WPF – Binding

Petr Novák / 2021-05-26

Obsah

1 Úvod

Co je to Binding?

- Schopnost automaticky propojit „položky GUI“ a „vlastnosti ve třídě“ (není potřeba programově vkládat / zapisovat data do GUI a zase je programově vyzvedávat / číst).
- Při použití se standardně vytvoří pouze propojení směrem „vlastnost ve třídě“ na „položku v GUI“. Při zobrazení GUI se tedy aktualizuje podle stavu vlastnosti ve třídě (zejména při prvním zobrazení GUI).
- Pokud se nastaví parametr „TwoWay“, tak se rovněž změny položek v GUI (například TextBox, CheckBox, atd.) přenesou / zapíší do vlastností třídy.
- Aby byl „Binding“ správný je potřeba k tomuto uzpůsobit programovou třídu i položky v GUI.

Jak pracuje Binding?

- Do GUI je potřeba zapsat jaká položka GUI je svázána s jakou vlastnosti ve třídě. Tedy hodnota jaké vlastnosti ve třídě se bude automaticky vkládat do jaké položky v GUI, nebo z jaké položky v GUI se bude hodnota automaticky vkládat (zpět) do jaké vlastnosti třídy.
- Vlastnost ve třídě musí mít „get“ pokud je její hodnota automaticky předávána do GUI a rovněž musí mít „set“ pokud je do ní automaticky zapisována hodnota z GUI.
- Při (prvotním) zobrazení GUI se vyzvednou všechny takto propojené hodnoty z vlastností třídy a vloží se do příslušných položek v GUI.

- Pokud je potřeba, aby se položky v GUI aktualizovali automaticky pokaždé, když se programově změní hodnota propojené vlastnosti ve třídě (ne pouze při prvotním zobrazení GUI), je potřeba aby třída implementovala rozhraní „INotifyPropertyChanged“.
- Pokud je potřeba, aby se hodnoty vlastnosti třídy aktualizovaly automaticky, pokud se změní hodnota položky v GUI (například CheckBox) musí být binding nastaven jako „TwoWay“ (avšak ne vždy je to potřeba, mám pocit).

2 Programová třída a její vlastnosti

Binding je tedy vždy svázání mezi (jednou) položkou GUI a (jednou) vlastností v programové třídě. Obě části však samozřejmě musí obsahovat stejný typ hodnoty. Například lze svázat vlastnost typu „bool“ s GUI položkou „CheckBox“, nebo vlastnost typu „string“ s GUI položkou „Label“ / „TextBox“ a podobně. Pokud je hodnota vlastnosti zcela jiného typu, než je potřeba pro položku v GUI je nutno vytvořit patřičný „Converter“. (Například pokud vlastnost typu „bool“ chceme navázat na „Label“ a v něm zobrazovat text „Zapnuto“ / „Vypnuto“. V případě „Label“ je však tuto vazbu možno udělat pouze směrem z vlastnosti třídy do položky v GUI.)
Pro programovou třídu a její vlastnosti platí tato pravidla:

- Vlastnost ve třídě musí být „public“.
- Pokud je hodnota z vlastnosti předávána do položky GUI tak musí vlastnost obsahovat „(public) get“.
- Pokud je hodnota rovněž předávána z položky GUI do vlastnosti třídy, tak musí vlastnost obsahovat „(public) set“.
- Vlastnost musí mít stejný typ hodnoty, jako je akceptován položkou GUI. Některé položky v GUI mohou akceptovat i několik typů hodnot (například i „object“, který pomocí „ToString()“ převedou na „String“).
- Pokud má vlastnost (zcela) jiný typ hodnoty, který nelze automaticky převést na hodnotu akceptovanou položkou GUI je potřeba vytvořit „Converter“, který převádí typ hodnoty z vlastnosti třídy na typ hodnoty do položky v GUI (případně i naopak).

V tomto případě budou hodnoty vlastností vyzvednuty pouze jednou při vytvoření / zobrazení GUI. Pokud bude hodnota vlastnosti v programové třídě později změněna (nastavena) na jinou hodnotu, tak nedojde k aktualizaci odpovídající položky v GUI. Aby tohoto bylo dosaženo tak musí třída obsahující (později měněnou) vlastnost implementovat interface „`INotifyPropertyChanged`“ a její metodu „`OnPropertyChanged(string)`“. Pokud programový kód změní hodnotu některé vlastnosti ve třídě (například s názvem „Stav“), tak musí zavolat metodu „`OnPropertyChanged(string)`“ (například tedy `OnPropertyChanged(„Stav“)`), aby bylo GUI informováno jakou položku je potřeba aktualizovat. Příklad může být následující:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

// trida pro polozky …
// (implementuje interface „INotifyPropertyChanged“)
public class SetupClass : INotifyPropertyChanged
{
  // vlastnost predavana do/z GUI (obsahuje „get“/„set“)
  private string text = 0;
  public string Text
  {
```

```
    // pro cteni aktuální hodnoty smerem do GUI
    get { return text; }
    // pri zapisu hodnoty z proramu je nutno informovat GUI o jeji zmene
    set { text = value; OnPropertyChanged("Text"); }
  }

  // vlastnost predavana do/z GUI (obsahuje „get“/„set“)
  private bool zapnuto = 0;
  public bool Zapnuto
  {
    // pro cteni aktuální hodnoty smerem do GUI
    get { return zapnuto; }
    // pri zapisu hodnoty z proramu je nutno informovat GUI o jeji zmene
    set { zapnuto = value; OnPropertyChanged("Zapnuto"); }
  }


