellingError error =
                txtData.GetSpellingError(txtData.CaretIndex);
  if (error != null)
  {
    // Készítsük el a helyesírási javaslatok sztringjét.
    foreach (string s in error.Suggestions)
    {
      spellingHints += string.Format("{0}\n", s);
    }

    // Mutassuk meg a javaslatokat!
    MessageBox.Show(spellingHints, "Try these instead");
  }
}
```

A kód meglehetősen egyszerű. Csak megkeressük a kurzor jelenlegi helyét a szövegdobozban a `CaretIndex` tulajdonság használatával, hogy kinyerjünk egy `SpellingError` objektumot. Ha valami hiba van a mondott helyen (vagyis az érték nem `null`), akkor végiglépkedünk a javaslatok listáján a stílszerűen elnevezett `Suggestions` tulajdonság révén. Végül megjelenítjük a lehetőségeket egy egyszerű `MessageBox.Show()` kérés használatával. A 29.15. ábra lehetséges tesztfutást mutat, amikor a kurzor a helytelenül írt „auromatically" szón áll.



29.15. ábra: *Egy egyedi helyesírás-ellenőrző!*

A PasswordBox típus használata

A `PasswordBox` típus – nem meglepően – egy olyan biztonságos helyet kínál, ahova érzékeny szöveges adatokat írhatunk be. Alapértelmezés szerint a jelszó karakterkör típusú, ez azonban megváltoztatható a `PasswordChar` tulajdonság használatával. A végfelhasználó által beírt érték megszerzéséhez egyszerűen ellenőrizzük a `Password` tulajdonságot. Módosítsuk a jelenlegi helyesírás-ellenőrző alkalmazásunkat a helyes jelszó kérésével, hogy láthassuk a helyesírási javaslatok listáját. Először módosítsuk a már meglévő `<StackPanel>` tárolót egy beágyazott `<StackPanel>` típussal, amely vízszintesen elhelyezi a meglévő `<Button>` típus mellé a `PasswordBox` mezőt:

```
<StackPanel>
  <Label FontSize ="15">Is this word spelled correctly?</Label>
  <TextBox SpellCheck.IsEnabled ="True" AcceptsReturn ="True"
           Name ="txtFavoriteColor" FontSize ="14"
           BorderBrush ="Blue" Height ="100">
  </TextBox>
  <StackPanel Orientation ="Horizontal">
    <PasswordBox Name ="pwdText" BorderBrush ="Black"
                 Width ="100"></PasswordBox>
    <Button Name ="btnOK" Content ="Get Selections"
            Width = "100" Click ="btnOK_Click"/>
  </StackPanel>
</StackPanel>
```

Most módosítsuk úgy a jelenlegi `Button` kattintási eseménykezelőjét, hogy meghívja a `CheckPassword()` segédfüggvényt, amely teszteli a kódolt sztringet. Győződjünk meg arról, hogy a javaslatok csak akkor jelenhessenek meg, ha a jelszó ellenőrzése sikeres volt. Íme a releváns módosítások:

```
public partial class MainWindow : System.Windows.Window
{
  ...
  protected void btnOK_Click(object sender, RoutedEventArgs args)
  {
    if (CheckPassword())
    {
      // Ugyanaz a helyesírás-ellenőrző logika, mint korábban...
    }
    else
      MessageBox.Show("Security error!!");
  }

  private bool CheckPassword()
 {
    if (pwdText.Password == "Chucky")
    return true;
  else
    return false;
  }
}
```

A `TextBox` és a `PasswordBox` mellett ne feledjük, hogy ha egy olyan alkalmazást készítünk, amely rendelkezik olyan szövegmezővel, amely bármilyen típusú tartalmat (grafikus renderelést, szöveget stb.) magában foglalhat, a WPF biztosítja számunkra a `RichTextBox` típust. Emellett, ha kell a teljesítmény egy különösen szövegintenzív alkalmazás készítéséhez, a WPF biztosít egy teljes dokumentumprezentációs API-t, amelyet elsősorban a `System.Windows.Documents` névtér képvisel.

Itt találunk olyan típusokat, amelyek lehetővé teszik *folyószöveg-dokumentumok* (flow document) készítését, és ezek segítségével programozott módon ábrázolhatunk (XAML- vagy C#-kódban) bekezdéseket, kapcsolódó szövegblokk-részeket, feljegyzéseket, jegyzeteket, táblázatokat és egyéb gazdag dokumentumcentrikus típusokat. A könyv jelen kiadása nem tárgyalja a `RichTextBox` típust, illetve a folyószöveg-dokumentum API-t; ha érdekesnek bizonyul a téma, további részletekért lapozzuk fel a .NET Framework 3.5 SDK dokumentációját.

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

Ezzel befejeztük a WPF vezérlőelemkészlet áttekintését. A fejezet későbbi részében látni fogjuk, hogyan készíthetünk menürendszereket, állapotsávokat és eszköztárakat. A következő feladatunk azonban annak áttekintése, hogyan rendezhetjük a felhasználói felület elemeit `Window` típuson belül tetszőleges számú paneltípus használatával.

A tartalomelrendezés kezelése panelek használatával

Egy valós WPF-alkalmazás mindenképpen tartalmaz jó néhány felhasználóifelület-elemet (beviteli vezérlőelemeket, grafikus tartalmat, menürendszereket, állapotsávokat stb.), amelyeket jól kell elrendezni a tároló ablakban. Emellett, ha egyszer elhelyeztük a felhasználói felület vezérlőit az új helyükre, biztosak akarunk lenni abban, hogy az elvártak szerint viselkednek, amikor a végfelhasználó átméretezi az ablakot vagy akár az ablak egy részét (osztott ablak esetében). Ahhoz, hogy biztosítsuk, a WPF-vezérlőelemek megtartsák pozíciójukat a hosztoló ablakban, több *paneltípus* áll rendelkezésünkre.

Ahogy az előző fejezetből emlékezhetünk rá, hogy amikor tartalmat helyezünk el olyan ablakban, amely nem használ paneleket, az hajszálpontosan a tároló közepére kerül. Tekintsük át az alábbi egyszerű ablakdeklarációt, amely egyetlen `Button` típust tartalmaz. Függetlenül attól, hogyan méretezzük át az ablakot, a felhasználóifelület-vezérlő mindig egyenlő távolságra jelenik meg az adott terület négy oldalától.

