A <ToolBar> típusunk két Button típust tartalmaz, amelyek történetesen ugyanazokat az eseményeket kezelik, és a kódfájlunk ugyanazon metódusai kezelik őket. A módszer segítségével összerakhatjuk az eseménykezelőinket, hogy kiszolgálják mind a menüelemeket, mind az eszköztárgombokat. Noha ez az eszköztár tipikus nyomógombokat használ, tudnunk kell, hogy a ToolBar típus „azegy" ContentControl, ezért nyugodtan beágyazhatunk bármilyen típust a felületébe (legördülő listákat, képeket, grafikákat stb.). Az egyetlen érdekes pont az, hogy a Check gomb egyedi egérkurzot támogat a Cursor tulajdonság révén.

Megjegyzés A ToolBar típus tetszés szerint becsomagolható <ToolBarTray> elembe, amely ToolBar objektumok esetében szabályozza az elrendezést, a dokkolást és a húzd-és-dobd műveleteket. További részletekért forduljunk a .NET Framework 3.5 SDK dokumentációjához.

A StatusBar típus készítése

A StatusBar típus a <DockPanel> alsó részére kerül, és egyetlen <TextBlock> típust tartalmaz, amelyet a fejezet ezen pontjáig még nem használtunk. A TextBoxhoz hasonlóan a TextBlock típus is szöveget tárol. A TextBlock típusok emellett támogatják többfajta szöveges jelölés használatát, például a félkövér szöveget, az aláhúzott szöveget, a sortöréseket stb. Míg a StatusBar típusunknak gyakorlatilag nincs szüksége ilyen támogatásra, a TextBlock típus másik előnye az, hogy olyan kis fülszövegekhez optimalizált, mint amilyenek a felhasználói felület megjegyzései az állapotsávpanelen. Adjuk a kódhoz a következő markupot, közvetlenül az előző ToolBar definíció után:

```
<!-- Helyezzünk alulra egy StatusBar típust -->
<StatusBar DockPanel.Dock ="Bottom" Background="Beige" >
  <StatusBarItem>
    <TextBlock Name="statBarText">Ready</TextBlock>
  </StatusBarItem>
</StatusBar>
```

Ezen a ponton a Visual Studio tervezőjének valahogy úgy kell kinéznie, mint a 29.27. ábrán látható képnek.

29.27. ábra: *Helyesírásellenőrző-alkalmazásunk aktuális felhasználói felülete*

A felhasználói felület véglegesítése

Felhasználó felületünk tervezetének utolsó aspektusa egy olyan osztható `Grid` típus definiálása, amely két oszlopot határoz meg. A bal oldalon lesz az Expander típus, amely `<StackPanel>` típusba csomagolva megjeleníti a helyesírási javaslatok listáját. A jobb oldalon lesz egy `TextBox` típus, amely több sort támogat, és engedélyezte a helyesírás-ellenőrzést. A teljes `<Grid>` a szülő `<DockPanel>` bal oldalához kerül. Adjuk hozzá a következő XAML-markupot, hogy befejezzük az ablak felhasználói felületének definícióját:

```
<Grid DockPanel.Dock ="Left" Background ="AliceBlue">
  <!-- A sorok és az oszlopok definiálása -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <GridSplitter Grid.Column ="0" Width ="5" Background ="Gray" />
  <StackPanel Grid.Column="0" VerticalAlignment ="Stretch" >
    <Label Name="lblSpellingInstructions"
           FontSize="14" Margin="10,10,0,0">
      Spelling Hints
    </Label>
    <Expander Name="expanderSpelling"
              Header ="Try these!" Margin="10,10,10,10">
      <!-- Ezt programozott módon töltjük fel -->
      <Label Name ="lblSpellingHints" FontSize ="12"/>
    </Expander>
  </StackPanel>

  <!-- Ez az a terület, ahova lehet majd gépelni -->
  <TextBox Grid.Column ="1"
           SpellCheck.IsEnabled ="True"
           AcceptsReturn ="True"
           Name ="txtData" FontSize ="14"
           BorderBrush ="Blue">
  </TextBox>
</Grid>
```

A megvalósítás véglegesítése

A felhasználói felület immár készen van, még implementációt kell biztosítanunk a megmaradt eseménykezelők számára. Itt van a szóban forgó releváns kód, amely a fejezet ezen pontján már nem igényel sok magyarázatot:

```
public partial class MainWindow : System.Windows.Window
{
  ...
  protected void ToolsSpellingHints_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 a Label és az Expander
          // típuson belül.
          lblSpellingHints.Content = spellingHints;

          // Bontsuk ki a bővítőt.
          expanderSpelling.IsExpanded = true;
        }
      }
      protected void MouseEnterExitArea(object sender,
                                        RoutedEventArgs args)
      {
        statBarText.Text = "Exit the Application";
      }
      protected void MouseEnterToolsHintsArea(object sender,
                                              RoutedEventArgs args)
      {
        statBarText.Text = "Show Spelling Suggestions";
      }
      protected void MouseLeaveArea(object sender, RoutedEventArgs args)
      {
        statBarText.Text = "Ready";
      }
    }
```

Készen vagyunk! Mindössze néhány sornyi imperatív kóddal (és egy egészséges adag XAML-lel) elkészültek egy működő szövegszerkesztő alapjai. Ahhoz, hogy még érdekesebbé tegyük, meg kell ismerkednünk a *vezérlő utasításokkal.*

A WPF vezérlőutasításai

A fejezet következő főbb egysége a *vezérlőutasítások* témakörének vizsgálatával foglalkozik. A Windows Presentation Foundation a vezérlőutasítások révén támogatja azt a jelenséget, amelyet „vezérlőelem-független eseményeknek" tekinthetünk. Mint tudjuk, egy tipikus .NET-eseményt adott alaposztály határoz meg, és csak az az osztály vagy annak egy leszármazottja használhatja. Ezenfelül, a normális .NET-események szorosan kötődnek ahhoz az osztályhoz, amelyben definiálták őket.

A WPF-vezérlőutasítások ezzel szemben olyan eseményszerű entitások, amelyek függetlenek egy adott vezérlőelemtől, és sok esetben sikeresen alkalmazhatók több (és látszólag nem kapcsolódó) vezérlőelem-típusra. A WPF például támogatja a másolás, a beillesztés és kivágás parancsokat, amelyeket széles körben lehet alkalmazni különféle felhasználóifelület-elemekre (menüelemek, eszköztárgombok, egyedi gombok), valamint gyorsbillentyűkre (Ctrl + C, Ctrl + V stb.).

Míg más felhasználóifelület-eszközrendszerek (mint a Windows Forms) szabványos eseményeket biztosítanak ilyen célokra, a végeredmény általában redundáns, és a kódot nehéz karbantartani. A WPF-modell alatt a parancsok alternatívaként használhatók. A végeredmény általában kisebb és rugalmasabb kód.

A belső vezérlőelem parancsobjektumok

A WPF számos beépített vezérlő utasítással érkezik, amelyek mindegyikéhez beállíthatunk gyorsbillentyűt (vagy más beviteli lehetőséget). Programozási szempontból egy WPF-vezérlőutasítás egy objektum, amely támogat egy olyan tulajdonságot (gyakran `Command` néven), amely visszaad egy itt látható, `ICommand` interfészt megvalósító objektumot:

```
public interface ICommand
{
  // Akkor fordul elő, amikor a módosítások befolyásolják, hogy
  // a parancs végrehajtásra kerüljön-e vagy sem.
  event EventHandler CanExecuteChanged;

  // Definiálja azt a metódust, amely meghatározza, hogy a parancs
  // végrehajtható-e aktuális állapotában.
  bool CanExecute(object parameter);

  // Definiálja a parancs indításakor meghívandó metódust.
  void Execute(object parameter);
}
```

Bár a saját interfészimplementációnkat is megalkothatnánk egy vezérlőutasításhoz, kicsi annak az esélye, hogy erre szükség van, köszönhetően az azonnal működő öt WPF-parancsobjektumnak, amelyek biztosítják ezt a működést. Ezek a statikus osztályok több olyan tulajdonságot definiálnak, amelyek feltárják az `ICommand` parancsot megvalósító objektumokat - általában a `RoutedUICommand` típust, amely támogatja a WPF továbbított esemény modellt.

A 29.4. táblázat az egyes belső parancsobjektumok által kiajánlott alaptulajdonságok közül ismertet néhányat (további részletekért tekintsük át a .NET Framework 3.5 SDK dokumentációját).

A WPF-vezérlőelem parancsobjektuma	Példa a vezérlőelem parancstulajdonságaira	Jelentés
`ApplicationCommands`	`Close`, `Copy`, `Cut`, `Delete`, `Find`, `Open`, `Paste`, `Save`, `SaveAll`, `Redo`, `Undo`	Olyan tulajdonságokat definiál, amelyek az alkalmazásszintű parancsokat képviselik.
`ComponentsCommands`	`MoveDown`, `MoveFocusBack`, `MoveLeft`, `MoveRight`, `ScrollToEnd`, `ScrollToHome`	Azokat a tulajdonságokat határozza meg, amelyek a felhasználói felület elemei által végrehajtott közös utasításokra képezhetők le.
`MediaCommands`	`BoostBase`, `ChannelUp`, `ChannelDown`, `FastForward`, `NextTrack`, `Play`, `Rewind`, `Select`, `Stop`	Olyan tulajdonságokat határoz meg, amelyek lehetővé teszik, hogy különböző médiaközpontú vezérlőelemek közös parancsokat adjanak ki.
`NavigationCommands`	`BrowseBack`, `BrowseForward`, `Favorites`, `LastPage`, `NextPage`, `Zoom`	Több olyan tulajdonságot definiál, amelyek a WPF navigációs modelljét kihasználó alkalmazások esetében használatosak.
`EditingCommands`	`AlignCenter`, `CorrectSpellingError`, `DecreaseFontSize`, `EnterLineBreak`, `EnterParagraphBreak`, `MoveDownByLine`, `MoveRightByWord`	Olyan tulajdonságokat definiál, amelyeket általában a WPF-dokumentum API által feltárt objektumokkal történő programozása esetén használunk.

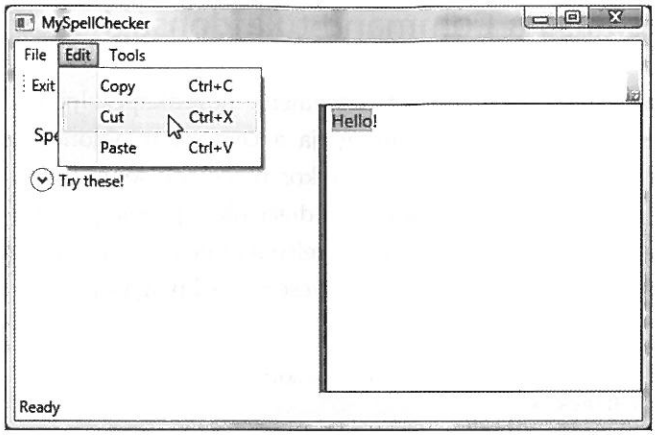
29.4. táblázat: *A belső WPF-vezérlőelem parancsobjektumok*

Utasítások kapcsolása a Command tulajdonsághoz

Ha ezen parancstulajdonságok bármelyikét szeretnénk hozzákapcsolni egy olyan felhasználóifelület-elemhez, amely támogatja a `Command` tulajdonságot (ilyen például egy `Button` vagy egy `MenuItem`), akkor nincs sok tennivalónk. Ahhoz, hogy lássuk, hogyan történik mindez, módosítsuk a jelenlegi menürendszert úgy, hogy támogasson egy Edit nevű új felső menüelemet, és három részelemet a szöveges adatok másolásához, beillesztéséhez és kivágásához:

```
<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>

  <!-- Új menüelem parancsokkal! -->
  <MenuItem Header="_Edit">
    <MenuItem Command ="ApplicationCommands.Copy"/>
    <MenuItem Command ="ApplicationCommands.Cut"/>
    <MenuItem Command ="ApplicationCommands.Paste"/>
  </MenuItem>

  <MenuItem Header="_Tools">
    <MenuItem Header ="_Spelling Hints"
              MouseEnter ="MouseEnterToolsHintsArea"
              MouseLeave ="MouseLeaveArea"
              Click ="ToolsSpellingHints_Click"/>
  </MenuItem>
</Menu>
```

Figyeljük meg, hogy minden részelem rendelkezik egy, a `Command` tulajdonsághoz rendelt értékkel. Ezáltal a menüelemek automatikusan megkapják a megfelelő nevet és gyorsbillentyűt (pl. Ctrl + C a másolás művelethez) a menüelem grafikus felületében, így az alkalmazás már imperatív kód nélkül is képes „másolás, kivágás és beillesztés" műveletekre. Ezért, ha futtatnánk az alkalmazást, és kijelölnénk valamennyi szöveget, akkor a 29.28. ábrán látható módon azonnal használhatnánk az új menüelemeket.