  // --- nutna cast pro aktualizaci GUI ---

  // pro vynuceni aktualizace zobrazeni v GUI
  public event PropertyChangedEventHandler PropertyChanged;
  protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
     { PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName)); }
}
```

3 GUI a XAML

Když je vytvořena programová třída a její vlastnosti, je potřeba tyto vlastnosti propojit na položky v GUI. Lze říci, že lze použít binding na jakýkoli parametr položky v GUI. Tedy například u „Label“ lze použít binding na „Content“ (zobrazený text) i „Visibility“ (viditelnost) nebo „Color“ (barva). Nejzákladnější zápis je stylem:

```
<Label Content="{Binding Text}">
```

Lze tedy použít například:

```
<Label Content="{Binding Text}" Visibility="{Binding Zobrazeni}">
```

Z programové třídy se tedy hodnota vlastnosti „Text“ vloží do „Content“ a hodnota vlastnosti „Zobrazení“ do „Visibility“ (samozřejmě obě vlastnosti musí mít odpovídající typy hodnot). Nyní tedy kdykoliv programový kód nastaví novou hodnotu vlastnosti „Text“, například stylem:

mojeTrida.Text = „NovyText“;

tak se její nová hodnota (textový řetězec) okamžitě aktualizuje / zobrazí ve svázané položce v GUI.
Další příklad je svázání vlastnosti „Zapnuto“ s „CheckBox“ a to následovně:

```
<CheckBox IsChecked="{Binding Path=Zapnuto, Mode=TwoWay}" >
```

Pokud je při binding použito více parametrů je dobré je pojmenovávat (jsou odděleny čárkou). Proto u názvu vlastnosti je potřeba uvádět „Path“, což je cesta k vlastnosti ve třídy (v podstatě název vlastnosti). Další parametr je zde „Mode“. Jeho výchozí hodnota je „OneWay“ určující, že se pouze

propojí vlastnost třídy směrem na položku GUI. Pokud je parametr nastaven na „Mode=TwoWay“, tak rovněž pokud pojde ke změně hodnoty položky v GUI (uživatelem), je tato nová hodnota automaticky uložena zpět do vlastnosti v programové třídě. Toto je velký přínos pro konfiguraci / nastavování hodnot ve třídě pomocí GUI.

4 Svázání kódu s GUI

Vytvoření potřebných vlastností v programové třídě a zápis „binding“ do položek v XAML je sice velmi důležité, ale ještě ne vše. Ještě je potřeba zadat jaká programová třída bude svázána s jakými položkami v GUI. V XAML mohou být totiž vytvořeny bloky / sekce GUI položek a každý tento blok / sekce může být svázána s jinou programovou třídou (tedy jak bloky / sekce v GUI, tak i programové třídy mohou obsahovat stejné názvy vlastností).
Pokud je potřeba s položkami v GUI (XAMLu) svázat pouze jednu programovou třídu, tak stačí použít následující (samozřejmě pouze ve třídě kde je GUI):

```
this.DataContext = instanceTridyObsahujícíVlastnostiProSvazani;
```

„this“ je v podstatě odkaz na GUI této třídy (tato třída obsahuje GUI, tedy XAML) a „`instanceTridyObsahujícíVlastnostiProSvazani`“ je instance třídy, jejích položky budou pomocí bnding svázány s položkami v GUI. Takto lze tedy s GUI svázat pouze jednu programovou třídu.
Pokud je potřeba svázat s GUI více programových tříd, je nutno tyto třídy navázat přímo na bloky / sekce GUI (tedy části XAMLu) na které se mají mapovat vlastnosti těchto tříd. Příkladem je GUI obsahující dva bloky pro text:

```
<Border Name="bPrvni">
  <Label Content="{Binding Text}">
<Border>
<Border Name="bDruhy">
  <Label Content="{Binding Text}">
<Border>
```

Pokud máme například dvě instance stejného typu třídy (nemusí to takto být samozřejmě, může jít o zcela jiné třídy), obě tedy obsahují například vlastnost“ Text“ a požadujeme, aby se obsah jedné zobrazoval v prvním bloku a obsah druhé ve druhém bloku, je nutno postupovat následovně:

```
this.bPrvni.DataContext = instanceTridyPrvni;
this.bDruhy.DataContext = instanceTridyDruhy;
```

Všechny položky GUI obsažené v bloku „Border“ se jménem „bPrvní“ jsou navázány na vlastnosti instance „`instanceTridyPrvni`“. Současně všechny položky GUI obsažené v bloku „Border“ se jménem „bDruhy“ jsou navázány na vlastnosti instance „`instanceTridyDruhy`“. Vždy je tedy nutno použít „`DataContext`“ takové položky v GUI, která je nadřazena položkám obsahující „binding“ pro vlastnosti požadované programové třídy. Pokud je použita jedna programová třída lze v podstatě použít „`DataContext`“ celého dialogu. Takto lze vytvořil libovolný počet bloků v GUI a k nim nezávisle navázat libovolné programové třídy. Příkladem může být nastavení aplikace, které je uloženo v několika třídách a každá třída je navázána na jiný blok položek v GUI pro určitý typ nastavení v aplikaci (nastavení vzhledu, nastavení typu ukládání, nastavení chování aplikace …).

5 Binding pouze v rámci prvků XAML

Binding není pouze pro svázání vlastností programové třídy s položkami v GUI, ale rovněž jej lze použít pro svázání některých položek v GUI mezi sebou (nastavení jedné ovlivňuje činnost jiných). Představme si případ, kdy je potřeba zablokovat nějaké (jiné) položky podle toho zda je (určitý) „CheckBox“ zatržen (tedy povolen) nebo nezatržen (tedy nepovolen). Zápis je následující:

```
<CheckBox Name="cbAdresa" >
<TextBox IsEnabled="{Binding IsChecked, ElementName=cbAdresa}">
```

TextBox obsahuje vlastnost „IsEnabled“, která povoluje / blokuje zápis v něm obsaženého textu (v podstatě povoluje/blokuje celý „TextBox“). Ukázka propojuje vlastnost „IsChecked“ položky „CheckBox“ s názvem „cbAdresa“ na vlastnost „IsEnabled“ do „TextBox“, jde tedy o předávání hodnoty z „CheckBox.IsChecked“ do „TextBox.IsEnabled“. Tedy pokud je „CheckBox“ zatržen, tak jeho vlastnost „IsChecked“ obsahuje hodnotu „True“ a ta se promítne do vlastnosti „IsEnabled“ u „TextBox“ a tím je „TextBox“ povolen (je tedy povolen zápis jeho textu). Naopak pokud je „CheckBox“ nezatržen, tak jeho vlastnost „IsChecked“ obsahuje hodnotu „False“ a tato hodnota přes „IsEnabled“ v „TextBox“ blokuje zápis jeho textu.
Další příklad může být následující:

```
<CheckBox Name="cbAdresa" >
<TextBox Visibility="{Binding IsChecked, ElementName=cbAdresa,
       Converter={StaticResource BoolToVisibilityConverter}}">
```

Nyní pokud je „CheckBox“ zatržen, tak je „TextBox“ zobrazen a naopak. Protože parametr „CheckBox.IsChecked“ je typu „bool“ a parametr „TextBox.Visibility“ je typu „Visibility“, tak je nutno použít konvertor z hodnoty typu „bool“ na hodnotu typu „Visibilty“ (popsáno dále).

6 Možnosti „binding“ s textem

Pokud se „binding“ účastní vlastnost typu „string“ a položka v GUI obsahuje vlastnost „Text“ (nikoli „Content“) jako například „TextBlock“, lze v binding využít „StringFormát“. Příkladem může být:

```
<!-- trida dialogu -->
<Window x:Class="..."
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       xmlns:system="clr-namespace:System;assembly=mscorlib"
       Name="wnd">
  <!-- polozky budou pod sesebou -->
  <StackPanel Margin="10">
    <!-- zobrazeni aktualni sirky dialogu -->
    <TextBlock Text="{Binding ElementName=wnd, Path=ActualWidth,
StringFormat=Window width: {0:#,#.0}}" />
    <!-- zobrazeni aktualni vysky dialogu -->
    <TextBlock Text="{Binding ElementName=wnd, Path=ActualHeight,
StringFormat=Window height: {0:C}}" />
    <!-- zobrazeni aktualniho data -->
    <TextBlock Text="{Binding Source={x:Static system:DateTime.Now},
StringFormat=Date: {0:dddd, MMMM dd}}" />
    <!-- zobrazeni aktualniho data -->
    <TextBlock Text="{Binding Source={x:Static system:DateTime.Now},
StringFormat=Time: {0:HH:mm}}" />
  </StackPanel>
```

```
</Window>
```

Pokud není použit žádný úvodní doplňkový text tak je zápis následující (úvodní „{}“ znamená, že není použit žádný úvodní text):

```
<TextBlock Text="{..., StringFormat={}{0:#,#.0}}" />
```

Poznámky:

- Pokud položka GUI neobsahuje „Text“, ale například „Content“, lze použít „ContentStringFormat“. Pozor však na skutečnost, že „ContentStringFormat“ není součástí „binding“, ale je to samostatná vlastnost pro „Label“.
  ```
  <Label Content="{Binding Path=...}" ContentStringFormat="Projects: {0}" />
  <Label Content="{Binding Path=...}" ContentStringFormat="{}{0}" />
  ```
- Dá se říci, že většinu textového formátování lze vyřešit i pomocí „Converter“, kdy je vstupní parametr doplněn potřebným textem a vrácen jako jeden výstupní text pro zobrazení.
- Pokud by „binding“ hlásil nějaké chyby, občas je potřeba zadat „Mode=OneWay“, protože se jedná skutečně pouze o jednosměrný „binding“ (s textem).

7 Multi-binding

Někdy je vhodné výstup do GUI složit současně z hodnot několika vlastností z programové třídy. V minulé kapitole byl příklad jak tohoto dosáhnout pomocí „TextBlock“ a jeho „Run“. Jsou však i jiné možnosti, například:

```
<!-- prvek v GUI bude „TextBlock“ -->
<TextBlock>
  <!-- jeho vlastnost „Text“ -->
  <TextBlock.Text>
    <!-- bude složena z několika „binding(s)“ pomoci „StringFormat“ -->
    <MultiBinding StringFormat="{}{0} + {1}">
      <!-- prvni honota pro {0} je z vlastnosti „Name“ -->
      <Binding Path="Name" />
      <!-- druha honota pro {1} je z vlastnosti „ID“ -->
      <Binding Path="ID" />
    </MultiBinding>
  </TextBlock.Text>
</TextBlock>
```

Bude tedy zobrazen text obsahující dvě hodnoty, první je vyzvednuta z vlastnosti „Name“ a druhá z vlastnosti „ID“.

8 Binding na statickou vlastnost a metodu

(Neříkám že jsou zde uvedeny nejlepší způsoby, určitě jich existuje mnoho jiných a postupným vývojem .NETu se rovněž zjednodušují a obsahují větší možnosti.)
Někdy je potřeba použít „binding“ na statickou vlastnost nebo metodu ve třídě. Například pokud je potřeba naplnit **ComboBox** hodnotami / možnostmi na výběr.

Na statickou vlastnost (ve statické / nestatické třídě)
Nejprve vytvořit statickou / nestatickou třídu se statickou vlastností (třída i vlastnost musí být **public**). Například následovně:

```
namespace TestWPF
{
    // trida obsahujici statickou metodu
    public (static) class TestClassStatic
    {
        // staticka vlastnost navazovana do XAML
        public static List<int> MyData
        {
            // vracene hodnoty
            get { return new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; }
        }
    }
}
```

Stačí přímo v XAML vytvořit binding na vlastnost **MyData** takto:

```
xmlns:local="clr-namespace:TestWPF"

<ComboBox ItemsSource="{Binding Path=(local:TestClassStatic.MyData), Mode=OneWay}" />
```

Na statickou metodu (ve statické třídě)

Nejprve vytvořit statickou třídu se statickou metodou (třída i metoda musí být **public**). Například následovně:

```
namespace TestWPF
{
    // trida obsahujici statickou metodu
    public static class TestClassStatic
    {
        // staticka metoda navazovana do XAML
        public static List<int> GetMyData()
        {
            // vracene hodnoty
            return new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        }
    }
}
```

V XAML, v jeho **resources**, je nutno vytvořit **ObjectDataProvider** představující v podstatě převod názvu metody na interní položku v XAML (v **binding** nelze zapsat přímo název metody, pouze vlastnosti):

```
xmlns:local="clr-namespace:TestWPF"

<Window.Resources>
  <ObjectDataProvider x:Key="data"
    ObjectType="{x:Type local:TestClassStatic}"
    MethodName="GetMyData" />
</Window.Resources>
```

Význam je následující. Metoda **GetMyData** ve třídě **TestClassStatic** nacházející se v namespace **local** bude v XAML přístupna pod názvem **data**. Nyní lze použít položku **data** již existující v XAML k binding možností do **ComboBox** takto:

```
<ComboBox ItemsSource="{Binding Source={StaticResource data}}" />
```

Na statickou metodu s parametry (ve statické třídě)

Nejprve vytvořit statickou třídu se statickou metodou (třída i metoda musí být **public**). Například následovně:

```
namespace TestWPF
{
    // trida obsahujici statickou metodu
    public static class TestClassStatic
    {
        // staticka metoda navazovana do XAML
        public static List<int> GetMyDataPrm(int value = 0)
        {
            // vracene hodnoty
            return new List<int>() { value, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        }
    }
}
```

Metoda přijímá pouze jeden vstupní parametr typu **int** a jeho (výchozí) hodnota je uvedena v **ObjectDataProvider.MethodParameters**.

```
<Window.Resources>
  <ObjectDataProvider x:Key="dataPrm"
    ObjectType="{x:Type local:TestClassStatic}"
    MethodName="GetMyDataPrm">
    <ObjectDataProvider.MethodParameters>
      <System:Int32>123</System:Int32>
    </ObjectDataProvider.MethodParameters>
  </ObjectDataProvider>
</Window.Resources>
```

Pokud se tedy zavolá metoda **GetMyDataPrm** tak se jí předá jako parametr hodnota **123**. V XAML bude zápis následující.

```
<ComboBox ItemsSource="{Binding Source={StaticResource dataPrm}}" />
```

V tomto případě pokaždé, když se zavolá metoda **GetMyDataPrm** tak se bude vždy předávat parametr **123**. I když je parametr konstantní, lze jej vhodně využít pro různé WPF dialogy / stránky. V každém dialogu / stránce musí být definován **ObjectDataProvider** a ten může obsahovat jiný parametr. Parametr tedy může specifikovat do jakého dialogu / stránky je výsledek volání metody směřován a podle toho může být upraven.

9 Binding pomocí ICommand

Binding lze použít i pro vstup uživatele a to ne pouze na tlačítka v GUI/XAML, ale rovněž na klávesnici a myš. Nejprve ukázka jak takový biding principiálně pracuje. Jde o jednoduchou demonstraci, kterou lze udělat i pomocí běžného způsobu, ale je vhodná pro vysvětlení činnosti. Nejprve obsah XAML:

```
<!-- hlavni trida GUI -->
<Window x:Class="CustomCommandTest.CommandWindow"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="Custom Command Test" Height="300" Width="300">

  <!-- definovani / seznam „CommandBindings“ pro toto GUI -->
```

```
  <Window.CommandBindings>
    <!-- definice „Command“ s nazvem „Help“ a metod „CanExecute“ a „Execute“ -->
    <CommandBinding Command="Help" CanExecute="HelpCanExecute"
              Executed="HelpExecuted"/>
  </Window.CommandBindings>

  <!-- tlacitko vyuzivajici navazany „Help“ -->
  <Button Command="Help" Content="Help Command Button" />

</Window>
```

Každý „Command“ obsahuje dvě (hlavní) metody „CanExecute“ a „Execute“. První určuje, zda lze „Command“ (právě / aktuálně) použít a druhá určuje, co se stane, když se „Command“ použije / vykoná. V XAML v sekci „CommandBindings“ se definuje název pro „Command“ a jaké skutečné metody se vyvolají právě pro „CanExecute“ a „Execute“. Základní implementace těchto metod je následující (zde umístěny v programovém souboru patřící k uvedenému XAMLu):

```
// povoluje / blokuje tlacitko pro vykonani
private void HelpCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = true;    // vykonani povoleno (nejjednodušší varianta)
    e.Handled = true;       // obsluha metody vykonana
}
// vykonava požadovanou akci
private void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Hey, I'm some help."); // zobrazeni hlaseni
    e.Handled = true;       // obsluha metody vykonana
}
```

Nyní poněkud komplexnější příklad ukazující jak vytvořit binding, část v XAML, část v programovém kódu a rovněž jak jej použít pro klávesnici a myš. Nejprve obsah XAML:

```
<!-- hlavni trida GUI -->
<Window x:Class="CustomCommandTest.CommandWindow"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="Custom Command Test" Height="300" Width="300">

  <!-- definovani / seznam „CommandBindings“ (tlacitka) pro toto GUI -->
  <Window.CommandBindings>
  <!-- definice „Command“ s nazvem „Help“ a metod „CanExecute“ a „Execute“ -->
  <CommandBinding Command="Help" CanExecute="HelpCanExecute" Executed="HelpExecuted"/>
  </Window.CommandBindings>