```
<!-- Ez a gomb mindig az ablak közepén van -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">
  <Button Name="btnOK" Height = "100" Width="80">OK</Button>
</Window>
```

Arra is emlékezzünk, hogy ha több elemet próbálunk elhelyezni közvetlenül egy `<Window>` típus hatókörén belül, akkor címkézési és/vagy fordítási idejű hibát kapunk. Ezeknek a hibáknak az az oka, hogy egy ablak (vagy ami azt illeti, a `ContentControl` bármelyik leszármazottja) csak egyetlen objektumot rendelhet a `Content` tulajdonságához:

```
<!-- Hiba! A Content tulajdonságot implicit módon többször
     állítottuk be! -->
<Window x:Class="MyWPFApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">

  <!-- Nyugtázva! A <Window> két közvetlen gyermekeleme! -->
  <Label Name="lblInstructions"
         Width="328" Height="27" FontSize="15">Enter
         Car Information</Label>
  <Button Name="btnOK" Height = "100" Width="80">OK</Button>
</Window>
```

Nyilvánvaló, hogy nem sok haszna van egy olyan ablaknak, amely csak egy elemet tartalmazhat. Ha arra van szükség, hogy egy ablak több elemet tartalmazzon, akkor azokat tetszőleges számú panelbe kell rendeznünk. A panel tartalmazza az ablakot képviselő felhasználóifelület-elemeket, amely után a panelt magát csak a `Content` tulajdonsághoz rendelt objektumként használjuk.

A WPF alapvető paneltípusai

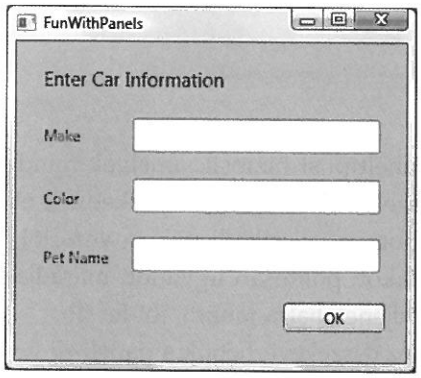
A `System.Windows.Controls` névtér több paneltípust biztosít, amelyek mindegyike azt szabályozza, hogy az egyes részelemek hogyan helyezkednek el. A panelek használatával megadhatjuk, hogyan viselkedjenek a vezérlők, amikor a végfelhasználó átméretezi az ablakot: pontosan ugyanott maradjanak, ahova tervezéskor kerültek; átrendeződjenek balra-jobbra, föl-le; stb.

Bonyolult felhasználói felület készítéséhez összekeverhetjük a panelbeli vezérlőelemeket (pl. egy `StackPanelt` tartalmazó `DockPanel`), ezáltal nagyfokú rugalmasságot és kezelhetőséget biztosíthatunk. Ezenfelül a paneltípusok együtt tudnak dolgozni más dokumentumközpontú vezérlőelemekkel (mint a `ViewBox`, a `TextBlock`, a `TextFlow` és a `Paragraph` típusok), hogy tovább lehessen finomítani a tartalom elrendezését egy adott panelen belül. A 29.3. táblázat bemutatja néhány gyakran használt WPF panelbeli vezérlőelem szerepét.

Panelbeli vezérlőelem	Jelentés
`Canvas`	„Klasszikus" mód a tartalom elhelyezésére. Az elemek pontosan ugyanott maradnak, ahová a tervezéskor tettük őket.
`DockPanel`	Zárolja a tartalmat a panel megadott oldalára (`Top`, `Bottom`, `Left` vagy `Right`).
`Grid`	Cellák sorozatába rendezi a tartalmat, táblázatos rácsformában.
`StackPanel`	Függőlegesen vagy vízszintesen egymás után helyezi a tartalmat, ahogy azt az `Orientation` tulajdonság megkívánja.
`WrapPanel`	Balról jobbra pozícionálja a tartalmat, új sorba tördelve azt a tároló doboz szélénél. A további rendezés folyamatosan történik felülről lefelé vagy jobbról balra, az `Orientation` tulajdonság értékétől függően.

29.3. táblázat: *Alapvető WPF panelbeli vezérlőelemek*

A következő részben bemutatjuk ezeknek a gyakran használt paneltípusoknak a használatát, elkészítjük a 29.16. ábrán látható grafikus felületet különböző paneleken belül, és megvizsgáljuk, hogyan változik az elhelyezés, ha az ablakot átméretezzük.



29.16. ábra: *A felhasználói felület megcélzott elrendezése*

Tartalom elhelyezése a Canvas paneleken belül

Vitathatatlanul a legegyszerűbb panel a Canvas. A Canvas az a panel, amelyben leginkább otthon érezzük magunkat, ugyanis egy Windows Forms alkalmazás alapértelmezett elrendezését utánozza. A Canvas panel lehetővé teszi a

felhasználói felület tartalmának abszolút elhelyezését. Ha a végfelhasználó a Canvas panelen fenntartott elrendezésnél kisebb területűre méretezi az ablakot, a belső tartalom addig nem lesz látható, amíg a tárolót ugyanakkorára vagy nagyobbra nem terjeszti ki, mint a Canvas területe.

Ahhoz, hogy a `Canvas` típushoz tartalmat adjunk, határozzuk meg meg a szükséges részelemeket a nyitó `<Canvas>` és a záró `</Canvas>` címkék hatókörén belül, valamint adjuk meg a renderelés helyét (jegyezzük meg, hogy a tartalom pozíciója relatív lehet a Canvas jobb/bal vagy alsó/felső oldalához képest, de nem mindkettőhöz). Ha azt szeretnénk, hogy a Canvas területe a tároló egész területét fedje, egyszerűen hagyjuk el a `Height` és a `Width` tulajdonságokat. Tekintsük meg a következő XAML-markupot, amely a 29.16. ábrán látható elrendezést definiálja:

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fun with Panels!" Height="285" Width="325">

  <Canvas Background="LightSteelBlue">
    <Button Canvas.Left="212" Canvas.Top="203" Name="btnOK"
            Width="80">OK</Button>
    <Label Canvas.Left="17" Canvas.Top="14" Name="lblInstructions"
           Width="328" Height="27"
           FontSize="15">Enter Car Information</Label>
    <Label Canvas.Left="17" Canvas.Top="60"
           Name="lblMake">Make</Label>
    <TextBox Canvas.Left="94" Canvas.Top="60" Name="txtMake"
             Width="193" Height="25"/>
    <Label Canvas.Left="17" Canvas.Top="109"
           Name="lblColor">Color</Label>
    <TextBox Canvas.Left="94" Canvas.Top="107" Name="txtColor"
             Width="193" Height="25"/>
    <Label Canvas.Left="17" Canvas.Top="155"
           Name="lblPetName">Pet Name</Label>
    <TextBox Canvas.Left="94" Canvas.Top="153" Name="txtPetName"
             Width="193" Height="25"/>

  </Canvas>

</Window>
```