29.28. ábra: *A parancsobjektumok gazdag funkcionalitást biztosítanak ingyen*

Utasítások kapcsolása a felhasználói felület tetszőleges elemeihez

Ha a felhasználói felület olyan eleméhez szeretnénk utasítást kapcsolni, amely nem támogatja a `Command` tulajdonságot, akkor szükségünk lesz imperatív kódra. Nem túlságosan bonyolult dolog, de egy kicsit több logikát igényel, mint amennyit a XMAL-ben láthatunk. Például mi történik, ha azt szeretnénk, hogy az egész ablak reagáljon az F1 billentyűre, így amikor a végfelhasználó megnyomja ezt a gombot, akkor aktiválhatja a társított menürendszert?

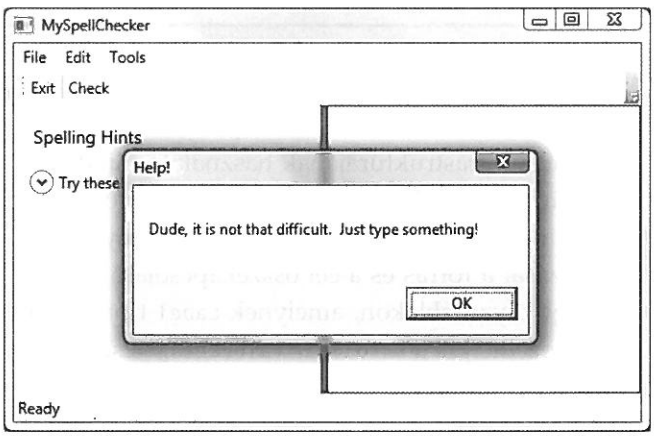
Tételezzük fel, hogy a főablak kódfájlja definiál egy új, `SetF1CommandBinding()` nevű metódust, amelyet a konstruktorban hívunk meg az `InitializeComponent()` meghívása után. Ez az új metódus programozott módon létrehoz egy új `CommandBinding` objektumot, amelyet úgy konfigurálunk, hogy az `ApplicationCommands.Help` lehetőséggel működjön, amely automatikusan reagál az F1-re:

```
private void SetF1CommandBinding()
{
  CommandBinding helpBinding =
                 new CommandBinding(ApplicationCommands.Help);
  helpBinding.CanExecute += CanHelpExecute;
  helpBinding.Executed += HelpExecuted;
  CommandBindings.Add(helpBinding);
}
```

A legtöbb `CommandBinding` objektum kezelni akarja majd a `CanExecute` eseményt (amely lehetővé teszi, hogy megadjuk, hogy az utasítást a rendszer a program működése alapján végrehajtsa-e vagy sem) és az `Executed` eseményt (amelyben meghatározhatjuk az utasítás végrehajtásakor megjelenített tartalmat). Adjuk hozzá a következő eseménykezelőket a `Window`-leszármazott típusunkhoz (figyeljük meg, hogy a társított metódusreferenciák az egyes metódusok milyen formátumát követelik meg):

```
private void CanHelpExecute(object sender,
                            CanExecuteRoutedEventArgs e)
{
  // Itt beállíthatjuk a CanExecute false értékét, ha szükség esetén
  // megakadályoznánk, hogy az utasítás végrehajtása megtörténjen.
  e.CanExecute = true;
}

private void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
{
  MessageBox.Show("Dude, it is not that difficult.
                   Just type something!", "Help!");
}
```

Itt implementáltuk a `CanHelpExecute()` metódust, és mindig lehetővé tesszük az F1 súgó megjelenítését a `true` érték visszaadásával. Azonban, ha bizonyos helyzetekben a súgórendszernek nem kellene megjelennie, erről gondoskodhatunk, és szükség esetén `false` értéket adhatunk vissza. A `HelpExecute()` metódusban megjelenő „súgórendszerünk" alig több, mint egy üzenetdoboz. Most futtathatjuk az alkalmazásunkat. Ha megnyomjuk az F1 billentyűt, megtekinthetjük a 29.29. ábrán látható, felhasználóknak készült útmutatórendszerünket (amely nem túl hasznos, és kicsit talán sértő is).



29.29. ábra: *Az egyedi súgórendszerünk*

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

A WPF adatkötési modell

A vezérlőelemek gyakran célpontjai különböző adatkötési műveleteknek. Az *adatkötés* tulajdonképpen vezérlőelem-tulajdonságok kapcsolásának művelete olyan adatértékekhez, amelyek változhatnak az alkalmazás élettartama során. Ezáltal a felhasználói interfész eleme megjelenítheti a kódunkban egy változó állapotát; például:

- `CheckBox` vezérlőelem bejelölését az adott objektum `Boolean` tulajdonságának alapján.
- Adatok megjelenítését `TextBox` típusokban egy relációs adatbázis-táblázatból.
- Egész számhoz kapcsolt `Label`t, amely a fájlok számát mutatja egy mappában.

A belső WPF adatkötési motor használata során tisztában kell lennünk a kötési művelet *forrása* és *célja* közötti különbséggel. Ahogyan várhatjuk, egy adatkötési művelet forrása maga az adat (egy `Boolean` tulajdonság, relációs adat stb.), míg a cél (vagy célpont) az a felhasználói felületi vezérlőelem-tulajdonság, amely az adattartalmat használja majd (egy `CheckBox`, egy `TextBox` stb.).

Megjegyzés Egy adatkötési művelet cél tulajdonságának egy felhasználóifelület-elem függőségi tulajdonságának kell lennie.

Igazság szerint a WPF adatkötési infrastruktúrájának használata mindig opcionális. Ha egy fejlesztő a saját adatkötési logikáját alkalmazná, a forrás és a cél közötti kapcsolat általában magában foglalná különböző események kezelését és imperatív kód létrehozását a forrás és a cél összekapcsolásához. Például, ha van egy `ScrollBar` egy olyan ablakon, amelynek `Label` típuson kell megjelenítenie értékét, akkor kezelhetjük a `ScrollBar` `ValueChange` eseményét, és megfelelően frissíthetjük a `Label` tartalmát.

A <ToolBar> típusunk két Button típust tartalmaz, amelyek történetesen ugyanazokat az eseményeket kezelik, és a kódfájlunk ugyanazon metódusai kezelik őket. A módszer segítségével összerakhatjuk az eseménykezelőinket, hogy kiszolgálják mind a menüelemeket, mind az eszköztárgombokat. Noha ez az eszköztár tipikus nyomógombokat használ, tudnunk kell, hogy a ToolBar típus „azegy" ContentControl, ezért nyugodtan beágyazhatunk bármilyen típust a felületébe (legördülő listákat, képeket, grafikákat stb.). Az egyetlen érdekes pont az, hogy a Check gomb egyedi egérkurzot támogat a Cursor tulajdonság révén.

Megjegyzés A ToolBar típus tetszés szerint becsomagolható <ToolBarTray> elembe, amely ToolBar objektumok esetében szabályozza az elrendezést, a dokkolást és a húzd-és-dobd műveleteket. További részletekért forduljunk a .NET Framework 3.5 SDK dokumentációjához.

A StatusBar típus készítése

A StatusBar típus a <DockPanel> alsó részére kerül, és egyetlen <TextBlock> típust tartalmaz, amelyet a fejezet ezen pontjáig még nem használtunk. A TextBoxhoz hasonlóan a TextBlock típus is szöveget tárol. A TextBlock típusok emellett támogatják többfajta szöveges jelölés használatát, például a félkövér szöveget, az aláhúzott szöveget, a sortöréseket stb. Míg a StatusBar típusunknak gyakorlatilag nincs szüksége ilyen támogatásra, a TextBlock típus másik előnye az, hogy olyan kis fülszövegekhez optimalizált, mint amilyenek a felhasználói felület megjegyzései az állapotsávpanelen. Adjuk a kódhoz a következő markupot, közvetlenül az előző ToolBar definíció után:

```
<!-- Helyezzünk alulra egy StatusBar típust -->
<StatusBar DockPanel.Dock ="Bottom" Background="Beige" >
  <StatusBarItem>
    <TextBlock Name="statBarText">Ready</TextBlock>
  </StatusBarItem>
</StatusBar>
```

Ezen a ponton a Visual Studio tervezőjének valahogy úgy kell kinéznie, mint a 29.27. ábrán látható képnek.

29.27. ábra: *Helyesírásellenőrző-alkalmazásunk aktuális felhasználói felülete*

A felhasználói felület véglegesítése

Felhasználó felületünk tervezetének utolsó aspektusa egy olyan osztható `Grid` típus definiálása, amely két oszlopot határoz meg. A bal oldalon lesz az Expander típus, amely `<StackPanel>` típusba csomagolva megjeleníti a helyesírási javaslatok listáját. A jobb oldalon lesz egy `TextBox` típus, amely több sort támogat, és engedélyezte a helyesírás-ellenőrzést. A teljes `<Grid>` a szülő `<DockPanel>` bal oldalához kerül. Adjuk hozzá a következő XAML-markupot, hogy befejezzük az ablak felhasználói felületének definícióját:

```
<Grid DockPanel.Dock ="Left" Background ="AliceBlue">
  <!-- A sorok és az oszlopok definiálása -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <GridSplitter Grid.Column ="0" Width ="5" Background ="Gray" />
  <StackPanel Grid.Column="0" VerticalAlignment ="Stretch" >
    <Label Name="lblSpellingInstructions"
           FontSize="14" Margin="10,10,0,0">
      Spelling Hints
    </Label>
    <Expander Name="expanderSpelling"
              Header ="Try these!" Margin="10,10,10,10">
      <!-- Ezt programozott módon töltjük fel -->
      <Label Name ="lblSpellingHints" FontSize ="12"/>
    </Expander>
  </StackPanel>

  <!-- Ez az a terület, ahova lehet majd gépelni -->
  <TextBox Grid.Column ="1"
           SpellCheck.IsEnabled ="True"
           AcceptsReturn ="True"
           Name ="txtData" FontSize ="14"
           BorderBrush ="Blue">
  </TextBox>
</Grid>
```

A megvalósítás véglegesítése

A felhasználói felület immár készen van, még implementációt kell biztosítanunk a megmaradt eseménykezelők számára. Itt van a szóban forgó releváns kód, amely a fejezet ezen pontján már nem igényel sok magyarázatot:

```
public partial class MainWindow : System.Windows.Window
{
  ...
  protected void ToolsSpellingHints_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 a Label és az Expander
          // típuson belül.
          lblSpellingHints.Content = spellingHints;

          // Bontsuk ki a bővítőt.
          expanderSpelling.IsExpanded = true;
        }
      }
      protected void MouseEnterExitArea(object sender,
                                        RoutedEventArgs args)
      {
        statBarText.Text = "Exit the Application";
      }
      protected void MouseEnterToolsHintsArea(object sender,
                                              RoutedEventArgs args)
      {
        statBarText.Text = "Show Spelling Suggestions";
      }
      protected void MouseLeaveArea(object sender, RoutedEventArgs args)
      {
        statBarText.Text = "Ready";
      }
    }
```

Készen vagyunk! Mindössze néhány sornyi imperatív kóddal (és egy egészséges adag XAML-lel) elkészültek egy működő szövegszerkesztő alapjai. Ahhoz, hogy még érdekesebbé tegyük, meg kell ismerkednünk a *vezérlő utasításokkal.*

A WPF vezérlőutasításai

A fejezet következő főbb egysége a *vezérlőutasítások* témakörének vizsgálatával foglalkozik. A Windows Presentation Foundation a vezérlőutasítások révén támogatja azt a jelenséget, amelyet „vezérlőelem-független eseményeknek" tekinthetünk. Mint tudjuk, egy tipikus .NET-eseményt adott alaposztály határoz meg, és csak az az osztály vagy annak egy leszármazottja használhatja. Ezenfelül, a normális .NET-események szorosan kötődnek ahhoz az osztályhoz, amelyben definiálták őket.