  <!-- definovani / seznam „InputBindings“ (ne tlacitka) pro toto GUI -->
  <Window.InputBindings>
    <!-- klavesnice - pri „Ctrl+H“ se vykona „Help“ (z „CommandBindings“) -->
    <KeyBinding Command="Help" Key="H" Modifiers="Ctrl"/>
    <!-- mys - pri „LeftDoubleClick“ se vykona „Help“ (z „CommandBindings“) -->
    <MouseBinding Command="Help" MouseAction="LeftDoubleClick" />
  </Window.InputBindings>

  <StackPanel>
    <!-- tlacitko definujici využití „Help“ v XAML -->
    <Button Command="Help" Content="Help Command Button" />
    <!-- tlacitko definujici využití „Help“ v kódu (uvedeno dále) -->
    <Button Content="My Command" x:Name="MyCommandButton" />
  </StackPanel>
```

```
</Window>
```

XAML tedy definuje „Command“ s názvem „Help“ a jeho metody „CanExecute“ a „Execute“. Tento „Help“ přiřazuje tlačítku. Dále definuje gesto klávesnice pro „Ctrl+H“ a to mapuje na „Help“. Rovněž vytváří gesto myši „LeftDoubleClick“ a to mapuje také na „Help“. Tedy tlačítko, klávesnice i myš využívá jeden definovaný „Command“. Nyní programový kód umístění na pozadí k XAML:

```
public partial class CommandWindow : Window
{
  // „Command“ vytvareny v programovem kodu
  public static RoutedCommand MyCommand = new RoutedCommand();

  private bool _helpCanExecute = true;

  public CommandWindow()
  {
    InitializeComponent();

    // vytvoreni „CommandBinding“ v kódu
    CommandBinding cb = new CommandBinding(MyCommand,
              // odkazy na metody „…Execute“ a „…CanExecute“
              MyCommandExecute, MyCommandCanExecute);
    // pridani tohoto „CommandBinding“ do dialogu
    this.CommandBindings.Add(cb);
    // obsluha svazani tohoto „Command“ s tlacitkem na dialogu
    MyCommandButton.Command = MyCommand;

    // vytvoreni „KeyGesture“ pro „Control+M“
    KeyGesture kg = new KeyGesture(Key.M, ModifierKeys.Control);
    // vytvoreni „InputBinding“ z tohoto „Kesture“ a jeho svazani s „MyCommand“
    InputBinding ib = new InputBinding(MyCommand, kg);
    // pridani tohoto „InputBinding“ do dialogu
    this.InputBindings.Add(ib);
  }
  // obsluha „CanExecute“ pro „Help“
  private void HelpCanExecute(object sender, CanExecuteRoutedEventArgs e)
    { e.CanExecute = _helpCanExecute; e.Handled = true; }
  // obsluha „Execute“ pro „Help“
  private void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
    { MessageBox.Show("Hey, I'm some help."); }

  // obsluha „CanExecute“ pro „MyCommand“
  private void MyCommandCanExecute(object sender, CanExecuteRoutedEventArgs e)
    { e.CanExecute = true; }
  // obsluha „Execute“ pro „MyCommand“ (povoluje / blokuje „Help“)
  private void MyCommandExecute(object sender, ExecutedRoutedEventArgs e)
    { MessageBox.Show("My Command!"); _helpCanExecute = !_helpCanExecute; }
```

Zde je programová obsluha pro „Help“ definovaná (pouze) v XAML a pro „MyCommand“ definovaná (pouze) v programovém kódu. Toto demonstruje, jak lze tento binding definovat v XAML i v programovém kódu.
Předešlá demonstrace je sice pěkná, ale lze ji celkem snadno vytvořit i bez „binding“. Nyní ukázka jak tento „binding“ skutečně využít. Představme si, že máme kolekci tříd a ty se zobrazují jako seznam v GUI (třeba pomocí „ListView“ nebo „DataGrid“). U každé položky v GUI chceme mít tlačítko pro nějakou činnost, například nastavit aktuální čas záznamu. Jak tohoto dosáhnout. Nejprve vytvořit základ třídy pro zobrazení hodnot z jejích vlastností v GUI:

```
// jeden zaznam do zobrazeni
```

```
public class OneItem : INotifyPropertyChanged
{
  // vlastnost predavana do/z GUI (obsahuje „get“/„set“)
  private string text1 = 0;
  public string Text1
  {
    // pro cteni aktuální hodnoty smerem do GUI
    get { return text1; }
    // pri zapisu hodnoty z proramu je nutno informovat GUI o jeji zmene
    set { text1 = value; OnPropertyChanged("Text1"); }
  }

  // vlastnost predavana do/z GUI (obsahuje „get“/„set“)
  private string text2 = 0;
  public string Text2
  {
    // pro cteni aktuální hodnoty smerem do GUI
    get { return text2; }
    // pri zapisu hodnoty z proramu je nutno informovat GUI o jeji zmene
    set { text2 = value; OnPropertyChanged("Text2"); }
  }

  // vlastnost predavana do/z GUI (obsahuje „get“/„set“)
  private string aktualniCas = String.Empty;
  public string AktualniCas
  {
    // pro cteni aktuální hodnoty smerem do GUI
    get { return aktualniCas; }
    // pri zapisu hodnoty z proramu je nutno informovat GUI o jeji zmene
    set { aktualniCas = value; OnPropertyChanged("AktualniCas"); }
  }