Ebben a példában a `<Canvas>` hatókörén belül lévő összes elemet egy `Canvas.Left` és egy `Canvas.Top` érték minősít, amelyek a tartalom felső és bal oldali elhelyezkedését vezérlik a panelen belül, a csatolt tulajdonságszintaxis használatával (lásd a 28. fejezetet). Ahogy kitalálhattuk, a függőleges elhelyezkedést a Top, illetve a `Bottom` tulajdonság, míg a vízszintes elhelyezkedést a `Left`, illetve a `Right` tulajdonság vezérli.

Mivel minden vezérlőt a <Canvas> elemen belül helyeztünk el, látni fogjuk, hogy az ablak átméretezésekor a rendszer eltakarja a vezérlőket, ha a tároló területe kisebb lesz, mint a tartalom (lásd a 29.17. ábrát).

29.17. ábra: *A Canvas panel tartalma lehetővé teszi az abszolút pozícionálást*

Annak a sorrendje, hogyan deklarálunk tartalmat egy Canvas típuson belül, befolyásolja az elhelyezkedés kiszámítását, ugyanis ez a vezérlőelem méretén, valamint a Canvas.Top, a Canvas.Bottom, a Canvas.Left és a Canvas.Right tulajdonságokon alapul. Ezek alapján az alábbi markup (amely csoportosítja a hasonló vezérlőelemeket) ugyanolyan renderelést eredményez:

```
<Canvas Background="LightSteelBlue">
  <TextBox Canvas.Left="94" Canvas.Top="153" Name="txtColor"
           Width="193" Height="25"/>
  <TextBox Canvas.Left="94" Canvas.Top="60" Name="txtPetName"
           Width="193" Height="25"/>
  <TextBox Canvas.Left="94" Canvas.Top="107" Name="txtMake"
           Width="193" Height="25"/>
  <Label Canvas.Left="17" Canvas.Top="14" Name="lblInstructions"
         Width="328" Height="27"
         FontSize="15">Enter Car Information</Label>
  <Label Canvas.Left="17" Canvas.Top="109"
         Name="lblColor">Color</Label>
  <Label Canvas.Left="17" Canvas.Top="155"
         Name="lblMake">Pet Name</Label>
  <Label Canvas.Left="17" Canvas.Top="60"
         Name="lblPetName">Make</Label>

  <Button Canvas.Left="212" Canvas.Top="203" Name="btnOK"
          Width="80">OK</Button>
</Canvas>
```

Megjegyzés Ha egy Canvas típuson belüli részelemek nem határoznak meg adott helyet a csatolt tulajdonságszintaxis révén, akkor azok automatikusan a bal felső sarokba kerülnek.

Noha a Canvas típus használata kívánatosnak tűnhet a tartalom elrendezéséhez (mivel olyan ismerősnek tűnik), mégis vannak korlátai. Először is, egy Canvas típuson belüli elemek nem méretezik át magukat dinamikusan stílusok vagy sablonok alkalmazásakor (pl. a betűk mérete nem változik). A másik nyilvánvaló korlát az, hogy a Canvas nem próbálja meg láthatóan megtartani az elemeket, amikor a végfelhasználó kisebbre méretezi át az ablakot.