A WPF-vezérlőutasítások ezzel szemben olyan eseményszerű entitások, amelyek függetlenek egy adott vezérlőelemtől, és sok esetben sikeresen alkalmazhatók több (és látszólag nem kapcsolódó) vezérlőelem-típusra. A WPF például támogatja a másolás, a beillesztés és kivágás parancsokat, amelyeket széles körben lehet alkalmazni különféle felhasználóifelület-elemekre (menüelemek, eszköztárgombok, egyedi gombok), valamint gyorsbillentyűkre (Ctrl + C, Ctrl + V stb.).

Míg más felhasználóifelület-eszközrendszerek (mint a Windows Forms) szabványos eseményeket biztosítanak ilyen célokra, a végeredmény általában redundáns, és a kódot nehéz karbantartani. A WPF-modell alatt a parancsok alternatívaként használhatók. A végeredmény általában kisebb és rugalmasabb kód.

A belső vezérlőelem parancsobjektumok

A WPF számos beépített vezérlő utasítással érkezik, amelyek mindegyikéhez beállíthatunk gyorsbillentyűt (vagy más beviteli lehetőséget). Programozási szempontból egy WPF-vezérlőutasítás egy objektum, amely támogat egy olyan tulajdonságot (gyakran `Command` néven), amely visszaad egy itt látható, `ICommand` interfészt megvalósító objektumot:

```
public interface ICommand
{
  // Akkor fordul elő, amikor a módosítások befolyásolják, hogy
  // a parancs végrehajtásra kerüljön-e vagy sem.
  event EventHandler CanExecuteChanged;

  // Definiálja azt a metódust, amely meghatározza, hogy a parancs
  // végrehajtható-e aktuális állapotában.
  bool CanExecute(object parameter);

  // Definiálja a parancs indításakor meghívandó metódust.
  void Execute(object parameter);
}
```

Bár a saját interfészimplementációnkat is megalkothatnánk egy vezérlőutasításhoz, kicsi annak az esélye, hogy erre szükség van, köszönhetően az azonnal működő öt WPF-parancsobjektumnak, amelyek biztosítják ezt a működést. Ezek a statikus osztályok több olyan tulajdonságot definiálnak, amelyek feltárják az `ICommand` parancsot megvalósító objektumokat – általában a `RoutedUICommand` típust, amely támogatja a WPF továbbított esemény modellt.

A 29.4. táblázat az egyes belső parancsobjektumok által kiajánlott alaptulajdonságok közül ismertet néhányat (további részletekért tekintsük át a .NET Framework 3.5 SDK dokumentációját).

A WPF-vezérlőelem parancsobjektuma	Példa a vezérlőelem parancstulajdonságaira	Jelentés
`ApplicationCommands`	`Close`, `Copy`, `Cut`, `Delete`, `Find`, `Open`, `Paste`, `Save`, `SaveAll`, `Redo`, `Undo`	Olyan tulajdonságokat definiál, amelyek az alkalmazásszintű parancsokat képviselik.
`ComponentsCommands`	`MoveDown`, `MoveFocusBack`, `MoveLeft`, `MoveRight`, `ScrollToEnd`, `ScrollToHome`	Azokat a tulajdonságokat határozza meg, amelyek a felhasználói felület elemei által végrehajtott közös utasításokra képezhetők le.
`MediaCommands`	`BoostBase`, `ChannelUp`, `ChannelDown`, `FastForward`, `NextTrack`, `Play`, `Rewind`, `Select`, `Stop`	Olyan tulajdonságokat határoz meg, amelyek lehetővé teszik, hogy különböző médiaközpontú vezérlőelemek közös parancsokat adjanak ki.
`NavigationCommands`	`BrowseBack`, `BrowseForward`, `Favorites`, `LastPage`, `NextPage`, `Zoom`	Több olyan tulajdonságot definiál, amelyek a WPF navigációs modelljét kihasználó alkalmazások esetében használatosak.
`EditingCommands`	`AlignCenter`, `CorrectSpellingError`, `DecreaseFontSize`, `EnterLineBreak`, `EnterParagraphBreak`, `MoveDownByLine`, `MoveRightByWord`	Olyan tulajdonságokat definiál, amelyeket általában a WPF-dokumentum API által feltárt objektumokkal történő programozása esetén használunk.

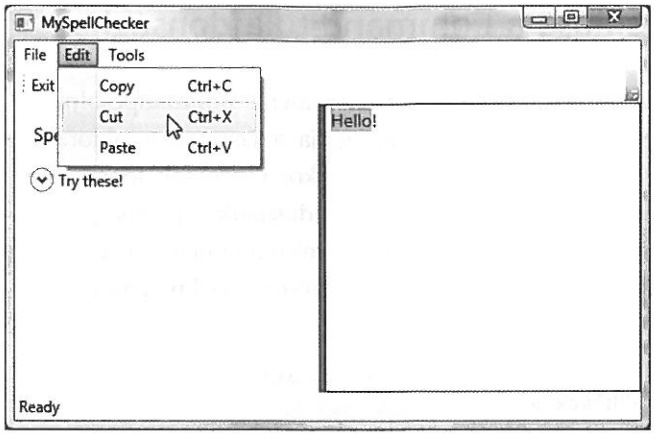
29.4. táblázat: *A belső WPF-vezérlőelem parancsobjektumok*

Utasítások kapcsolása a Command tulajdonsághoz

Ha ezen parancstulajdonságok bármelyikét szeretnénk hozzákapcsolni egy olyan felhasználóifelület-elemhez, amely támogatja a `Command` tulajdonságot (ilyen például egy `Button` vagy egy `MenuItem`), akkor nincs sok tennivalónk. Ahhoz, hogy lássuk, hogyan történik mindez, módosítsuk a jelenlegi menürendszert úgy, hogy támogasson egy Edit nevű új felső menüelemet, és három részelemet a szöveges adatok másolásához, beillesztéséhez és kivágásához:

```
<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>

  <!-- Új menüelem parancsokkal! -->
  <MenuItem Header="_Edit">
    <MenuItem Command ="ApplicationCommands.Copy"/>
    <MenuItem Command ="ApplicationCommands.Cut"/>
    <MenuItem Command ="ApplicationCommands.Paste"/>
  </MenuItem>

  <MenuItem Header="_Tools">
    <MenuItem Header ="_Spelling Hints"
              MouseEnter ="MouseEnterToolsHintsArea"
              MouseLeave ="MouseLeaveArea"
              Click ="ToolsSpellingHints_Click"/>
  </MenuItem>
</Menu>
```

Figyeljük meg, hogy minden részelem rendelkezik egy, a `Command` tulajdonsághoz rendelt értékkel. Ezáltal a menüelemek automatikusan megkapják a megfelelő nevet és gyorsbillentyűt (pl. Ctrl + C a másolás művelethez) a menüelem grafikus felületében, így az alkalmazás már imperatív kód nélkül is képes „másolás, kivágás és beillesztés" műveletekre. Ezért, ha futtatnánk az alkalmazást, és kijelölnénk valamennyi szöveget, akkor a 29.28. ábrán látható módon azonnal használhatnánk az új menüelemeket.



29.28. ábra: *A parancsobjektumok gazdag funkcionalitást biztosítanak ingyen*

Utasítások kapcsolása a felhasználói felület tetszőleges elemeihez

Ha a felhasználói felület olyan eleméhez szeretnénk utasítást kapcsolni, amely nem támogatja a `Command` tulajdonságot, akkor szükségünk lesz imperatív kódra. Nem túlságosan bonyolult dolog, de egy kicsit több logikát igényel, mint amennyit a XMAL-ben láthatunk. Például mi történik, ha azt szeretnénk, hogy az egész ablak reagáljon az F1 billentyűre, így amikor a végfelhasználó megnyomja ezt a gombot, akkor aktiválhatja a társított menürendszert?

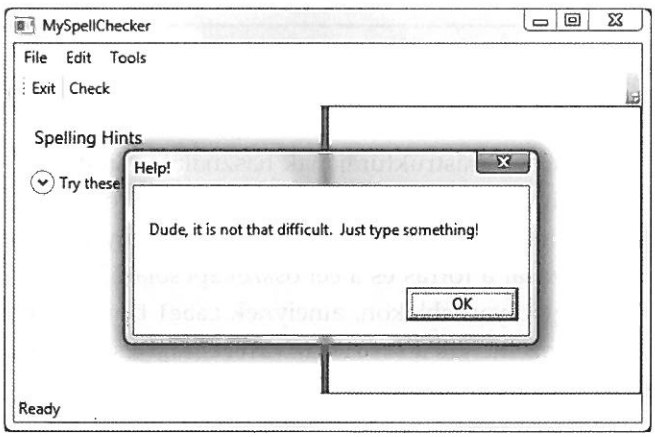
Tételezzük fel, hogy a főablak kódfájlja definiál egy új, `SetF1CommandBinding()` nevű metódust, amelyet a konstruktorban hívunk meg az `InitializeComponent()` meghívása után. Ez az új metódus programozott módon létrehoz egy új `CommandBinding` objektumot, amelyet úgy konfigurálunk, hogy az `ApplicationCommands.Help` lehetőséggel működjön, amely automatikusan reagál az F1-re:

```
private void SetF1CommandBinding()
{
  CommandBinding helpBinding =
                 new CommandBinding(ApplicationCommands.Help);
  helpBinding.CanExecute += CanHelpExecute;
  helpBinding.Executed += HelpExecuted;
  CommandBindings.Add(helpBinding);
}
```

A legtöbb `CommandBinding` objektum kezelni akarja majd a `CanExecute` eseményt (amely lehetővé teszi, hogy megadjuk, hogy az utasítást a rendszer a program működése alapján végrehajtsa-e vagy sem) és az `Executed` eseményt (amelyben meghatározhatjuk az utasítás végrehajtásakor megjelenített tartalmat). Adjuk hozzá a következő eseménykezelőket a `Window`-leszármazott típusunkhoz (figyeljük meg, hogy a társított metódusreferenciák az egyes metódusok milyen formátumát követelik meg):

```
private void CanHelpExecute(object sender,
                            CanExecuteRoutedEventArgs e)
{
  // Itt beállíthatjuk a CanExecute false értékét, ha szükség esetén
  // megakadályoznánk, hogy az utasítás végrehajtása megtörténjen.
  e.CanExecute = true;
}

private void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
{
  MessageBox.Show("Dude, it is not that difficult.
                   Just type something!", "Help!");
}
```

Itt implementáltuk a `CanHelpExecute()` metódust, és mindig lehetővé tesszük az F1 súgó megjelenítését a `true` érték visszaadásával. Azonban, ha bizonyos helyzetekben a súgórendszernek nem kellene megjelennie, erről gondoskodhatunk, és szükség esetén `false` értéket adhatunk vissza. A `HelpExecute()` metódusban megjelenő „súgórendszerünk" alig több, mint egy üzenetdoboz. Most futtathatjuk az alkalmazásunkat. Ha megnyomjuk az F1 billentyűt, megtekinthetjük a 29.29. ábrán látható, felhasználóknak készült útmutatórendszerünket (amely nem túl hasznos, és kicsit talán sértő is).



