WPF – Controls

Petr Novák / novakpe@fel.cvut.cz / 2021-06-07

Obsah

1 Úvod

Zde jsou poznámky a postřehy pro použití některých prvků (controls) ve WPF. Některé věci nejsou zřejmé na první pohled zcela zřejmé a mohou způsobit mnoho problémů.

2 Button

Pokud je potřeba do **Style** (stylu) pro **Button** vložit **Click** (stejná událost pro několik **Button**) je nutno toto udělat následovně:

```
<Style x:Key="…" TargetType="Button">
  <EventSetter Event="Button.Click" Handler="CmdTestsSetOnClick" />
</Style>
```

3 CheckBox

Pokud je použit „binding“ na **CheckBox**, tak, aby se změna v GUI promítla do proměnné v kódu, je potřeba použít následující:

```
<CheckBox IsChecked="{Binding …, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" />
```

4 ComboBox

Přímé vložení položek do **ComboBox** v XAML lze zapsat následovně:

```
<ComboBox>
  <System:Int32>123</System:Int32>
  <System:Int32>1234</System:Int32>
</ComboBox>
```

5 TextBox a Run

Pokud je potřeba vytvořit text složený z několika částí a pouze některou část měnit v závislosti na vlastnostech programové třídy, zle použít **TextBlock** a v něm **Run** následovně:

```
<TextBlock>
  <Run Text="{Binding CelsiusTemp}"/>
  <Run Text="°C"/>
  <Run Text=" ("/>
  <Run Text="{Binding Fahrenheit}"/>
  <Run Text="°F)"/>
</TextBlock>
```

Indikace / detekce textu v **TextBox** pouze pomoci XAML. Pokud neobsahuje žádný text je pozadí bílé (neplatné), pokud obsahuje nějaký text je pozadí zelené (platné).

```
<!-- prvek 'TextBox' -->
<TextBox Grid.Row="1" Name="tbName" ...>
  <!-- definovani jeho stylu -->
  <TextBox.Style>
    <!-- styl je urcen pro 'TextBox' -->
    <Style TargetType="TextBox">
     <!-- výchozi barva pozadi  -->
     <Setter Property="Background" Value="Green" />
       <!-- udalost pro zmenu stylu -->
       <Style.Triggers>
         <!-- pokud je delka obsazeneho textu '0' (není zadny text / prazdny) -->
         <DataTrigger Binding="{Binding ElementName=tbName, Path=Text.Length,
              Mode=OneWay}" Value="0">
          <!-- nastaví se tato barva pozadi -->
          <Setter Property="Background" Value="White" />
        </DataTrigger>
      </Style.Triggers>
    </Style>
  </TextBox.Style>
</TextBox>
```

Příklad stylu pro změnu barvy číselného textu. Pokud je text / číslo „0“ tak je text černý, pokud je text / číslo různé od „0“, tak má nějakou barvu:

```
<Style x:Key="…" TargetType="Run">
  <!-- vyzchozi barva textu -->
  <Setter Property="Foreground" Value="Green" />
  <Style.Triggers>
    <!-- pokud text obsahuje „0“ -->
    <DataTrigger Binding="{Binding Path=Text, RelativeSource={RelativeSource Self}}"
          Value="0">
      <!-- tak tato barva textu -->
      <Setter Property="Foreground" Value="Black" />
    </DataTrigger>
  </Style.Triggers>
</Style>
```

Poznámky:

- Pokud u **TextBlock** a **Run** budou mezi částmi textu nežádoucí mezery, tak je potřeba všechny XAML tagy napsat na jeden řádek editoru.