A Canvas típus legjobb használata talán a grafikus tartalom elhelyezése. Például, ha egyedi képet készítenénk XAML használatával, akkor bizonyára azt akarnánk, hogy a vonalak, a formák és a szöveg ugyanazon a helyen maradjon ahelyett, hogy dinamikusan áthelyeződjenek, ha a felhasználó átméretezi az ablakot. Újra megvizsgáljuk a Canvas típust a következő fejezetben, amikor a WPF grafikus renderelési szolgáltatásairól lesz szó.

Forráskód A SimpleCanvas.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

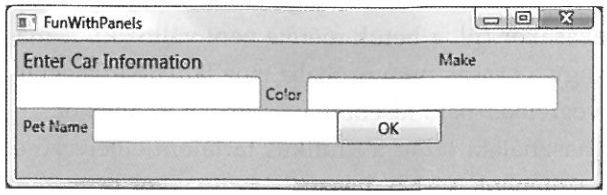
Tartalom elhelyezése a WrapPanel paneleken belül

A WrapPanel lehetővé teszi olyan tartalom definiálását, amely átnyúlik a panelen, ha az ablakot átméretezik. Amikor elhelyezünk elemeket egy WrapPanel panelben, nem kell megadnunk felső, alsó, bal, illetve jobb oldali elhelyezési értékeket, mint általában a Canvas esetében tesszük. Ehelyett minden részelem szabadon meghatározhat egy Height és egy Width értéket (egyéb tulajdonságértékek mellett), hogy szabályozza az általános méretét a tárolóban.

Mivel a WrapPanelen belül a tartalom nem „rögzül" a panel egy adott oldalához, ezért fontos a sorrend, amelyben az elemeket deklaráljuk (a tartalom renderelése az első elemtől az utolsóig történik). Nézzük meg a következő XAML-kódrészletet:

```
<WrapPanel Background="LightSteelBlue">
  <Label Name="lblInstruction" Width="328"
         Height="27" FontSize="15">Enter Car Information</Label>
  <Label Name="lblMake">Make</Label>
  <TextBox Name="txtMake" Width="193" Height="25"/>
  <Label Name="lblColor">Color</Label>
  <TextBox Name="txtColor" Width="193" Height="25"/>
  <Label Name="lblPetName">Pet Name</Label>
  <TextBox Name="txtPetName" Width="193" Height="25"/>
  <Button Name="btnOK" Width="80">OK</Button>
</WrapPanel>
```

Amikor megtekintjük ezt a markupot, a tartalom változtatja a formáját a szélesség átméretezésekor, mivel balról jobbra halad az ablakban (lásd a 29.18. ábrát).



29.18. ábra: *Egy WrapPanel panelben a tartalom úgy viselkedik, mint egy közönséges HTML-oldal*

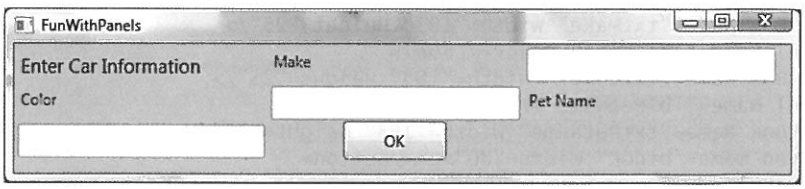
Alapértelmezés szerint a WrapPanel tartalma balról jobbra halad. Ha azonban megváltoztatjuk az `Orientation` tulajdonság értékét `Vertical`-ra, a tartalom fentről lefelé halad:

```
<WrapPanel Background="LightSteelBlue" Orientation ="Vertical">
```

A WrapPanel (valamint néhány más paneltípus) deklarálható az `ItemWidth` és az `ItemHeight` értékek meghatározásával, amelyek az egyes elemek alapértelmezett méretét szabályozzák. Ha egy részelem megadja a saját `Height` és/vagy `Width` értékét, akkor az a panel által megállapított mérethez viszonyítva helyezzük el. Tekintsük meg a következő markupot:

```
<WrapPanel Background="LightSteelBlue" ItemWidth ="200"
           ItemHeight ="30">
  <Label Name="lblInstruction"
         FontSize="15">Enter Car Information</Label>
  <Label Name="lblMake">Make</Label>
  <TextBox Name="txtMake"/>
  <Label Name="lblColor">Color</Label>
  <TextBox Name="txtColor"/>
  <Label Name="lblPetName">Pet Name</Label>
  <TextBox Name="txtPetName"/>
  <Button Name="btnOK" Width ="80">OK</Button>
</WrapPanel>
```

Renderelés után a 29.19. ábrán látható kimenetet kapjuk (figyeljük meg a Button vezérlő méretét és pozícióját).



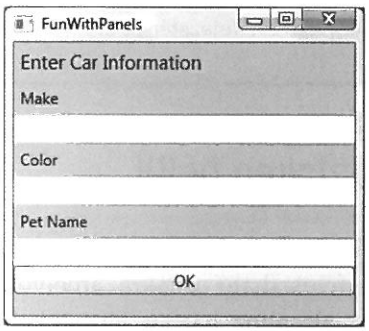
29.19. ábra: *A WrapPanel megállapíthatja egy adott elem szélességét és magasságát*

A 29.19. ábra megtekintése után egyetérthetünk abban, hogy a WrapPanel általában nem a legjobb választás tartalom közvetlen elrendezésére egy ablakban, mivel az elemek összekeveredhetnek, ha a felhasználó átméretezi az ablakot. A WrapPanel a legtöbb esetben egy másik paneltípus részeleme lesz, és lehetővé teszi az ablak egy kis területének, hogy átméretezéskor becsomagolja a tartalmát.