29.29. ábra: *Az egyedi súgórendszerünk*

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

A WPF adatkötési modell

A vezérlőelemek gyakran célpontjai különböző adatkötési műveleteknek. Az *adatkötés* tulajdonképpen vezérlőelem-tulajdonságok kapcsolásának művelete olyan adatértékekhez, amelyek változhatnak az alkalmazás élettartama során. Ezáltal a felhasználói interfész eleme megjelenítheti a kódunkban egy változó állapotát; például:

- `CheckBox` vezérlőelem bejelölését az adott objektum `Boolean` tulajdonságának alapján.
- Adatok megjelenítését `TextBox` típusokban egy relációs adatbázis-táblázatból.
- Egész számhoz kapcsolt `Label`t, amely a fájlok számát mutatja egy mappában.

A belső WPF adatkötési motor használata során tisztában kell lennünk a kötési művelet *forrása* és *célja* közötti különbséggel. Ahogyan várhatjuk, egy adatkötési művelet forrása maga az adat (egy `Boolean` tulajdonság, relációs adat stb.), míg a cél (vagy célpont) az a felhasználói felületi vezérlőelem-tulajdonság, amely az adattartalmat használja majd (egy `CheckBox`, egy `TextBox` stb.).

Megjegyzés Egy adatkötési művelet cél tulajdonságának egy felhasználóifelület-elem függőségi tulajdonságának kell lennie.

Igazság szerint a WPF adatkötési infrastruktúrájának használata mindig opcionális. Ha egy fejlesztő a saját adatkötési logikáját alkalmazná, a forrás és a cél közötti kapcsolat általában magában foglalná különböző események kezelését és imperatív kód létrehozását a forrás és a cél összekapcsolásához. Például, ha van egy `ScrollBar` egy olyan ablakon, amelynek `Label` típuson kell megjelenítenie értékét, akkor kezelhetjük a `ScrollBar` `ValueChange` eseményét, és megfelelően frissíthetjük a `Label` tartalmát.

A <ToolBar> típusunk két Button típust tartalmaz, amelyek történetesen ugyanazokat az eseményeket kezelik, és a kódfájlunk ugyanazon metódusai kezelik őket. A módszer segítségével összerakhatjuk az eseménykezelőinket, hogy kiszolgálják mind a menüelemeket, mind az eszköztárgombokat. Noha ez az eszköztár tipikus nyomógombokat használ, tudnunk kell, hogy a ToolBar típus „az-egy" ContentControl, ezért nyugodtan beágyazhatunk bármilyen típust a felületébe (legördülő listákat, képeket, grafikákat stb.). Az egyetlen érdekes pont az, hogy a Check gomb egyedi egérkurzot támogat a Cursor tulajdonság révén.

Megjegyzés A ToolBar típus tetszés szerint becsomagolható <ToolBarTray> elembe, amely ToolBar objektumok esetében szabályozza az elrendezést, a dokkolást és a húzd-és-dobd műveleteket. További részletekért forduljunk a .NET Framework 3.5 SDK dokumentációjához.

A StatusBar típus készítése

A StatusBar típus a <DockPanel> alsó részére kerül, és egyetlen <TextBlock> típust tartalmaz, amelyet a fejezet ezen pontjáig még nem használtunk. A TextBoxhoz hasonlóan a TextBlock típus is szöveget tárol. A TextBlock típusok emellett támogatják többfajta szöveges jelölés használatát, például a félkövér szöveget, az aláhúzott szöveget, a sortöréseket stb. Míg a StatusBar típusunknak gyakorlatilag nincs szüksége ilyen támogatásra, a TextBlock típus másik előnye az, hogy olyan kis fülszövegekhez optimalizált, mint amilyenek a felhasználói felület megjegyzései az állapotsávpanelen. Adjuk a kódhoz a következő markupot, közvetlenül az előző ToolBar definíció után:

```
<!-- Helyezzünk alulra egy StatusBar típust -->
<StatusBar DockPanel.Dock ="Bottom" Background="Beige" >
  <StatusBarItem>
    <TextBlock Name="statBarText">Ready</TextBlock>
  </StatusBarItem>
</StatusBar>
```

Ezen a ponton a Visual Studio tervezőjének valahogy úgy kell kinéznie, mint a 29.27. ábrán látható képnek.

29.27. ábra: *Helyesírásellenőrző-alkalmazásunk aktuális felhasználói felülete*

A felhasználói felület véglegesítése

Felhasználó felületünk tervezetének utolsó aspektusa egy olyan osztható `Grid` típus definiálása, amely két oszlopot határoz meg. A bal oldalon lesz az Expander típus, amely `<StackPanel>` típusba csomagolva megjeleníti a helyesírási javaslatok listáját. A jobb oldalon lesz egy `TextBox` típus, amely több sort támogat, és engedélyezte a helyesírás-ellenőrzést. A teljes `<Grid>` a szülő `<DockPanel>` bal oldalához kerül. Adjuk hozzá a következő XAML-markupot, hogy befejezzük az ablak felhasználói felületének definícióját:

```
<Grid DockPanel.Dock ="Left" Background ="AliceBlue">
  <!-- A sorok és az oszlopok definiálása -->
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <GridSplitter Grid.Column ="0" Width ="5" Background ="Gray" />
  <StackPanel Grid.Column="0" VerticalAlignment ="Stretch" >
    <Label Name="lblSpellingInstructions"
           FontSize="14" Margin="10,10,0,0">
      Spelling Hints
    </Label>
    <Expander Name="expanderSpelling"
              Header ="Try these!" Margin="10,10,10,10">
      <!-- Ezt programozott módon töltjük fel -->
      <Label Name ="lblSpellingHints" FontSize ="12"/>
    </Expander>
  </StackPanel>

  <!-- Ez az a terület, ahova lehet majd gépelni -->
  <TextBox Grid.Column ="1"
           SpellCheck.IsEnabled ="True"
           AcceptsReturn ="True"
           Name ="txtData" FontSize ="14"
           BorderBrush ="Blue">
  </TextBox>
</Grid>
```

A megvalósítás véglegesítése

A felhasználói felület immár készen van, még implementációt kell biztosítanunk a megmaradt eseménykezelők számára. Itt van a szóban forgó releváns kód, amely a fejezet ezen pontján már nem igényel sok magyarázatot:

```
public partial class MainWindow : System.Windows.Window
{
  ...
  protected void ToolsSpellingHints_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 a Label és az Expander
          // típuson belül.
          lblSpellingHints.Content = spellingHints;

          // Bontsuk ki a bővítőt.
          expanderSpelling.IsExpanded = true;
        }
      }
      protected void MouseEnterExitArea(object sender,
                                        RoutedEventArgs args)
      {
        statBarText.Text = "Exit the Application";
      }
      protected void MouseEnterToolsHintsArea(object sender,
                                              RoutedEventArgs args)
      {
        statBarText.Text = "Show Spelling Suggestions";
      }
      protected void MouseLeaveArea(object sender, RoutedEventArgs args)
      {
        statBarText.Text = "Ready";
      }
    }
```

Készen vagyunk! Mindössze néhány sornyi imperatív kóddal (és egy egészséges adag XAML-lel) elkészültek egy működő szövegszerkesztő alapjai. Ahhoz, hogy még érdekesebbé tegyük, meg kell ismerkednünk a *vezérlő utasításokkal.*

A WPF vezérlőutasításai

A fejezet következő főbb egysége a *vezérlőutasítások* témakörének vizsgálatával foglalkozik. A Windows Presentation Foundation a vezérlőutasítások révén támogatja azt a jelenséget, amelyet „vezérlőelem-független eseményeknek" tekinthetünk. Mint tudjuk, egy tipikus .NET-eseményt adott alaposztály határoz meg, és csak az az osztály vagy annak egy leszármazottja használhatja. Ezenfelül, a normális .NET-események szorosan kötődnek ahhoz az osztályhoz, amelyben definiálták őket.

A WPF-vezérlőutasítások ezzel szemben olyan eseményszerű entitások, amelyek függetlenek egy adott vezérlőelemtől, és sok esetben sikeresen alkalmazhatók több (és látszólag nem kapcsolódó) vezérlőelem-típusra. A WPF például támogatja a másolás, a beillesztés és kivágás parancsokat, amelyeket széles körben lehet alkalmazni különféle felhasználóifelület-elemekre (menüelemek, eszköztárgombok, egyedi gombok), valamint gyorsbillentyűkre (Ctrl + C, Ctrl + V stb.).

Míg más felhasználóifelület-eszközrendszerek (mint a Windows Forms) szabványos eseményeket biztosítanak ilyen célokra, a végeredmény általában redundáns, és a kódot nehéz karbantartani. A WPF-modell alatt a parancsok alternatívaként használhatók. A végeredmény általában kisebb és rugalmasabb kód.

A belső vezérlőelem parancsobjektumok

A WPF számos beépített vezérlő utasítással érkezik, amelyek mindegyikéhez beállíthatunk gyorsbillentyűt (vagy más beviteli lehetőséget). Programozási szempontból egy WPF-vezérlőutasítás egy objektum, amely támogat egy olyan tulajdonságot (gyakran `Command` néven), amely visszaad egy itt látható, `ICommand` interfészt megvalósító objektumot:

```
public interface ICommand
{
  // Akkor fordul elő, amikor a módosítások befolyásolják, hogy
  // a parancs végrehajtásra kerüljön-e vagy sem.
  event EventHandler CanExecuteChanged;

  // Definiálja azt a metódust, amely meghatározza, hogy a parancs
  // végrehajtható-e aktuális állapotában.
  bool CanExecute(object parameter);

  // Definiálja a parancs indításakor meghívandó metódust.
  void Execute(object parameter);
}
```

Bár a saját interfészimplementációnkat is megalkothatnánk egy vezérlőutasításhoz, kicsi annak az esélye, hogy erre szükség van, köszönhetően az azonnal működő öt WPF-parancsobjektumnak, amelyek biztosítják ezt a működést. Ezek a statikus osztályok több olyan tulajdonságot definiálnak, amelyek feltárják az `ICommand` parancsot megvalósító objektumokat – általában a `RoutedUICommand` típust, amely támogatja a WPF továbbított esemény modellt.

A 29.4. táblázat az egyes belső parancsobjektumok által kiajánlott alaptulajdonságok közül ismertet néhányat (további részletekért tekintsük át a .NET Framework 3.5 SDK dokumentációját).