6 ListBox

Jednoduchý způsob změny položky / řádku v **ListBox**.

```
<!-- prvek 'TextBox' -->
<ListBox Grid.Row="0" x:Name="lbName" >
  <!-- vzor pro polozku -->
  <ListBox.ItemTemplate>
    <!-- vzor pro data -->
    <DataTemplate>
      <!-- polozky jako bezne texty s temito parametry ... -->
      <!-- samotné 'Binding' znamena vazbu na vychozi datovou hodnotu  -->
      <Label Content="{Binding}" ... />
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

7 ListView

ListView je celkem sofistikovaný GI prvek. Zde jsou ukázány některé jeho schopnosti.

Binding vlastností do GUI zobrazení
Svázání vlastností programové třídy se statickými položkami v GUI je sice velmi užitečné, ale výhoda bindingu je v něčem ještě mnohem sofistikovanějším. Představme si, že máme soubor tříd (například každá nese údaje o člověku) a ty chceme zobrazit v nějakém seznamu v GUI. Požadujeme tedy, když přidáme novou třídu do kolekce, aby se v seznamu v GUI zobrazila / přidala nová položka (blok obsahující soubor položek) a v ní budou zobrazeny údaje z jedné instance programové třídy. Tedy, aby každá strukturovaná položka v seznamu v GUI obsahoval údaje právě z jedné instance programové třídy v kolekci, tedy údaje jednoho člověka. Stručný postup je následující:

- Vytvořit kolekci „`ObservableCollection<...>`“ pro soubor instancí programových tříd (obsahující data). Jde o kolekci, která do GUI zasílá události, pokud je do ní položka přidána, nebo naopak odebrána. Tím se GUI dozví, zda má v seznamu v GUI novou (grafickou) položku vytvořit, nebo ji naopak odebrat.
- Vytvořit třídu obsahující interface „`INotifyPropertyChanged`“ a všechny vlastnosti jejich hodnoty se budou zobrazovat v jedné strukturované položce v seznamu v GUI. O této třídě platí vše, co bylo napsáno dříve.
- V XAMLu vytvořit „`ListView`“ s potřebnými parametry.
- V „`ListView`“ vytvořit „`ListView.ItemTemplate`“ a v něm „`DataTemplate`“. Zde definovat vzhled jedné (složené) grafické GUI položky a jak jsou její GUI položky navázány na vlastnosti programové třídy.
- Nakonec přes „`ItemsSource`“ svázat kolekci „`ObservableCollection<...>`“ z programového kódu s „`ListView`“ v GUI / XAML.

Nyní vše podrobněji. Nejprve vytvoříme třídu „OneItem“ pro uložení dat pro zobrazení, například následovně:

```
// 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"); }
  }

  // --- 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)); }
}
```

Obsahuje pouze dvě vlastnosti, které budou zobrazeny v jednou záznamu v GUI. Implementace interface „INotifyPropertyChanged“ je potřeba pouze pokud se budou hodnoty vlastností v programové třídě měnit i po dobu jejich zobrazení v GUI a je tedy požadavek na jejich aktualizaci v GUI.
Nyní vytvořit kolekcí pro ukládání tříd „OneItem“:

```
// položku lze kdykoli pridat pomoci: items.Add(new OneItem())
ObservableCollection<OneItem> items = new ObservableCollection<OneItem>();
```

Tato kolekce bude později spojena s GUI pomocí řádku („myListView“ je název „ListView“ vytvořeného v GUI / XAML):

```
myListView.ItemsSource = items;
```

V GUI / XAML je tedy potřeba vytvořit „ListView“ (jednoduchá varianta):

```
<!-- seznam pro zobrazeni libovoneho poctu zaznamu -->
<ListView Name="myListView">
  <!-- vzory pro zobrazeni jedne položky -->
  <ListView.ItemTemplate>
    <!-- format zobrazeni dat jedne položky -->
    <DataTemplate>
      <!-- každý zaznam bude v ramecku -->
      <Border Padding="10">
        <!-- polozky v ramecku budou pod sebou -->
        <StackPanel Orientation="Vertical">
          <!-- zde bude zobrazen obsah vlastnosti „Text1“ -->
          <Label Content="{Binding Text1}" />
          <!-- zde bude zobrazen obsah vlastnosti „Text2“ -->
          <Label Content="{Binding Text2}" />
```