Forráskód A SimpleWrapPanel.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Tartalom elhelyezése a StackPanel paneleken belül

A WrapPanelhez hasonlóan a StackPanel vezérlőelem egy sorba rendezi a tartalmat, amely vízszintesen vagy függőlegesen igazítható (ez az alapértelmezett) az `Orientation` tulajdonsághoz rendelt érték alapján. A különbség az, hogy a StackPanel *nem* kísérli meg a tartalom becsomagolását, amikor a felhasználó átméretezi az ablakot. Ehelyett, a StackPanel panelben lévő elemek egyszerűen megnyúlnak (a tájolásuk alapján), hogy illeszkedjenek a StackPanel méretéhez.



29.20. ábra: *A tartalom függőleges egymás után helyezése*

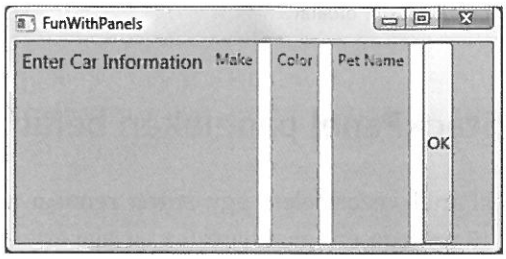
A következő markup például a 29.20. ábrán látható kimenetet eredményezi:

```
<StackPanel Background="LightSteelBlue">
  <Label Name="lblInstruction"
         FontSize="15">Enter Car Information</Label>
  <Label Name="lblMake">Make</Label>
  <TextBox Name="txtMake"/>
  <Label Name="lblColor">Color</Label>
  <TextBox Name="txtColor"/>
```

```
    <Label Name="lblPetName">Pet Name</Label>
    <TextBox Name="txtPetName"/>
    <Button Name="btnOK">OK</Button>
  </StackPanel>
```

Ha az alábbiak szerint `Horizontal` értékre változtatjuk az `Orientation` tulajdonságot, akkor a renderelt kimenet a 29.21. ábrán láthatóval egyezik:

```
<StackPanel Background="LightSteelBlue" Orientation ="Horizontal">
```



29.21. ábra: *A tartalom vízszintes egymás mellé helyezése*

Ugyanúgy, mint a WrapPanel esetében, ritkán használjuk majd a `StackPanel` típust arra, hogy közvetlenül rendezzük el a tartalmat egy ablakon belül. A StackPanel ehelyett jobban alkalmazható egy főpanel részpaneleként.

Forráskód A SimpleStackPanel.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Tartalom elhelyezése a Grid paneleken belül

A WPF API-kon belül kínált panelek közül a Grid messze a legrugalmasabb. A HTML-táblázatokhoz hasonlóan a `Grid` típus feloszthatóra cellákra, amelyek mindegyikében lehet tartalom. Egy `Grid` definiálásához három lépést kell végrehajtanunk:

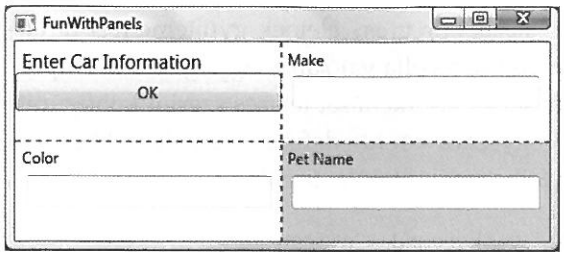
1. Az egyes oszlopok meghatározása és konfigurálása.
2. Az egyes sorok meghatározása és konfigurálása.
3. Tartalom hozzárendelése a rács minden cellájához a csatolt tulajdonságszintaxis használatával.

Megjegyzés Ha nem határozunk meg sorokat vagy oszlopokat, akkor a <Grid> alapértelmezés szerint egyetlen cella, amely kitölti az ablak teljes felületét. Továbbá, ha nem rendelünk hozzá cellaértéket egy részelemhez a <Grid> típuson belül, akkor az automatikusan a 0. oszlop 0. sorába kerül.

Az első két lépést (az oszlopok és a sorok definiálását) a <Grid.ColumnDefinitions> és a <Grid.RowDefinitions> elemek használatával érhetjük el, amelyek a <ColumnDefinition>, illetve a <RowDefinition> elemek gyűjteményét tartalmazzák. Mivel egy rácson belül minden cella valójában egy igazi .NET-típus, ezért az egyes elemek megjelenését és viselkedését tetszés szerint konfigurálhatjuk. Íme, egy meglehetősen egyszerű <Grid> definíció, amely a 29.22. ábrán látható módon rendezi el a felhasználói felület tartalmát:

```
<Grid ShowGridLines ="True" Background ="AliceBlue">
  <!-- A sorok/oszlopok meghatározása -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
  </Grid.RowDefinitions>

  <!-- Most hozzáadjuk az elemeket a rács celláihoz -->
  <Label Name="lblInstruction" Grid.Column ="0" Grid.Row ="0"
         FontSize="15">Enter Car Information</Label>
  <Button Name="btnOK" Height ="30" Grid.Column ="0"
          Grid.Row ="0" >OK</Button>
  <Label Name="lblMake" Grid.Column ="1" Grid.Row ="0">Make</Label>
  <TextBox Name="txtMake" Grid.Column ="1" Grid.Row ="0"
           Width="193" Height="25"/>
  <Label Name="lblColor" Grid.Column ="0"
         Grid.Row ="1" >Color</Label>
  <TextBox Name="txtColor" Width="193" Height="25" Grid.Column ="0"
           Grid.Row ="1" />

  <!-- Hogy érdekesebb legyen, adjunk valamilyen színt a PetName
       cellának -->
  <Rectangle Fill ="LightGreen" Grid.Column ="1" Grid.Row ="1" />
  <Label Name="lblPetName" Grid.Column ="1"
         Grid.Row ="1" >Pet Name</Label>
  <TextBox Name="txtPetName" Grid.Column ="1" Grid.Row ="1"
           Width="193" Height="25"/>
</Grid>
```