A WPF-vezérlőelem parancsobjektuma	Példa a vezérlőelem parancstulajdonságaira	Jelentés
`ApplicationCommands`	`Close`, `Copy`, `Cut`, `Delete`, `Find`, `Open`, `Paste`, `Save`, `SaveAll`, `Redo`, `Undo`	Olyan tulajdonságokat definiál, amelyek az alkalmazásszintű parancsokat képviselik.
`ComponentsCommands`	`MoveDown`, `MoveFocusBack`, `MoveLeft`, `MoveRight`, `ScrollToEnd`, `ScrollToHome`	Azokat a tulajdonságokat határozza meg, amelyek a felhasználói felület elemei által végrehajtott közös utasításokra képezhetők le.
`MediaCommands`	`BoostBase`, `ChannelUp`, `ChannelDown`, `FastForward`, `NextTrack`, `Play`, `Rewind`, `Select`, `Stop`	Olyan tulajdonságokat határoz meg, amelyek lehetővé teszik, hogy különböző médiaközpontú vezérlőelemek közös parancsokat adjanak ki.
`NavigationCommands`	`BrowseBack`, `BrowseForward`, `Favorites`, `LastPage`, `NextPage`, `Zoom`	Több olyan tulajdonságot definiál, amelyek a WPF navigációs modelljét kihasználó alkalmazások esetében használatosak.
`EditingCommands`	`AlignCenter`, `CorrectSpellingError`, `DecreaseFontSize`, `EnterLineBreak`, `EnterParagraphBreak`, `MoveDownByLine`, `MoveRightByWord`	Olyan tulajdonságokat definiál, amelyeket általában a WPF-dokumentum API által feltárt objektumokkal történő programozása esetén használunk.

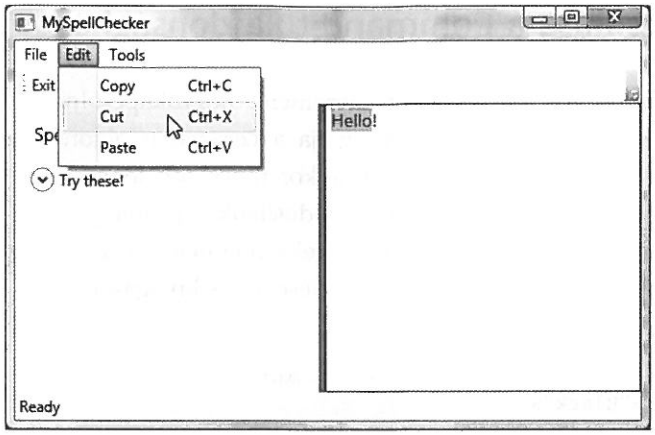
29.4. táblázat: *A belső WPF-vezérlőelem parancsobjektumok*

Utasítások kapcsolása a Command tulajdonsághoz

Ha ezen parancstulajdonságok bármelyikét szeretnénk hozzákapcsolni egy olyan felhasználóifelület-elemhez, amely támogatja a `Command` tulajdonságot (ilyen például egy `Button` vagy egy `MenuItem`), akkor nincs sok tennivalónk. Ahhoz, hogy lássuk, hogyan történik mindez, módosítsuk a jelenlegi menürendszert úgy, hogy támogasson egy Edit nevű új felső menüelemet, és három részelemet a szöveges adatok másolásához, beillesztéséhez és kivágásához:

```
<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>

  <!-- Új menüelem parancsokkal! -->
  <MenuItem Header="_Edit">
    <MenuItem Command ="ApplicationCommands.Copy"/>
    <MenuItem Command ="ApplicationCommands.Cut"/>
    <MenuItem Command ="ApplicationCommands.Paste"/>
  </MenuItem>

  <MenuItem Header="_Tools">
    <MenuItem Header ="_Spelling Hints"
              MouseEnter ="MouseEnterToolsHintsArea"
              MouseLeave ="MouseLeaveArea"
              Click ="ToolsSpellingHints_Click"/>
  </MenuItem>
</Menu>
```

Figyeljük meg, hogy minden részelem rendelkezik egy, a `Command` tulajdonsághoz rendelt értékkel. Ezáltal a menüelemek automatikusan megkapják a megfelelő nevet és gyorsbillentyűt (pl. Ctrl + C a másolás művelethez) a menüelem grafikus felületében, így az alkalmazás már imperatív kód nélkül is képes „másolás, kivágás és beillesztés" műveletekre. Ezért, ha futtatnánk az alkalmazást, és kijelölnénk valamennyi szöveget, akkor a 29.28. ábrán látható módon azonnal használhatnánk az új menüelemeket.



29.28. ábra: *A parancsobjektumok gazdag funkcionalitást biztosítanak ingyen*

Utasítások kapcsolása a felhasználói felület tetszőleges elemeihez

Ha a felhasználói felület olyan eleméhez szeretnénk utasítást kapcsolni, amely nem támogatja a `Command` tulajdonságot, akkor szükségünk lesz imperatív kódra. Nem túlságosan bonyolult dolog, de egy kicsit több logikát igényel, mint amennyit a XMAL-ben láthatunk. Például mi történik, ha azt szeretnénk, hogy az egész ablak reagáljon az F1 billentyűre, így amikor a végfelhasználó megnyomja ezt a gombot, akkor aktiválhatja a társított menürendszert?

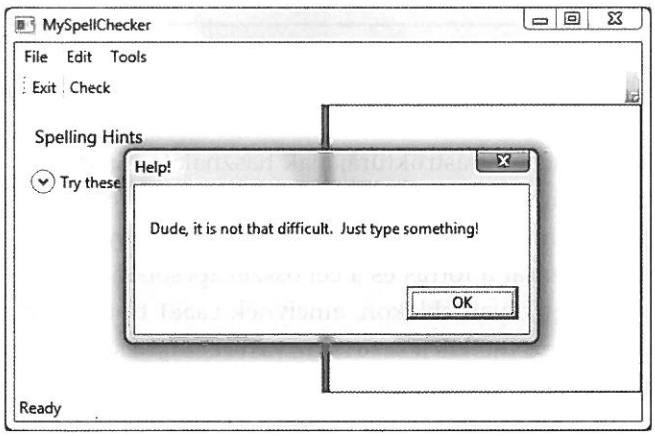
Tételezzük fel, hogy a főablak kódfájlja definiál egy új, `SetF1CommandBinding()` nevű metódust, amelyet a konstruktorban hívunk meg az `InitializeComponent()` meghívása után. Ez az új metódus programozott módon létrehoz egy új `CommandBinding` objektumot, amelyet úgy konfigurálunk, hogy az `ApplicationCommands.Help` lehetőséggel működjön, amely automatikusan reagál az F1-re:

```
private void SetF1CommandBinding()
{
  CommandBinding helpBinding =
                 new CommandBinding(ApplicationCommands.Help);
  helpBinding.CanExecute += CanHelpExecute;
  helpBinding.Executed += HelpExecuted;
  CommandBindings.Add(helpBinding);
}
```

A legtöbb `CommandBinding` objektum kezelni akarja majd a `CanExecute` eseményt (amely lehetővé teszi, hogy megadjuk, hogy az utasítást a rendszer a program működése alapján végrehajtsa-e vagy sem) és az `Executed` eseményt (amelyben meghatározhatjuk az utasítás végrehajtásakor megjelenített tartalmat). Adjuk hozzá a következő eseménykezelőket a `Window`-leszármazott típusunkhoz (figyeljük meg, hogy a társított metódusreferenciák az egyes metódusok milyen formátumát követelik meg):

```
private void CanHelpExecute(object sender,
                            CanExecuteRoutedEventArgs e)
{
  // Itt beállíthatjuk a CanExecute false értékét, ha szükség esetén
  // megakadályoznánk, hogy az utasítás végrehajtása megtörténjen.
  e.CanExecute = true;
}

private void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
{
  MessageBox.Show("Dude, it is not that difficult.
                   Just type something!", "Help!");
}
```

Itt implementáltuk a `CanHelpExecute()` metódust, és mindig lehetővé tesszük az F1 súgó megjelenítését a `true` érték visszaadásával. Azonban, ha bizonyos helyzetekben a súgórendszernek nem kellene megjelennie, erről gondoskodhatunk, és szükség esetén `false` értéket adhatunk vissza. A `HelpExecute()` metódusban megjelenő „súgórendszerünk" alig több, mint egy üzenetdoboz. Most futtathatjuk az alkalmazásunkat. Ha megnyomjuk az F1 billentyűt, megtekinthetjük a 29.29. ábrán látható, felhasználóknak készült útmutatórendszerünket (amely nem túl hasznos, és kicsit talán sértő is).



29.29. ábra: *Az egyedi súgórendszerünk*

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

A WPF adatkötési modell

A vezérlőelemek gyakran célpontjai különböző adatkötési műveleteknek. Az *adatkötés* tulajdonképpen vezérlőelem-tulajdonságok kapcsolásának művelete olyan adatértékekhez, amelyek változhatnak az alkalmazás élettartama során. Ezáltal a felhasználói interfész eleme megjelenítheti a kódunkban egy változó állapotát; például:

- `CheckBox` vezérlőelem bejelölését az adott objektum `Boolean` tulajdonságának alapján.
- Adatok megjelenítését `TextBox` típusokban egy relációs adatbázistáblázatból.
- Egész számhoz kapcsolt `Label`t, amely a fájlok számát mutatja egy mappában.

A belső WPF adatkötési motor használata során tisztában kell lennünk a kötési művelet *forrása* és *célja* közötti különbséggel. Ahogyan várhatjuk, egy adatkötési művelet forrása maga az adat (egy `Boolean` tulajdonság, relációs adat stb.), míg a cél (vagy célpont) az a felhasználói felületi vezérlőelem-tulajdonság, amely az adattartalmat használja majd (egy `CheckBox`, egy `TextBox` stb.).

Megjegyzés Egy adatkötési művelet cél tulajdonságának egy felhasználóifelület-elem függőségi tulajdonságának kell lennie.

Igazság szerint a WPF adatkötési infrastruktúrájának használata mindig opcionális. Ha egy fejlesztő a saját adatkötési logikáját alkalmazná, a forrás és a cél közötti kapcsolat általában magában foglalná különböző események kezelését és imperatív kód létrehozását a forrás és a cél összekapcsolásához. Például, ha van egy `ScrollBar` egy olyan ablakon, amelynek `Label` típuson kell megjelenítenie értékét, akkor kezelhetjük a `ScrollBar` `ValueChange` eseményét, és megfelelően frissíthetjük a `Label` tartalmát.

Azonban a WPF-adatkötés használatakor közvetlenül XAML-ben (vagy C#-kód használatával a kódfájlban) kapcsolhatjuk össze a forrást és a célt anélkül, hogy kezelnünk kellene különböző eseményeket, vagy kódolnunk kellene a kapcsolatokat a forrás és a cél között. Attól függően, hogyan állítottuk fel az adatkötési logikát, biztosíthatjuk, hogy a forrás és a cél szinkronban marad, ha valamelyikük értékei megváltoznak.