```
        </StackPanel >
      </Border>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

Pokud se nyní přidá instance třídy „OneItem“ do seznamu „items“ (samozřejmě i až po zobrazení GUI/dialogu), tak je GUI o tomto informováno, „ListView“ vytvoří nový záznam pro zobrazení podle „`ListView.ItemTemplate`“ a „`DataTemplate`“ a podle nastaveného „binding“ zobrazí požadované hodnoty z vlastností odpovídající instance programové třídy „OneItem“. V oblasti „`DataTemplate`“ lze použít většinu prvků dostupných v XAML a tím vytvořit zobrazení jednoho záznamu zcela na míru. Samozřejmě lze použít i prvky jako „CheckBox“ nebo „TextBox“ a v binding nastavit „Mode=TwoWay“ a tím využít zpětné uložení hodnot z GUI do instance programové třídy, pokud je uživatel v GUI změní. V této ukázce „ListView“ nepoužívá pro zobrazení formát tabulky, tedy sloupce a řádky, ale každý záznam jako zobrazen jako zcela vlastní definovaná grafická položka obsahující soubor GUI/XAML prvků, tedy může jít i několik řádků pro jeden záznam (seznam informací).

Pokud je potřeba obsah „ListView“ zobrazit formou tabulky (sloupce a řádky), tedy každý záznam na jednom řádku obsahující několik sloupců lze použít následující:

```
<ListView Name="myListView">
  <!-- definice zobrazeni seznamu -->
  <ListView.View>
    <!-- seznam bude zobrazen jako tabulka -->
    <GridView>
      <!-- definice sloupcu, nazev do zahlavi, svazani s vlastnosti tridy -->
      <GridViewColumn Header="Name" DisplayMemberBinding="{Binding Name}" />
      <GridViewColumn Header="Age" DisplayMemberBinding="{Binding Age}" />
    </GridView>
  </ListView.View>
</ListView>
```

Když je používán výběr / označení záznamu v „`ListView`“ a jeho čtení / získání například pomocí „myListView.`SelectedItem`“ nebo „myListView.`SelectedValue`“, tak tyto metody vrací sice „object“, ale ve skutečnosti je vrácen odkaz na instanci třídu „OneItem“ což je velkou výhodou (je tedy vrácena přímo vybraná / označená instance třídu „OneItem“ ze seznamu).

„ListView“ umožňuje šikovné vizuální seskupování záznamů v zobrazení podle nějakého klíče (hodnoty jedné vybrané vlastnosti z programové třídy). Postup pro vytvoření je následující. Nejprve je nutno definovat objekt, jenž se bude používat jako zámek seznamu (při jeho změně):

```
private object myCollectionLock = new object();
```

dále navázat seznam „`ObservableCollection<...>`“ na „ListView“ (jako dříve):

```
myListView.ItemsSource = value;
```

Poté stanovit vlastnost z programové třídy, která bude používána jako klíč pro rozdělení záznamů do skupin:

```
// zamek s nazvem „myCollectionLock“ se spoji se seznamem s nazvem „items“
// (jsou priklady kde tento zamek neni)
```

```
BindingOperations.EnableCollectionSynchronization(items, myCollectionLock);
// vytvori se ...pro rozdeleni zaznamu do skupin
CollectionView view =
(CollectionView)CollectionViewSource.GetDefaultView(myListView.ItemsSource);
// pro rozdeleni do skupin se pouzije vlastnost tridy s nazvem „Key“
// (vlastnost „Key“ tedy musí programova trida obsahovat)
PropertyGroupDescription groupDescription = new PropertyGroupDescription("Key");
// prida se popis jak rozdělovat záznamy do skupin
view.GroupDescriptions.Add(groupDescription);
```