Figyeljük meg, hogy minden elem (beleértve a ráadásként hozzáadott világoszöld `Rectangle` elemet is) a rács egy cellájához kapcsolja magát a `Grid.Row` és a `Grid.Column` csatolt tulajdonságok segítségével. Alapértelmezés szerint a rácsban lévő cellák rendezése a bal felső sarokban kezdődik, amelyet a `Grid.Column="0" Grid.Row="0"` ad meg. Mivel a mi rácsunk összesen négy cellát definiál, a jobb alsó cellát a `Grid.Column="1" Grid.Row="1"` azonosítja.



29.22. ábra: *A Grid panel működés közben*

Forráskód A SimpleGrid.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

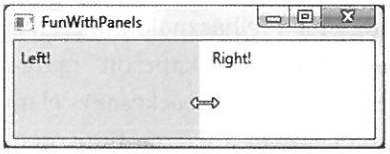
Rácsok GridSplitter típusokkal

A `Grid` típusok az úgynevezett választóvonalakat támogatják. Ahogy bizonyára tudjuk, a választóvonalak lehetővé teszik, hogy a végfelhasználó átméretezze egy rácstípus sorait vagy oszlopait. Ha ez készen van, az egyes átméretezhető cellákban lévő tartalom átalakítja magát a befoglalt elemek alapján. Nagyon könnyű választóvonalat hozzáadni egy `Grid` típushoz; egyszerűen definiáljuk a `<GridSplitter>` típust a csatolt tulajdonságszintaxis használatával, és határozzuk meg, mely sorra vagy oszlopra érvényes. Figyeljünk arra, hogy hozzá kell rendelni egy `Width` vagy egy `Height` értéket (attól függően, hogy függőleges vagy vízszintes osztásról van szó) annak érdekében, hogy látható legyen a képernyőn. Vizsgáljuk meg a következő `Grid` típust, amelyben választóvonal van az első oszlopon (`Grid.Column = "0"`):

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="FunWithPanels" Height="191" Width="436">
  <Grid Background ="AliceBlue">
    <!-- Az oszlopok meghatározása -->
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width ="Auto"/>
      <ColumnDefinition/>
    </Grid.ColumnDefinitions>
```

```
    <!-- Címke hozzáadása a 0 cellához -->
    <Label Name="lblLeft" Background ="GreenYellow"
           Grid.Column="0" Content ="Left!"/>

    <!-- A választóvonal meghatározása -->
    <GridSplitter Grid.Column ="0" Width ="5"/>

    <!-- Címke hozzáadása az 1 cellához -->
    <Label Name="lblRight" Grid.Column ="1" Content ="Right!"/>
  </Grid>
</Window>
```

Mindenekelőtt vegyük észre, hogy az oszlop, amely támogatja a választóvonalat, rendelkezik az `Auto.Next Width` tulajdonságával, míg a `<GridSplitter>` a csatolt tulajdonságszintaxis segítségével állapítja meg, melyik oszloppal dolgozik. Ha megnéznénk ezt a kimenetet, látnánk egy öt képpontos választóvonalat, amely lehetővé teszi, hogy átméretezzük az egyes Label típusokat (mivel nem adtunk meg `Height` vagy `Width` tulajdonságot a `Label` elemekhez, ezért azok betöltik az egész cellát). Nézzük meg a 29.23. ábrát.



29.23. ábra: *Választóvonalakat tartalmazó Grid típusok*

Forráskód A GridWithSplitter.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Tartalom elhelyezése a DockPanel paneleken belül

A `DockPanel` típust általában olyan főpanelként használjuk, amely tetszőleges számú további panelt foglalhat magában a kapcsolódó tartalom csoportosítására. Ahogyan a `Canvas` típusnál láttuk, a `DockPanel` panelek a csatolt tulajdonságszintaxis segítségével szabályozzák, hogy a bal felső sarkuk (az alapértelmezett) hova csatolja magát a panelen belül. Íme, egy nagyon egyszerű `DockPanel` definíció, amely a 29.24. ábrán látható kimenetet eredményezi:

```
<DockPanel LastChildFill ="True">
  <!-- Elemek dokkolása a panelhez -->
  <Label DockPanel.Dock ="Top" Name="lblInstruction"
         FontSize="15">Enter Car Information</Label>
```