Ismerkedés az adatkötéssel

Kezdjük el a WPF adatkötési lehetőségeinek vizsgálatát, és tételezzük fel, hogy van egy új WPF-alkalmazás projektünk (SimpleDataBinding névvel), amely a következő markupot határozza meg egy `Window` típus számára:

```
<Window x:Class="SimpleDataBinding.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Simple Data Binding" Height="152" Width="300"
  WindowStartupLocation="CenterScreen">

  <StackPanel Width="250">
    <Label Content="Move the scroll bar to see the current value"/>

    <!-- A görgetősáv értéke az adatkötés forrása -->
    <ScrollBar Orientation="Horizontal" Height="30" Name="mySB"
               Maximum = "100" LargeChange="1" SmallChange="1"/>

    <!-- A címke tartalomértéke az adatkötés célja -->
    <Label Height="30" BorderBrush="Blue" BorderThickness="2"
           Content = "{Binding ElementName=mySB, Path=Value}"
    />
  </StackPanel>
</Window>
```

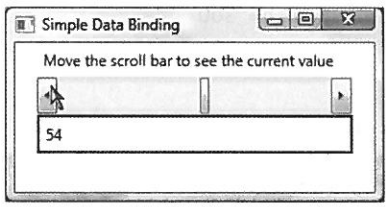
Vegyük észre, hogy a `<ScrollBar>` típust (amelynek a mySB nevet adtuk) a 0 és 100 közötti tartománnyal konfiguráltuk. Ahogy újrapozícionáljuk a görgetősáv csúszkáját (vagy rákattintunk a balra vagy jobbra nyílra), a `Label` automatikusan frissül az aktuális értékkel. Az a „ragasztó", amely ezt lehetővé teszi, a `{Binding}` jelölő kiterjesztés, amelyet a `Label Content` tulajdonságához rendelünk hozzá. Itt az `ElementName` érték képviseli az adatkötési művelet forrását (a `ScrollBar` objektumot), míg a `Path` érték jelképezi (jelen esetben) a forráselem megszerezni kívánt tulajdonságát.

Megjegyzés Az `ElementName` és a `Path` elnevezése furcsának tűnhet, ha olyan intuitív nevekre számítottunk, mint a „Source" és a „Destination". Azonban a fejezet későbbi részében látni fogjuk, hogy XML-dokumentumok lehetnek egy adatkötési művelet forrásai (általában az XPath használatával). Ebben az esetben az `ElementName` és a `Path` nevek a megfelelőek.

Alternatív formátumként kibővíthetjük a `{Binding}` jelölő kiterjesztés által megadott értéket úgy, hogy kizárólagosan beállítjuk a `DataContext` tulajdonságot a kötési művelet forrásaként, az alábbiak szerint:

```
<!-- Szétválasztja az objektumot/értéket a DataContexten
     keresztül -->
<Label Height="30" BorderBrush="Blue" BorderThickness="2"
       DataContext = "{Binding ElementName=mySB}"
       Content = "{Binding Path=Value}"
/>
```

Ha futtatnánk ezt az alkalmazást, akkor nagy örömünkre a `Label` típus anélkül frissülne, hogy bármilyen C#-kódot kellene írnunk (lásd a 29.30. ábrát).



29.30. ábra: *A ScrollBar érték kötése Label típushoz*

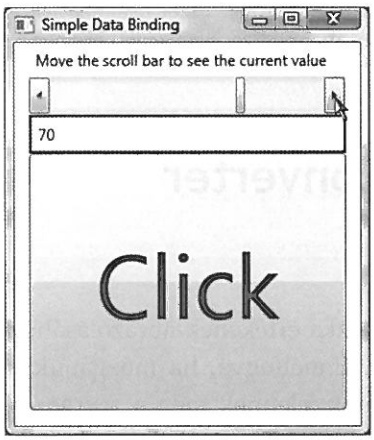
A DataContext tulajdonság

A jelenlegi példánkban láttunk két megközelítést adatkötési művelet forrásának és céljának meghatározására, amelyek ugyanazt a kimenetet eredményezték. Ezen a ponton elgondolkozhatunk, hogy mikor lenne szükségünk a `DataContext` tulajdonság kizárólagos beállítására. Ez a tulajdonság nagyon hasznos lehet, mert függőségi tulajdonság, ezért az értékét örökölheti a részelemeitől. Ily módon könnyedén beállíthatjuk ugyanazt az adatforrást vezérlőelemek családjára ahelyett, hogy rengeteg redundáns `"{Binding ElementName=X, Path=Y}"` XAML-értéket kéne ismételnünk több vezérlőelemnél. Vizsgáljuk meg az alábbi frissített XAML-definíciót a jelenlegi `<StackPanel>` tárolónkhoz:

```
<!-- Jegyezzük meg, hogy a StackPanel beállítja a DataContext
     tulajdonságot -->
<StackPanel Width="250" DataContext = "{Binding ElementName=mySB}">
  <Label Content="Move the scroll bar to see the current value"/>
```

```
    <ScrollBar Orientation="Horizontal" Height="30" Name="mySB"
               Maximum = "100" LargeChange="1" SmallChange="1"/>

    <!-- Most a felhasználói felület elemei egyedi módon használják
         a görgetősáv értékét  -->
    <Label Height="30" BorderBrush="Blue" BorderThickness="2"
           Content = "{Binding Path=Value}"/>

    <Button Content="Click" Height="200"
            FontSize = "{Binding Path=Value}"/>
</StackPanel>
```

A `DataContext` tulajdonságot itt közvetlenül a `<StackPanel>` típuson állítottuk be. Ezért, ha mozgatjuk a csúszkát, nemcsak az aktuális értéket látjuk a `Label` elemen, hanem azt is tapasztaljuk, hogy a `Button` típus mérete nő, illetve csökken ugyanezen értéknek megfelelően. A 29.31. ábrán láthatunk egy lehetséges kimenetet.



29.31. ábra: *A ScrollBar érték kötése Label és Button típushoz*

A Mode tulajdonság

Adatkötési művelet létrehozásakor választhatunk a különböző működési módok között, ha a `Path` érték meghatározása során beállítjuk a `Mode` tulajdonság értékét. Alapértelmezés szerint a `Mode` tulajdonság értéke `OneWay`, amely azt határozza meg, hogy a célon végzett módosítások nem befolyásolják a forrást. A példánkban a `Label` `Content` tulajdonságának megváltoztatása nem állítja át a `ScrollBar` csúszkájának pozícióját.

Ha szinkronban szeretnénk tartani a forrás és a cél közti módosításokat, akkor a Mode tulajdonságot átállíthatjuk TwoWay értékre. Így, ha megváltoztatjuk a Label tartalmának értékét, változik a görgetősáv csúszkájának helyzete is. Természetesen a végfelhasználó nem módosíthatja a Label tartalmát, mivel az írásvédettként állítottuk be (természetesen programozott módon megváltoztathatjuk az értéket).

A TwoWay mód használatának bemutatásához tételezzük fel, hogy a görgetősáv aktuális értékét megjelenítő Label-t az alábbi TextBox-ra cseréltük (figyeljük meg a Text tulajdonság értékét). Ebben az esetben, ha új értéket írunk a szövegmezőbe, a csúszka pozíciója (és a gomb betűmérete) automatikusan módosul, amikor a Tab billentyűvel kilépünk a TextBox objektumból:

```
<TextBox Height="30" BorderBrush="Blue"
         BorderThickness="2" Text = "{Binding Path=Value}"/>
```

Megjegyzés A Mode tulajdonságot OneTime értékre is beállíthatjuk. Ezzel beállíthatjuk a célt a kezdő érték megadásakor, de a további változtatásokat a rendszer nem követi nyomon.

Adatátalakítás az IValueConverter segítségével

A ScrollBar típus double értéket használ a csúszka értékének ábrázolásához az elvárt egész szám (vagyis integer) helyett. Ennélfogva, ha mozgatjuk a csúszkát, akkor különböző lebegőpontos számok jelennek meg a TextBoxon belül (például 61.0576923076923), ami nem túl intuitív a végfelhasználó számára, aki minden bizonnyal egész számokat vár (mint a 61, 62, 63 stb.).

Ha szeretnénk átalakítani egy adatkötési művelet értékét alternatív formátumra, annak egyik módja, hogy létrehozunk egy egyedi osztályt, amely implementálja a System.Windows.Data névtér IValueConverter interfészét. Ez az interfész két olyan tagot definiál, amelyek lehetővé teszik a kétirányú konverzió elvégzését a célpont és a cél között. Amint meghatároztuk ezt az osztályt, segítségével tovább minősíthetjük az adatkötési műveletünk feldolgozását.

Megjegyzés Noha bármely adatkötési művelet megvalósítható pusztán imperatív kód használatával, a következő példák a XAML segítségével konvertálják az adattípusokat. Ehhez szükség van egyedi erőforrások bevonására, amelyeket részletesen a 30. fejezetben vizsgálunk meg. Ne nyugtalankodjunk, ha a markup egy része nem tűnik ismerősnek.

Tételezzük fel, hogy egész számokat szeretnénk megjeleníteni a `TextBox` vezérlőelemben, így az alábbi osztálytípust hozhatjuk létre (mindenképpen importáljuk a `System.Windows.Data` névteret a definiáló fájlban):

```
class MyDoubleConverter : IValueConverter
{
  public object Convert(object value, Type targetType,
                        object parameter,
  System.Globalization.CultureInfo culture)
  {
    // A double érték konvertálása egész számra.
    double v = (double)value;
    return (int)v;
  }

  public object ConvertBack(object value, Type targetType,
                      object parameter,
                      System.Globalization.CultureInfo culture)
  {
    // A bejövő értéket közvetlenül visszaadjuk.
    // Ezt használjuk a kétirányú kötésekhez
    // a példánkban, amikor a felhasználó Tab billentyűvel
    // kilép a TextBox objektumból.
    return value;
  }
}
```

Amikor az értéket átadjuk a forrásból (a ScrollBarból) a célnak (a TextBox `Text` tulajdonságának), a rendszer a `Convert()` metódust hívja meg. Míg több bejövő paramétert kapunk, ehhez az átalakításhoz csak a bejövő `objectet` kell kezelnünk, amely az aktuális `double` érték. Egyszerűen egész számra kasztoljuk a típust, és visszaadjuk az új számot.

A `ConvertBack()` metódust hívjuk, ha a forrás megkapja az értéket a céltól (ha engedélyeztük a kétirányú kötésmódot). Itt egyszerűen közvetlenül visszaadjuk az értéket. Ezáltal beírhatunk egy lebegőpontos értéket (pl. 99,9) a `TextBox` elembe, és automatikusan konvertálhatjuk egész számmá (99), ha a felhasználó a Tab billentyűvel kilép a vezérlőelemből. Ez a „szabad" konverzió annak köszönhető, hogy a `Convert()` metódust a `ConvertBack()` meghívása után a rendszer még egyszer meghívja. Ha egyszerűen `null` értéket adnánk vissza a `ConvertBack()` metódusból, akkor úgy tűnne, hogy a kötés nincs szinkronban, mivel a szövegdoboz még mindig lebegőpontos számot mutatna!