  // --- nutna cast pro aktualizaci GUI ---

  // pro vynuceni aktualizace zobrazeni v GUI
  public event PropertyChangedEventHandler PropertyChanged;
  protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
     { PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName)); }
}
```

Třída obsahuje vlastnosti „Text1“ a „Text2“ a rovněž vlastnost „AktualniCas“ pro zobrazení aktuálního času (pro jednoduchost) jako řetězec. Část XAMLu pro zobrazení obsahu tříd „OneItem“ jako seznam může být následující (principiálně):

```
<!-- pro posun seznamem -->
<StackPanel Margin="20,40,20,40">
  <!-- seznam pro zobrazeni zaznamu -->
  <ListView x:Name="myListView" ItemsSource="{Binding}" >
    <!-- vzor jednoho zaznamu -->
    <ListView.ItemTemplate>
      <DataTemplate x:Name="dataTemplate">
        <StackPanel Orientation="Horizontal">
          <StackPanel Orientation="Vertical" Margin="0,0,20,0">
            <!-- zobrazeni hodnot z „Text1“ a „Text2“ -->
            <TextBlock Text="{Binding Text1}" FontSize="24" />
            <TextBlock Text="{Binding Text2}" FontSize="24" />
          </StackPanel>
          <!-- zobrazeni hodnoty z „AktualniCas“ -->
          <TextBlock Text="{Binding AktualniCas}" FontSize="24" />
```

```
            <!-- tlacitko vyuzivajici „Command“ (popsano dale) -->
            <Button Content="AktualniCas" Width="100" Height="50" Margin="0,0,20,0"
                Command="{Binding AktualniCasCommand}" CommandParameter="{Binding}" />
          </StackPanel>
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>
</StackPanel>
```

V XAML je definován „ListView“ a vzor pro zobrazení hodnot jednoho záznamu z vlastností „Text1“, „Text2“ a „Aktuální čas“ z instance třídy „OneItem“. Záznam rovněž obsahuje tlačítko „AktualniCas“, jehož činnost bude vysvětlena později. Aby bylo možno využít „AktualniCasCommand“ je potřeba jej nejprve definovat, jeho základ je následující:

```
// trida pro obsluhu udalosti tlacitka
public class AktualniCasClick : ICommand
{
  // pro jednoduchos je akce vždy povolena
  public bool CanExecute(object parameter) { return true; }
  // byl zmene stav „CanExecute“ a je potřeba ji aktualizovat
  public event EventHandler CanExecuteChanged;

  // co se ma za akci vykonat
  public void Execute(object parameter)
    // telo bude vysvetleno pozdeji
    { }
}
```

Tato třída definuje „Command“ s názvem „AktualniCasClick“, který bude navázán na tlačítko v GUI u každého zobrazeného záznamu. Metoda „CanExecute“ vrací „true/false“ zda je možno akci vykonat nebo nikoli. Metoda „Execute“ je vyvolána při stisku tlačítka pro vykonání požadované akce. Lze si všimnout, že poskytuje parametr označený „parameter“, který lze předat z „Button“ v položce „CommandParameter“. V XAML je uveden tento zápis „`CommandParameter="{Binding}"`“. Znamená, že jako parametr se použije právě zobrazovaný záznam instance „OneItem“, tedy odkaz na instanci „OneItem“, který patři tomuto zobrazovanému záznamu obsahujícího tento „Button“.
Nyní je potřeba do třídy „OneItem“ doplnit možnost vazby na „AktualniCasCommand“

```
// „AktualniCasCommnad“ jako vlastnost, která se naváže na „Command“ v „Button“
public ICommand AktualniCasCommand { get; set; }
// v konstruktoru se vytvoři „AktualniCasCommand“ jako instance „AktualniCasClick“
public OneItem()
{
    AktualniCasCommand = new AktualniCasClick();
}
```

Nyní když se stiskne tlačítko u zobrazeného záznamu instance „OneItem“, tak se vyvolá metoda „Execute“ ve třídě „AktualniCasClick“ umístěné v instanci „OneItem“ právě tohoto záznamu. Můžeme tedy doplnit obsah metody pro vykonání akce:

```
// metoda „Execute“ v „AktualniCasClick“ (v instanci „OneItem“)
public async void Execute(object parameter)
{
  // „parameter“ je odkaz na instanci „OneItem“ a nastavi se aktualni cas
  ((OneItem)parameter).AktualniCas = DateTime.Now.ToString();
}
```

Pozor na skutečnost, že třída „AktualniCasClick“ (automaticky) neobsahuje odkaz na instanci třídy „OneItem“ ve které je umístěna. Tohoto lze dosáhnout například předáním odkazu na „OneItem“ v konstruktoru třídy „AktualniCasClick“ a to následovně „new AktualniCasClick(this)“.

Samozřejmě aby vše pracovalo je nutno navázat seznam tříd „OneItem“ v programovém kódu na „ListView“ v GUI/XAML. Například:

```
// kolekce trid „OneItem“
ObservableCollection<OnetItem> items = new ObservableCollection<OneItem>();
// navazani kolekce na „ListView“
myListView.ItemsSource = items;
```

Občas je potřeba do metody „Execute“ předat více parametrů (nejen odkaz na záznam, ale třeba i nějaký jiný stav v zobrazení). Do metody „Execute“ však vstupuje pouze jeden „parameter“, je tedy nutno více parametrů složit do jednoho. Řešení v XAML může být následující:

```
<Button Content="Zoom" Command="{Binding ...">
  <!-- parametr pro „Command“ -->
  <Button.CommandParameter>
    <!-- bude jich obsahovat nekolik -->
    <MultiBinding Converter="{StaticResource YourConverter}">
      <!-- zadani jednotlivych parametru -->
      <Binding Path="Width" ElementName="MyCanvas"/>
      <Binding Path="Height" ElementName="MyCanvas"/>
    </MultiBinding>
  </Button.CommandParameter>
</Button>
```

Tlačítko „Button“ obsahuje složený „CommandParameter“ mající dvě hodnoty (například šířku a výšku nějakého „Canvas“). Aby mohly být tyto dva parametry předány do „Command“ je potřeba vytvořit „MultiValueConverter“ (protože běžně nelze před více parametrů do jednoho) a to například následovně:

```
// converter pro několik vstupních parametrů a jeden výstupní parametr
public class YourConverter : IMultiValueConverter
{
  // metoda pro konverzi
  public object Convert(object[] values, ...)
    // pouze se vytvoří kopie (mělká), obsahuje seznam odkazu predanych parametru
    { return values.Clone(); }
  ...
}
```

Converter zde neobsahuje nic jiného než, že vezme pole vstupních parametrů a to předá do výstupu. Pole samo o sobě představuje v podstatě jednu hodnotu (odkaz na toto pole) a interně obsahuje odkazy na jednotlivé předané parametry. Přesněji řečeno do výstupu předá pole kopií odkazů na předané parametry.
V metodě „Execute“ si lze nyní všechny předané parametry vyzvednout následovně:

```
// metoda „Ecetute“ v ...
public void OnExecute(object parameter)
{
  // přetypování „object“ na „pole parametru“
  var values = (object[])parameter;
  // vyzvednuti parametru podle jejich pozice v poli / seznamu
```

```
  var width = (double)values[0];
  var height = (double)values[1];
}
```

Poznámky:

- Pokud je použit zápis "{Binding}" tak se vždy přiřazuje něco, co je jako hlavní předávané z programového kódu. Z programového kódu je předáván seznam instancí „OneItem“, ovšem do „CommandParameter“ nelze přiřadit seznam, tak se vždy přiřadí pouze jedna položka z tohoto seznamu, která je právě relevantní, tedy zobrazovaná v tomto záznamu.

10 Binding na (nějakou) výchozí hodnotu

Odkaz na celkovou třídu

Pokud se například v **ListBox** / **ListVew** / … vytváří binding na jednotlivé položky pro zobrazení, lze získat i odkaz na celkovou třídu obsahující tyto jednotlivé položky. Zde je uveden příklad tlačítka zobrazeného v nějaké složitější položce například v ListView, kdy při jeho stisku je vyžadováno, aby se předal odkaz na instanci třídy, které patří toto aktuální zobrazení / tlačítko:

```
<!-- v 'Tag' bude ulozen celkovy odkaz na třídu ze seznamu pro 'ListView' -->
<!-- v kodu je nutne jej patricne přetypovat 'Trida trida = (Trida)Button.Tag' -->
<Button Tag="{Binding}" />
```

11 Další možnosti binding

Několik ukázek pro „RelativeSource“. Ten obsahuje několik režimů. Příklady jsou spíše demonstrační pro pochopení, než pro ukázku skutečné činnosti:
Self Znamená odkaz na sebe sama a vzor pro použití je:

“{Binding RelativeSource={RelativeSource **Self**}, Path=PathToProperty}”

Představme si tvorbu čtverce, aby měl skutečně čtvercový tvar (stejnou šířky a výšku) pomocí XAML:

```
<Rectangle Name="rectangle" Height="100"
       Width="{Binding ElementName=rectangle, Path=Height}"/>
```

Nyní lze to samé vytvořit následovně pomocí „Self“. Činnost je velmi jednoduchá, definuje se pevně „Height“ a poté se „Width“ naváže na výšku tohoto prvku. Stačí tedy později měnit / nastavovat pouze „Height“ a „Width“ se upravuje automaticky podle potřeby.

```
<Rectangle Height="100"
       Width="{Binding RelativeSource={RelativeSource Self}, Path=Height}"/>
```

Další příklad je roztažení „TextBlock“ podle „ActualWidth“ je rodiče. „Width“ se naváže na současný prvek a dále na vlastnost „ActualWidth“ jeho rodiče.

```
<TextBlock
   Width="{Binding RelativeSource={RelativeSource Self}, Path=Parent.ActualWidth}"/>
```

FindAncestor Znamená najití nadřazeného prvku / rodiče a je jeho vzor je:

”{Binding RelativeSource={RelativeSource **FindAncestor**,

AncestorType= {x:Type **typeOfAncestor**}, AncestorLevel=2}, Path=PathToProperty}”

Principiální příklad ukazuje jak zobrazit název nějakého nadřazeného prvku podle zadaných parametru: typ, počet úrovní hledání (pro dobré pochopení je vhodné si zkusit změnit „AncestorType“ nebo „AncestorLevel“).

```
<!-- vnoreni nekolika prvku -->
<Canvas Name="Canvas1_Parent0”>
  <Border Name="Border1_Parent1”>
    <Canvas Name="Canvas2_Parent2”>
      <Border Name="Border2_Parent3”>
        <StackPanel Name="StackPanel1_Parent4”>
          <StackPanel Name="StackPanel2_Parent4”>
            <StackPanel Name="StackPanel3_Parent4”>
              <!-- informacni text -->
              <TextBlock FontSize="16” Margin="5” Text="Name of the ancestor"/>
              <!-- nalezeni předka typu „Border“, max. 2 úrovně, vyzvednuti „Name“ -->
              <TextBlock FontSize="16” Margin="50” Width="200”
                Text="{Binding RelativeSource={RelativeSource FindAncestor,
                  AncestorType={x:Type Border}, AncestorLevel = 2}, Path=Name}" />
            </StackPanel>
          </StackPanel>
        </StackPanel>
      </Border>
    </Canvas>
  </Border>
</Canvas>
```

Pozor! Položka `AncestorLevel` neznamená o kolik úrovní (předků) nahoru se hledá. Znamená totiž, kolikátá nalezená shodná úroveň typu se použije. Tedy pokud je požadován první nadřazený zadaný typ i když je v GUI-XAML o několik úrovní výše, tak zde musí být hodnota „1“, nebo lze tuto položku zcela vynechat (výchozí hodnota položky je totiž „1“).

TemplatedParent Využíváno při definování „ControlTemplate“ pokud je potřeba v „ControlTemplate“ použít nějakou vlastnost z prvku na který je „ControlTemplate“ aplikován. Vzor je:

Binding RelativeSource={RelativeSource **TemplatedParent**}, Path=PathToProperty}

Příklad „ControlTemplate“ pro „Text“. Vytvoří se „Canvas“, na něho se vykreslí „Ellipse“ a do ní se zobrazí pouze text z původního „TextBox“.