```
    <Label DockPanel.Dock ="Left" Name="lblMake">Make</Label>
    <Label DockPanel.Dock ="Right" Name="lblColor">Color</Label>
    <Label DockPanel.Dock ="Bottom" Name="lblPetName">Pet Name</Label>
    <Button Name="btnOK">OK</Button>
</DockPanel>
```

29.24. ábra: *Egy egyszerű DockPanel*

Megjegyzés Ha több elemet adunk hozzá egy DockPanel ugyanazon oldalához, azokat a megadott oldal mellett a deklaráció sorrendjében halmozza fel a rendszer.

A `DockPanel` típusok használatának előnye, hogy ha a felhasználó átméretezi az ablakot, akkor minden elem a panel megadott oldalához „kapcsolt" marad (a `DockPanel.Dock` révén). Azt is vegyük észre, hogy a nyitó `<DockPanel>` elem `true` értékre állítja a `LastChildFill` attribútumot. Mivel a `Button` típus nem adott meg semmilyen `DockPanel.Dock` értéket, ezért kitölti megmaradt helyet.

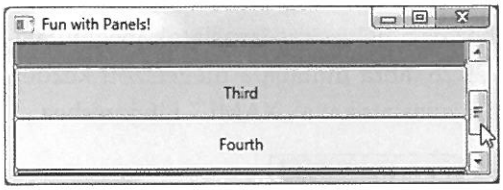
Forráskód A SimpleDockPanel.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A lapozás engedélyezése a paneltípusoknál

Érdemes rámutatni, hogy a WPF biztosítja a `<ScrollViewer>` típust, amely automatikus lapozási viselkedést kínál a beágyazott paneltípusok számára:

```
<ScrollViewer>
  <StackPanel>
    <Button Content ="First" Background = "Green" Height ="40"/>
    <Button Content ="Second" Background = "Red" Height ="40"/>
    <Button Content ="Third" Background = "Pink" Height ="40"/>
    <Button Content ="Fourth" Background = "Yellow" Height ="40"/>
    <Button Content ="Fifth" Background = "Blue" Height ="40"/>
  </StackPanel>
</ScrollViewer>
```

Az előző XAML-definíció eredménye a 29.25. ábrán látható.



29.25. ábra: *A ScrollViewer típus használata*

Forráskód A ScrollViewer.xaml fájlt a forráskódkönyvtár 29. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Ahogy várhatjuk, minden panel több tagot biztosít, amelyek lehetővé teszik a tartalom elhelyezésének finomhangolását. Ehhez kapcsolódó megjegyzés, hogy a WPF-vezérlőelemek mindegyike két érdekes tulajdonságot (`Padding` és `Margin`) támogat, amelyek segítségével maga a vezérlőelem tájékoztathatja a panelt arról, hogy milyen bánásmódot igényel. Pontosabban a `Padding` tulajdonság szabályozza, hogy mekkora plusz hely vegye körül a belső vezérlőelemet, míg a `Margin` szabályozza a vezérlőelem külső részét körülvevő extra helyet.

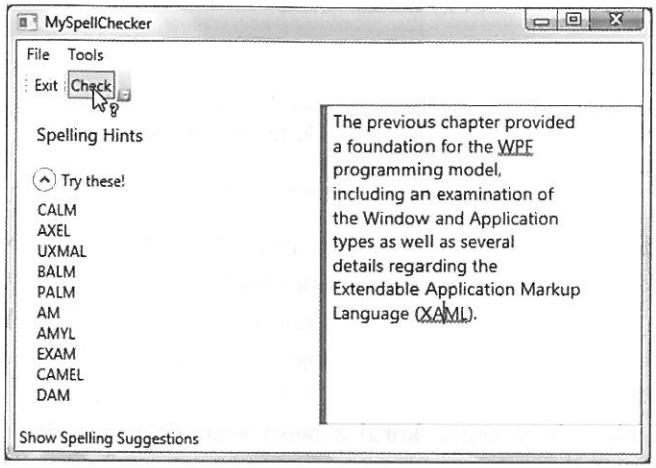
Ezzel befejeztük a WPF főbb paneltípusainak, valamint a tartalmuk pozícionálását szolgáló különböző módszerek áttekintését. Ezután megnézzük a beágyazott panelek használatának egy példáját, hogy létrehozzunk egy elrendezés-rendszert a főablak számára. Ehhez bővítjük a TextControls projekt (vagyis a helyesírásellenőrző-alkalmazás) funkcionalitását, hogy támogasson egy főmenüt, egy állapotsávot és egy eszköztárat.

Ablak kereteinek készítése beágyazott panelek használatával

Az alkalmazás frissített verzióját (amely az új, MySpellChecker nevű Visual Studio 2008 WPF alkalmazás projekt) kibővítjük és véglegesítjük a következő oldalakon, tehát pillanatnyilag az alapelrendezést és működést fogjuk összeállítani.

Célunk olyan elrendezés kialakítása, ahol a főablak rendelkezik egy felső menürendszerrel, egy eszköztárral, valamint az ablak alján elhelyezett állapotsávval. Az állapotsáv magában foglal egy panelt, amely megjelenít egy

szöveges sort akkor, ha a felhasználó kiválaszt egy menüelemet (vagy eszköztárgombot), míg a menürendszer és az eszköztár grafikusfelület-kapcsolókat kínál fel az alkalmazás bezárásához és a helyesírás-javaslatok megmutatásához egy Expander vezérlőben. A 29.26. ábra mutatja a megcélzott kezdeti elrendezést, megjelenítve a helyesírási javaslatokat a „XAML" kifejezéshez.



29.26. ábra: *Beágyazott panelek használata egy ablak felhasználói felületének létrehozásához*

Figyeljük meg, hogy a két eszköztárgombunk nem egy várt képet, hanem egy egyszerű szöveges értéket jelenít meg. Noha ez nem lenne elegendő egy termékszintű alkalmazáshoz, képek hozzárendelése az eszköztár gombjaihoz általában beágyazott erőforrások használatát igényli, amelynek témakörét a 30. fejezetben fogjuk megvizsgálni (tehát most megteszi a szöveges adat). Azt is vegyük észre, hogy amikor az egér a Check button fölé kerül, a kurzor megváltozik, az állapotsáv egyetlen panele hasznos felhasználóifelület-üzenetet jelenít meg.

A felület készítésének megkezdéséhez módosítsuk a `Window` típusunk kezdeti XAML-definícióját, hogy `<DockPanel>` gyermekelemet használjon `<Grid>` helyett:

```
<Window x:Class="MySpellChecker.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MySpellChecker" Height="331" Width="508"
  WindowStartupLocation ="CenterScreen" >

  <!-- Ez a panel alakítja ki az ablak tartalmát -->
  <DockPanel>
  </DockPanel>
</Window>
```

A menürendszer készítése

A WPF-ben a menürendszereket a `Menu` típus képviseli, amely `MenuItem` objektumok gyűjteményét tartja karban. Amikor XAML-ben készítünk menürendszert, minden `MenuItem` különböző - többek között kattintási - eseményeket kezelhet, amely akkor fordul elő, ha a végfelhasználó rákattint egy részelemre. A példánkban két felső menüelemet (File és Tools) hozunk létre, amelyek az Exit, illetve a Spelling Hints részelemeket tartalmazzák. Amellett, hogy kezeljük a kattintási eseményt az egyes részelemeknél, kezeljük a `MouseEnter` és a `MouseExit` eseményeket is, amelyeket arra használunk, hogy egy későbbi lépésben beállítsuk az állapotsáv szövegeit. Adjuk meg a következő kódot a `<DockPanel>` hatókörén belül:

```
<!-- Menürendszer elhelyezése az ablak tetején -->
<Menu DockPanel.Dock ="Top"
      HorizontalAlignment="Left" Background="White"
      BorderBrush ="Black">
  <MenuItem Header="_File" Click ="FileExit_Click" >
    <Separator/>
    <MenuItem Header ="_Exit" MouseEnter ="MouseEnterExitArea"
              MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
    </MenuItem>
    <MenuItem Header="_Tools">
      <MenuItem Header ="_Spelling Hints"
                MouseEnter ="MouseEnterToolsHintsArea"
                MouseLeave ="MouseLeaveArea"
                Click ="ToolsSpellingHints_Click"/>
  </MenuItem>
</Menu>
```

Láthatjuk, hogy a menürendszert a DockPanel tetejére helyeztük. Emellett a `<Separator>` elemet használtuk arra, hogy beszúrjunk egy vékony vízszintes vonalat a menürendszerbe, közvetlenül az Exit lehetőség elé. Azt is vegyük észre, hogy az egyes `MenuItem` elemek `Header` értékei tartalmaznak egy beágyazott aláhúzás tokent (például `_Exit`). Ezt használjuk annak meghatározására, hogy melyik betű legyen aláhúzott, ha a végfelhasználó lenyomja az Alt billentyűt (a gyorsbillentyűkhöz).

A kész menürendszer-definícióval most a különböző eseménykezelőket kell implementálnunk. Először is, adott a File > Exit kezelő, a `FileExit_Click()`, amely egyszerűen bezárja az alkalmazást az `Application.Current.Shutdown()` metódus révén. Az egyes részelemek `MouseEnter` és `MouseExit` eseménykezelői az állapotsávot fogják frissíteni; most azonban csak shelleket fogunk megadni. Végül, a Tools > Spelling Hints menüelem `ToolsSpellingHints_Click()` kezelője pillanatnyilag szintén egy shell lesz. Itt van a mögöttes kódfájl aktuális módosítása:

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

  protected void FileExit_Click(object sender, RoutedEventArgs args)
  {
    // Az alkalmazás bezárása.
    Application.Current.Shutdown();
  }

  protected void ToolsSpellingHints_Click(object sender,
                                          RoutedEventArgs args)
  {
  }
  protected void MouseEnterExitArea(object sender,
                                    RoutedEventArgs args)
  {
  }
  protected void MouseEnterToolsHintsArea(object sender,
                                          RoutedEventArgs args)
  {
  }
  protected void MouseLeaveArea(object sender, RoutedEventArgs args)
  {
  }
}
```

A ToolBar típus készítése

Az eszköztárak (amelyeket a WPF-ben a `ToolBar` típus képvisel) általában alternatív módszert biztosítanak egy menüelem aktiválására. Adjuk hozzá a következő markupot, közvetlenül a `<Menu>` definíció záró hatóköre után:

```
<!-- Az eszköztár elhelyezése a menü alá -->
<ToolBar DockPanel.Dock ="Top" >
  <Button Content ="Exit" MouseEnter ="MouseEnterExitArea"
          MouseLeave ="MouseLeaveArea" Click ="FileExit_Click"/>
  <Separator/>
  <Button Content ="Check" MouseEnter ="MouseEnterToolsHintsArea"
          MouseLeave ="MouseLeaveArea"
          Click ="ToolsSpellingHints_Click"
          Cursor="Help" />
</ToolBar>
```