Most, hogy az osztály a helyére került, vizsgáljuk meg a következő XAML-módosításokat, amelyek kihasználják az egyedi átalakító osztályunkat az adatok megjelenítéséhez a TextBoxban:

```
<Window x:Class="SimpleDataBinding.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

  <!-- Definiálni kell egy CLR-névteret, hogy hozzáférjünk
       a típusunkhoz -->
  xmlns:myConverters ="clr-namespace:SimpleDataBinding"

  Title="Simple Data Binding" Height="334" Width="288"
  WindowStartupLocation="CenterScreen">

  <!-- Az erőforrásszótárak lehetővé teszik olyan objektumok
       definiálását, amelyek a kulcsuk alapján megszerezhetők.
       További részletek a 30. fejezetben -->
  <Window.Resources>
    <myConverters:MyDoubleConverter x:Key="DoubleConverter"/>
  </Window.Resources>

  <!-- A panel beállítja az DataContext-et a görgetősáv-
       objektumra -->
    <StackPanel Width="250"
                DataContext = "{Binding ElementName=mySB}">

      <Label Content="Move the scroll bar to see the
                      current value"/>

      <ScrollBar Orientation="Horizontal" Height="30" Name="mySB"
                 Maximum = "100" LargeChange="1" SmallChange="1"/>

    <!-- Figyeljük meg, hogy a {Binding} kiterjesztés most beállítja
         a Converter tulajdonságot -->
    <TextBox Height="30" BorderBrush="Blue"
             BorderThickness="2" Name="txtThumbValue"
             Text = "{Binding Path=Value,
             Converter={StaticResource DoubleConverter}}"/>

    <Button Content="Click" Height="200"
    FontSize = "{Binding Path=Value}"/>

  </StackPanel>
</Window>
```

A fenti példában meghatároztunk olyan egyedi XML-névteret, amely leképezhető a projektünk gyökérnévterébe (lásd a 28. fejezetet), hozzáadtuk a Windows típus erőforrásszótárához a `MyDoubleConverter` típusunk egy példányát, amelyet később megszerezhetünk a XMAL-fájlban a `DoubleConverter` kulcsnév révén. A TextBox `Text` tulajdonságát módosítottuk, hogy a `MyDoubleConvert` típusunkat használja, hozzárendeltük a `Converter` tulajdonságot egy másik, `StaticResource` nevű jelölő kiterjesztéshez.

A WPF erőforrásrendszerének további jellemzőit a 30. fejezetben találhatjuk. Mindenesetre, ha futtatnánk az alkalmazásunkat, akkor azt látnánk, hogy csak egész számok jelennek meg a `TextBox` objektumban.

Konvertálás különböző adattípusok között

Az `IValueConverter` interfész implementációja használható bármilyen, adattípusok közötti átalakításra, még akkor is, ha úgy tűnik, hogy azok nem kapcsolódnak egymáshoz. Valójában a `ScrollBar` csúszkájának aktuális értéke használható arra, hogy visszaadjon bármilyen objektumtípust, hogy hozzákapcsolhassuk egy függőségi tulajdonsághoz. Tekintsük meg az alábbi `ColorConverter` típust, amely a csúszka értékét használja, hogy visszaadja az új, zöld `SolidColorBrush` típust (155 és 255 közötti zöld értékkel):

```
class MyColorConverter : IValueConverter
{
  public object Convert(object value, Type targetType,
                        object parameter,
                        System.Globalization.CultureInfo culture)
  {
    // A csúszka értékének használata egy változó zöld ecset
    // készítéséhez.
    double d = (double)value;
    byte v = (byte)d;

    Color color = new Color();
    color.A = 255;
    color.G = (byte) (155 + v);
    return new SolidColorBrush(color);
  }

  public object ConvertBack(object value, Type targetType,
                        object parameter,
                        System.Globalization.CultureInfo culture)
  {
    return value;
  }
}
```

Ha a következők szerint hozzáadnánk egy új tagot az erőforrásszótárunkhoz:

```
<Window.Resources>
  <myConverters:MyDoubleConverter x:Key="DoubleConverter"/>
  <myConverters:MyColorConverter x:Key="ColorConverter"/>
</Window.Resources>
```

akkor használhatnánk a kulcsnevet a `Button` típusunk `Background` tulajdonságának beállításához:

```
<Button Content="Click" Height="200"
        FontSize = "{Binding Path=Value}"
        Background= "{Binding Path=Value,
        Converter={StaticResource ColorConverter}}"/>
```

Ha megint futtatnánk az alkalmazásunkat, akkor a `Button` színe biztosan változna a görgetősáv pozíciója alapján. Hogy a WPF-adatkötések vizsgálatát befejezzük, nézzük meg, hogyan képezhetünk le egyedi objektumokat és XML-dokumentum adatokat a felhasználóifelület-rétegünkre.

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

Kötés egyedi objektumokhoz

Az adatkötés következő típusa, amelyet megvizsgálunk, annak módja, ahogyan az egyedi objektumok tulajdonságait a felhasználói felületünkhöz kapcsolhatjuk. Kezdjük egy új WPF-alkalmazás projekt létrehozásával (CarViewerApp néven), majd a 28. fejezetben felvázolt lépések segítségével változtassuk meg a kezdeti Window1 típusunk nevét MainWindow-ra. Ezután kezeljük a MainWindow `Loaded` eseményét, és módosítsuk a `<Grid>` definícióját, hogy két sort és két oszlopot tartalmazzon:

```
<Window x:Class="CarViewerApp.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Car Viewer Application" Height="294" Width="502"
  ResizeMode="NoResize" WindowStartupLocation="CenterScreen"
  Loaded="Window_Loaded">
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="200"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto"/>
    <RowDefinition Height="*"/>
  </Grid.RowDefinitions>
</Grid>
</Window>
```

A <Grid> első sora tartalmaz egy menürendszert File menüvel, amelyben két almenü van (Add New Car és Exit). Vegyük észre, hogy kezeljük minden almenü kattintási eseményét, és hozzárendelünk egy „beviteli mozdulatot" az Exit menühöz, lehetővé téve az elem aktiválását, ha a felhasználó megnyomja az Alt + F4 billentyűkombinációt. Végül figyeljük meg, hogy a Grid.ColumnSpan értéke 2, amely lehetővé teszi a menürendszer elhelyezését az első sor celláiban.

```
<!-- Menüsáv -->
<DockPanel
  Grid.Column="0"
  Grid.ColumnSpan="2"
  Grid.Row="0">
  <Menu DockPanel.Dock ="Top" HorizontalAlignment="Left"
        Background="White">
    <MenuItem Header="File">
      <MenuItem Header="New Car" Click="AddNewCarWizard"/>
      <Separator />
      <MenuItem Header="Exit" InputGestureText="Alt-F4"
                Click="ExitApplication"/>
    </MenuItem>
  </Menu>
</DockPanel>
```

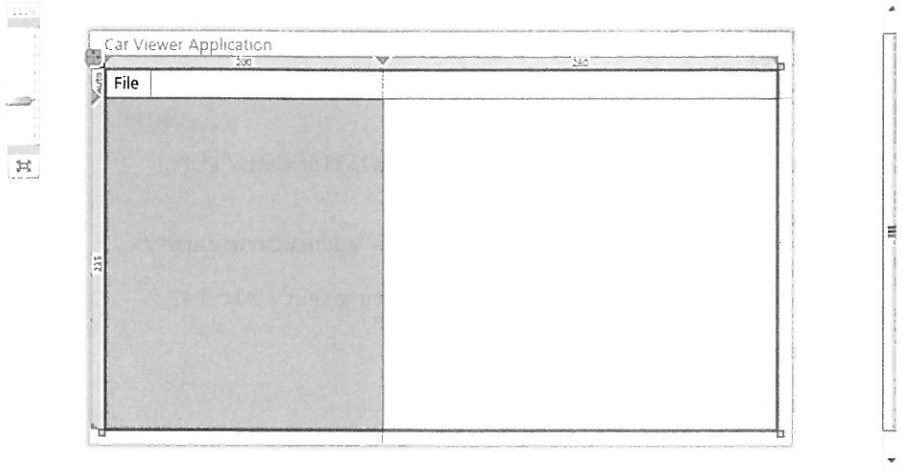
A <Grid> fennmaradó bal oldali része egy ListBox típust tartalmazó <DockPanel>, míg a jobb oldali rész egyetlen TextBlock típust foglal magában. A ListBox típus lesz végül az egyedi objektumok gyűjteményét magába foglaló adatkötési művelet célja, úgyhogy állítsuk be az ItemsSource tulajdonságot a {Binding} jelölő kiterjesztésre (a kötés forrását hamarosan kódban adjuk meg). Ahogy a felhasználó kiválaszt egy elemet a ListBox típusból, rögzítjük a SelectionChanged eseményt, hogy frissíthessük a tartalmat a TextBlock típusban. Itt van a fennmaradó típusok definíciója:

```
<!-- A rács bal oldali ablaktáblája -->
<ListBox Grid.Column="0"
         Grid.Row="2" Name="allCars"
         SelectionChanged="ListItemSelected"
         Background="LightBlue" ItemsSource="{Binding}">
</ListBox>

<!-- A rács jobb oldali ablaktáblája -->
<TextBlock Name="txtCarStats" Background="LightYellow"
           Grid.Column="1" Grid.Row="2"/>
```

Ezen a ponton az ablak felhasználói felületének úgy kellene kinéznie, mint a 29.32. ábrán látható elemnek.

Mielőtt megvalósítanánk az adatkötési logikát, véglegesítsük a File > Exit menü kezelőjét az alábbiak szerint:

```
private void ExitApplication(object sender, RoutedEventArgs e)
{
  Application.Current.Shutdown();
}
```



29.32. ábra: *A főablakunk felhasználói felülete*

Az ObservableCollection<T> típus használata

A .NET 3.0 bemutatott egy új gyűjteménytípust a `System.Collections.ObjectModel` névtéren belül, `ObservableCollection<T>` névvel. Ezen típus használatának előnye, hogy amikor tartalma megváltozik, figyelmeztetést küld az érdekelteknek, például az adatkötési művelet céljának. Szúrjunk be új C#-fájlt az alkalmazásunkba, amely a `CarList` nevű osztályt definiálja, amely bővíti az `ObservableCollection<T>` típust, ahol `T` valójában `Car` típusú. A `Car` típus ezen verziója a C# automatikus tulajdonságait használja alap állapotadatok meghatározására (amelyeket egyedi konstruktor használatával lehet beállítani), valamint biztosítja a `ToString()` metódus megfelelő implementációját:

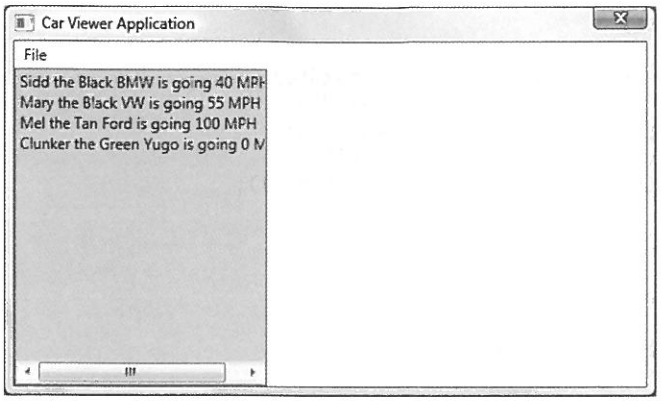
```
using System;
using System.Collections.ObjectModel;

namespace CarViewerApp
{
  public class CarList : ObservableCollection<Car>
  {
```

```
    public CarList()
    {
      // Néhány bejegyzés hozzáadása a listához.
      Add(new Car(40, "BMW", "Black", "Sidd"));
      Add(new Car(55, "VW", "Black", "Mary"));
      Add(new Car(100, "Ford", "Tan", "Mel"));
      Add(new Car(0, "Yugo", "Green", "Clunker"));
    }
  }

  public class Car
  {
    public int Speed { get; set; }
    public string Make { get; set; }
    public string Color { get; set; }
    public string PetName { get; set; }
    public Car(int speed, string make, string color, string name)
    {
      Speed = speed; Make = make; Color = color; PetName = name;
    }
    public Car(){}

    public override string ToString()
    {
      return string.Format("{0} the {1} {2} is going {3} MPH",
             PetName, Color, Make, Speed);
    }
  }
}
```