Jako lze vytvořit vzor pro zobrazení jednoho datového záznamu v „ListView“, tak lze rovněž vytvořit vzor pro zobrazení názvu skupiny v „ListView“ a to následovně:

```
<ListView >
  <!-- zde muze byt „ListView.ItemTemplate-DataTemplate“ -->

  <!-- styl pro zobrazeni „Group“ -->
  <ListView.GroupStyle>
    <GroupStyle>
      <!-- styl pro zobrazeni nazvu „Group“ -->
      <GroupStyle.HeaderTemplate>
        <!-- nazev „Group“ bude obsahovat nejaka data (datova polozka) -->
        <DataTemplate>
          <!-- nazev „Group“ bude zobrazen jako „Label“ techto parametru -->
          <TextBlock FontWeight="Bold" FontSize="22" Text="{Binding Name}"/>
        </DataTemplate>
      </GroupStyle.HeaderTemplate>
    </GroupStyle>
  </ListView.GroupStyle>
</ListView>
```

V GUI/XAML se vyskytuje „Text="{Binding Name}". Hodnota „Name“ je vytvořena (interně) ve WPF a obsahuje název příslušné skupiny („Group“). Rovněž lze do binding použít hodnotu „ItemCount“ obsahující aktuální počet položek v příslušné skupině.

Různá zobrazení pro různé třídy

ListView umožňuje vytvořit jiné zobrazení položek pro různé typy tříd ve vstupním seznamu. Představme si základní třídu a od ní několik poděděných tříd, například:

```
public class DataBase { }
public class DataPrvni : DataBase { }
public class DataDryhy : DataBase { }
public class DataTreti : DataBase { }
```

Pokud je potřeba vytvořit jiné / odlišné zobrazení pro každou z definovaných tříd, tak bude zápis v **ListView** následující:

```
<ListView … >
       <ListView.Resources>
              <!-- vzor pro zobrazeni 'DataPrvni' -->
              <DataTemplate DataType="{x:Type local:DataPrvni}">
                     <!-- obsah zobrazeni pro 'DataPrvni' -->
              </DataTemplate>
              <!-- vzor pro zobrazeni 'DataDruhy' -->
              <DataTemplate DataType="{x:Type local:DataPrvni}">
                     <!-- obsah zobrazeni pro 'DataDruhy' -->
              </DataTemplate>
```

```
            <!-- vzor pro zobrazeni 'DataTreti' -->
            <DataTemplate DataType="{x:Type local:DataPrvni}">
                  <!-- obsah zobrazeni pro 'DataTreti' -->
            </DataTemplate>
      </ListView.Resources>
</ListView>
```

Takto lze vytvořit zcela odlišné zobrazení pro poněkud odlišná data (například někde texty, někde obrázky, …). Jednotlivé `DataTemplate` mohou být zcela jiné. Jednotlivé třídy je vhodné zastřešit nějakou bázovou, aby je bylo možno dobře vkládat do společného (generického) seznamu. Pro napojení seznamu dat na **ListView** stačí použít **`lvZobrazení.ItemsSource = seznamData.`**

Další informace:

Pokud se definuje pouze jeden **DataTemplate** lze použít (nesmí se uvádět **DataType**, **DataTemplate** je pro všechny typy položek):

```
<ListView.ItemTemplate>
      <DataTemplate>
            …
      </DataTemplate>
</ListView.ItemTemplate>
```

Pokud se definuje více **DataTemplate** je nutno použít (musí se uvádět **DataType** u každého DataTemplate):

```
<ListView.Resources>
      <DataTemplate>
            …
      </DataTemplate>
      <DataTemplate>
            …
      </DataTemplate>
</ListView.Resources>
```

Poznámky:

- ```
  <!-- zrusi mezery za posledni zobrazenou polozkou -->
  <ListView ScrollViewer.CanContentScroll="False">
  ```
- Pokud se v **ListView** nezobrazuje vertikální posuvník **ScrollBar** je nutno v **Grid**, ve kterém je umístěn místo `<RowDefinition Height="Auto"/>` nastavit `<RowDefinition Height="*"/>`
  ```