```
<Window.Resources>
  <ControlTemplate x:Key="template">
    <Canvas>
      <Ellipse Height="100” Width="150” Fill="LightBlue"></Ellipse>
      <ContentPresenter Margin="35” Content =
          "{Binding RelativeSource = {RelativeSource TemplatedParent}, Path=Text}"/>
    </Canvas>
  </ControlTemplate>
</Window.Resources>
<Canvas Name="Parent0”>
  <TextBox Canvas.Left="0” Canvas.Top="0” Width="0” Margin="50” Height="0”
       Text="Hi WPF" Template="{StaticResource template}" FontSize="25”/>
</Canvas>
```

Možných jednoduchým příkladem je „ControlTemplate“ určený pro „Button“. Ten vykreslí „Border“ a do něho umístí (původní) „Button“. Avšak vytvořený „Border“ má takovou barvu okraje / rámečku jakou barvu pozadí má (původní) „Button“.

```
<ControlTemplate TargetType="{x:Type Button}">
    <Border BorderBrush="{Binding
              RelativeSource={RelativeSource TemplatedParent}, Path=Background}">
        <!-- zde se zobrazi „Content“ puvodniho „Button“ (nikoli celý „Button“) -->
        <ContentPresenter />
    </Border>
</ControlTemplate>
```

Previous Využíváno pokud je potřeba vytvořit „binding“ na předchozí hodnotu zadané vlastnosti (tedy hodnotu, kterou měla vlastnost v podstatě v minulém kroku). Vzor je:

“{Binding RelativeSource={RelativeSource **PreviousData**},Path=PathToProperty}”

Příklad využití: Zobrazuji například pomocí „DataTemplate“ postupně hodnoty z nějaké (zřejmě předem nastavené) kolekce. Když zobrazuji hodnotu pro aktuálně vykreslovanou položku, chtěl bych rovněž zobrazit, i o kolik se zvětšila / zmenšila od předešlé vykreslené položky. Mohu tedy vytvořit „binding“ na aktuální hodnotu a předchozí hodnotu, pro ně vytvořit „MultiValueConverter“, který vypočte jejich rozdíl a ten zobrazím s aktuální hodnotou také změnu od předešlé hodnoty.

Zde je uveden velmi jednoduchý příklad bez konvertoru. Nejprve se vytvoří kolekce hodnot pro „ListBox“ a naváže se na něho. Každý záznam v „ListBox“ je tvořen rámečkem obsahujícím dva texty, text aktuálně zobrazované položky a text z minulé / předešlé zobrazované položky. „ListBox“ postupně prochází kolekci hodnot a zobrazuje záznamy. Výsledek zobrazení je tedy: „One“, „Two One“, „Three Two“.

```
<Window x:Class="MyApp.MainWindow"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
       xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
       mc:Ignorable="d" Title="MainWindow" Height="350" Width="525">
  <Window.Resources>
    <!-- kolekce hodnot pro „ListBox“ -->
    <x:Array x:Key="listItems" Type="sys:String"
       xmlns:sys="clr-namespace:System;assembly=mscorlib">
      <sys:String>One</sys:String>
      <sys:String>Two</sys:String>
      <sys:String>Three</sys:String>
    </x:Array>
  </Window.Resources>

  <Grid>
    <!-- navazani kolekce na „ListBox“ -->
    <ListBox ItemsSource="{StaticResource ResourceKey=listItems}">
      <!-- vzor pro zobrazni jednoho zaznamu -->
      <ListBox.ItemTemplate>
        <DataTemplate>
          <!-- texty budou vedle sebe -->
          <StackPanel Orientation="Horizontal">
            <!-- aktualni hodnota -->
            <TextBlock Text="{Binding}"/>
            <!—predesla / minula hodnota -->
```

```
            <TextBlock Text="{Binding RelativeSource={RelativeSource PreviousData}}"/>
          </StackPanel>
        </DataTemplate>
      </ListBox.ItemTemplate>
    </ListBox>
  </Grid>
</Window>
```

12 Změna důležité vlastnosti s binding

Někdy lze zcela ovlivnit zobrazení / chování celého GUI prvku v závislosti na nějaké vlastnosti připojené třídy.

Skrytí pokud neobsahuj žádný text

Pokud vytváříme seznam a nějaké položka neobsahuje text, je tedy prázdná, tak je občas dobré ji nezobrazit, aby nezabírala místo. Lze tedy vytvořit (například styl) aby byla GUI položka zobrazena, pouze pokud obsahuje skutečně nějaký text:

```
<Style x:Key="…" TargetType="Label">
  <Style.Triggers>
    <!-- výchozi stav vlastnosti -->
    <Setter Property="Visibility" Value="Visible" />
    <!-- pokud je obsah 'Content' prazdny -->
    <DataTrigger Binding="{Binding Path=Content, RelativeSource={RelativeSource
            Self}}" Value="">
      <!-- tak se GUI položka vubec nezobrazi -->
      <Setter Property="Visibility" Value="Collapsed" />
    </DataTrigger>
  </Style.Triggers>
</Style>
```