Most nyissuk meg a `MainWindow` osztályunk kódfájlját, és definiáljunk egy `CarList` típusú tagváltozót `MyCars` névvel. A `Window` típus `Loaded` eseménykezelőjén belül állítsuk be az `allCars` `ListBox` `DataContext` tulajdonságát a `myCars` objektumra (jusson eszünkbe, hogy ezt az értéket nem XAML-ben állítottuk be a `{Binding}` kiterjesztéssel, ezért, hogy gyorsítsunk, ezt most C#-kód használatával tesszük meg):

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  // Az adatkontextus beállítása.
  allCars.DataContext = myCars;
}
```

Most már futtathatjuk az alkalmazásunkat, és láthatjuk a ListBox típust, amely tartalmazza a `ToString()` értékeket minden `Car` objektum számára az egyedi `ObservableCollection<T>` típusban (lásd a 29.33. ábrát).



29.33. ábra: *A kezdeti adatkötés művelet*

Egyedi adatsablon készítése

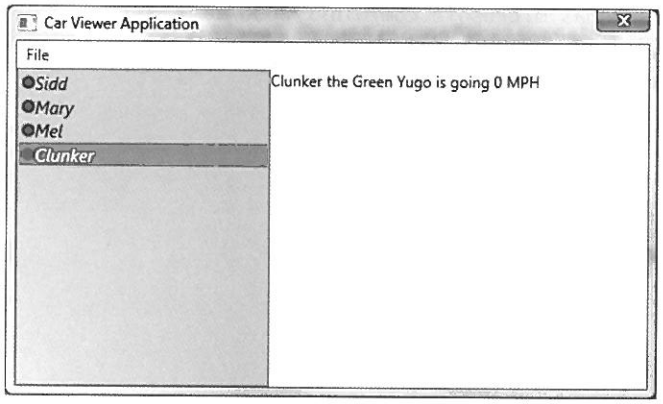
A `ListBox` jelenleg minden elemet megjelenít a `CarList` objektumban; mivel azonban nem adtunk meg kötési útvonalat, ezért minden listabejegyzés egyszerűen a `ToString()` meghívásának az eredménye az alobjektumokon. Mivel már megvizsgáltuk, hogyan állapítsunk meg egyszerű kötési útvonalakat, ezúttal egyedi *adatsablont* készítünk. Az adatsablon használható arra, hogy értesítse egy adatkötési művelet célját arról, hogyan jelenítse meg a hozzá kapcsolt adatokat. A sablonunk a `ListBox` minden elemét feltölti olyan `<StackPanel>` típussal, amely egy `Ellipse` objektumból és egy `TextBlock` típusból áll, amelyet hozzákötöttünk a `CarList` típus összes elemének `PetName` tulajdonságához. Itt van a `ListBox` típus módosított markupja.

```
<ListBox Grid.Column="0"
         Grid.Row="2" Name="allCars"
         SelectionChanged="ListItemSelected"
         Background="LightBlue" ItemsSource="{Binding}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <Ellipse Height="10" Width="10" Fill="Blue"/>
        <TextBlock FontStyle="Italic" FontSize="14"
                   Text="{Binding Path=PetName}"/>
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Itt csatoljuk a `<DataTemplate>` sablonunkat a `ListBox` típushoz a `<ListBox.ItemTemplate>` elem segítségével. Mielőtt megtekintenénk az adatsablon eredményét, implementáljuk a `ListBox` típusunk `SelectionChanged` kezelőjét, hogy megjelenítsük az aktuális kijelölés `ToString()` metódusát a leginkább jobb oldali `TextBlock` típusban:

```
private void ListItemSelected(object sender,
                              SelectionChangedEventArgs e)
{
  // A megfelelő autó beolvasása az ObservableCollection
  // gyűjteményből a listadobozból kiválasztott eleme alapján.
  // Majd a ToString() metódus meghívása.
  txtCarStats.Text = myCars[allCars.SelectedIndex].ToString();
}
```

Ezzel a módosítással most már stílusosabb külsőt kapott az adatok megjelenítése (lásd a 29.34. ábrát).



29.34. ábra: *Adatkötés egyedi adatsablonnal*

A felhasználói felület elemeinek kötése XML-dokumentumokhoz

A következő feladat olyan egyedi párbeszédablak készítése, amely adatkötést használ egy külső XML-fájl tartalmának megjelenítéséhez egy stilizált `ListView` objektumban. Először szúrjuk be a 24. fejezet NavigationWithLinqToXml projektjében létrehozott `Inventory.xml` fájlt a Project > Add Existing Item menüpont segítségével. Válasszuk ki az elemet a Solution Explorerben, és a Pro-

perties ablak használatával állítsuk a Copy to Output Directory opciót a Copy Always értékre. Ez biztosítja, hogy az alkalmazásunk lefordítása során az `Inventory.xml` fájlt a rendszer átmásolja a \bin\Debug könyvtárunkba.

Egyedi párbeszédablak készítése

Szúrjunk be új WPF Window típust a projektbe (AddNewCarDialog néven) a Visual Studio 2008 Project > Add Window menüpontjának használatával. Az új Window jeleníti meg az `Inventory.xml` fájl tartalmát egy testre szabott `ListView` típusban az adatkötés segítségével. Az első lépés a XAML megírása, hogy meghatározzuk az új ablak megjelenését és működését. Íme, a teljes markup, az elemzéssel együtt:

```
<Window x:Class="CarViewerApp.AddNewCarDialog"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="AddNewCarDialog" Height="234" Width="529"
  ResizeMode="NoResize" WindowStartupLocation="CenterScreen" >
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="144" />
      <RowDefinition Height="51" />
    </Grid.RowDefinitions>

    <!-- Az XmlDataProvider használata -->
    <Grid.Resources>
      <XmlDataProvider x:Key="CarsXmlDoc"
                       Source="Inventory.xml"/>
    </Grid.Resources>

    <!-- Most készítsünk egy adatrácsot, képezzük le az
         attribútumokat/elemeket az oszlopokra az XPath kifejezések
         használatával -->
    <ListView Name="lstCars" Grid.Row="0"
         ItemsSource="{Binding Source={StaticResource CarsXmlDoc},
         XPath=/Inventory/Car}">
      <ListView.View>
        <GridView>
          <GridViewColumn Width="100" Header="ID"
                  DisplayMemberBinding="{Binding XPath=@carID}"/>
          <GridViewColumn Width="100" Header="Make"
                  DisplayMemberBinding="{Binding XPath=Make}"/>
          <GridViewColumn Width="100" Header="Color"
                  DisplayMemberBinding="{Binding XPath=Color}"/>
          <GridViewColumn Width="150" Header="Pet Name"
                  DisplayMemberBinding="{Binding XPath=PetName}"/>
```

```
        </GridView>
      </ListView.View>
    </ListView>

    <WrapPanel Grid.Row="1">
      <Label Content="Select a Row to Add to your car collection"
             Margin="10" />
      <Button Name="btnOK" Content="OK" Width="80" Height="25"
              Margin="10" IsDefault="True" TabIndex="1"
              Click="btnOK_Click"/>
      <Button Name="btnCancel" Content="Cancel" Width="80"
              Height="25" Margin="10" IsCancel="True" TabIndex="2"/>
    </WrapPanel>
  </Grid>
</Window>
```

Kezdjük az elején, és figyeljük meg, hogy a nyitó `<Window>` elemet a `ResizeMode` attribútum `NoResize` értékre állításával definiáltuk, mivel a legtöbb párbeszédablak nem teszi lehetővé a felhasználó számára az ablak méretének megváltoztatását.

Azon túl, hogy a `<Grid>` típusunkat adott méretű sorokra bontjuk, a következő érdekes pont, hogy a rács erőforrásszótárában elhelyezünk egy `XmlDataProvider` típusú új objektumot. Ezt a típust hozzákapcsolhatjuk külső `*.xml` fájlhoz (vagy egy XML-databkához a XMAL-fájlban) a Source attribútum segítségével. Mivel úgy konfiguráltuk az `Inventory.xml` fájlt, hogy a jelenlegi projektünk alkalmazáskönyvtára tárolja, nem kell foglalkoznunk egy rögzített elérési út kódolásával.

A markup igazán nagy része a `ListView` típus definíciójában található. Először is, figyeljük meg, hogy az `ItemSource` attribútumot hozzárendeltük a `CarsXmlDoc` erőforráshoz, amelyet az `XPath` attribútum használatával minősítettünk. A tapasztalataink alapján tudhatjuk, hogy az `XPath` olyan XML-technológia, amely lekérdezésszerű szintaxis használatával lehetővé teszi a navigálást egy XML-dokumentumon belül. Itt azt mondjuk, hogy a kezdeti adatkötési útvonal az `<Inventory>` gyökér `<Car>` elemével kezdődik.

Ahhoz, hogy tájékoztassuk a `ListView` típust arról, hogy rácsszerű front endet jelenítsen meg, `<ListView.View>` elemet kell használnunk négy `<GridViewColumns>` típusból álló `<GridView>` definiálására. Ezen típusok mindegyike meghatároz egy `Header` értéket (megjelenítési célból), és legfőképp egy `DisplayMemberBinding` adatkötési értéket. Mivel maga a `<ListView>` már megadta, hogy a kezdeti elérési útvonal az XML-dokumentumon belül az `<Inventory>` elem `<Car>` részeleme legyen, ezért minden `XPath` kötés ezt használja kezdőpontként az oszloptípusok esetében.

Az első <GridViewColumn> jeleníti meg a <Car> elem ID attribútumát XPath-specifikus szintaxis használatával az attribútumértékek (@caID) kiszedéséhez. A megmaradó oszlopok egyszerűen tovább minősítik az elérési útvonalat az XML-dokumentumban úgy, hogy a {Binding} markupbővítmény XPath minősítőjének használatával csatolják a következő részelemet.

Végül, de nem utolsósorban, a felhasználói felület befejezéséhez a <Grid> utolsó sorában található a <WrapPanel> típus, amely két Button (és egy leíró Label) típust tartalmaz. Az egyetlen érdeklődésre számot tartó pont, hogy kezeljük az OK gomb kattintási eseményét, valamint használjuk az IsDefault és az IsCancel tulajdonságokat. Ezek állapítják meg, hogy az ablakon lévő gombok közül melyik reagáljon a kattintási eseményre, ha a felhasználó megnyomja az Enter, illetve az Esc billentyűket.

Végül figyeljük meg, hogy ezek a Button típusok megadnak egy TabIndex és egy Margin értéket, az utóbbi lehetővé teszi a <WrapPanel> típusban lévő elemek körüli tér beosztását.

A DialogResult érték hozzárendelése

Mielőtt megjelenítenénk ezt az új párbeszédablakot, implementálnunk kell az OK gomb kattintási eseménykezelőjét. A Windows Formshoz hasonlóan (lásd a 27. fejezetet) a WPF-párbeszédablakok tájékoztathatják a hívót a DialogResult tulajdonság révén, hogy a felhasználó melyik gombra kattintott. Azonban a Windows Forms DialogResult tulajdonságával *ellentétben*, a WPF-modellben ez a tulajdonság egy nullázható logikai értékkel működik, nem pedig egy erősen típusos felsorolással. Ezért, ha tájékoztatni szeretnénk a hívót, hogy a felhasználó alkalmazná a párbeszédablakban lévő adatokat a programban (amit általában egy OK, egy Igen vagy egy Elfogad gombra kattintással jelez), állítsuk be az örökölt DialogResult tulajdonságot true értékre az említett gomb kattintási eseménykezelőjében:

```
private void btnOK_Click(object sender, RoutedEventArgs e)
{
  DialogResult = true;
}
```

Mivel a DialogResult alapértelmezett értéke false, ezért nem kell tennünk semmit, ha a felhasználó a Cancel gombra kattint.